Value the Recent Past: Approximate Causal Consistency for Partially Replicated Systems

Ta-Yuan Hsu, *Student Member, IEEE* and Ajay D. Kshemkalyani, *Senior Member, IEEE*

Abstract—In wide-area distributed systems, data replication provides fault tolerance and low latency. Causal consistency in such systems is an interesting consistency model. Most existing works assume the data is fully replicated because this greatly simplifies the design of the algorithms to implement causal consistency. Recently, we proposed causal consistency under partial replication because it reduces the number of messages used under a wide range of workloads. One drawback of partial replication is that its meta-data tends to be relatively large when the message size is small. In this paper, we propose an algorithm Approx-Opt-Track which provides approximate causal consistency whereby we can reduce the meta-data at the cost of some violations of causal consistency. The amount of violations can be made arbitrarily small by controlling a tunable parameter, that we call *credits*. We present the analytic data to show the performance of Approx-Opt-Track. We then give simulation results to show the potential benefit of Approx-Opt-Track, viz., its ability to provide almost the same guarantees as causal consistency, at a smaller cost.

Index Terms—Causal consistency, causality, distributed shared memory, partial replication

1 Introduction

Distributed data repositories support a wide range of services, such as social networks. The application tiers of such repositories rely on the storage level to provide low latency, reliable, available data store systems [1], [2], [3]. Such data store systems often use data replication. However, data replication triggers the issue about the consistency of multiple data replicas. With data replication, consistency of data in the face of concurrent reads and updates becomes an important requirement. Different data consistency models place restrictions on the order of reads and writes to the shared store, based on different requirements.

There exists a spectrum of consistency models in distributed shared memory (DSM) systems [4] from the strongest (i.e., linearizability where each operation appears to take effect atomically), to the weakest (i.e., eventual consistency that is purely a liveness guarantee but offers no safety guarantees). DSM systems are subject to the CAP theorem [5], which states that for a replicated, distributed data store, it is possible to achieve at most two of the three features: Consistency of replicas, Availability of Writes, and Partition tolerance. This theorem is the major reason behind the increasing prevalence of eventual consistency, popularized by Amazon's Dynamo [1], which guarantees that if no further updates are made to a data object, all the replicas of this object will eventually converge. However, eventual consistency sacrifices consistency; replicas hosted at different data centers may have different results with different order.

Causal consistency can provide stronger convergence than eventual consistency without sacrificing low latency operations. This has gained interest as a highly attractive consistency model for distributed shared memory systems, [6], [3], [7], [8], [9]. It has been proved that causal consistency is the strongest form of consistency that satisfies low latency, defined as the latency less than the maximum (round-trip) wide-area delay between replicas [10]. Causally consistent DSM systems specify that two operations that are causally related must appear to every user in the same order. This makes web applications more intuitive for users, because actions appear to every user in the correct order. Consider, for example, a user who posts a new photo to his profile in a social network. Then, he comments on the photo on his timeline wall. Without causal consistency, his friends might observe the comment but not see this photo. It requires extra programming efforts to prevent this inconsistent scenario at the application level.

Although there have been several protocols designed for causal consistency in DSM systems [6], [3], [7], [8], [9], these protocols suffer from a common drawback that makes them less efficient in network bandwidth: they need to implement full replication, where each replica has to maintain a copy of all the data (Complete Replication and Propagation—CRP). These protocols become impractical due to the explosion of big data access and the larger numbers of data replicas. Partial replication is a promising paradigm to alleviate this problem. Partial replication protocols only require that each replica node holds a subset of all the data. Although partial replication can avoid taking unnecessary network capacity and hardware resources, extending partial replication protocols to implement causal consistency is full of challenges

- *T.-Y. Hsu is with the Department of Electrical and Computer Engineering, University of Illinois at Chicago, Chicago, IL 60607. E-mail: thsu4@uic.edu.*
- *A.D. Kshemkalyani is with the Department of Computer Science, University of Illinois at Chicago, Chicago, IL 60607-7053. E-mail: ajayk@cs.uic.edu.*

Manuscript received 23 June 2016; revised 15 July 2017; accepted 29 July 2017. Date of publication 15 Aug. 2017; date of current version 8 Dec. 2017. (Corresponding author: Ajay D. Kshemkalyani.)
Recommended for acceptance by B. Ravindran.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2017.2740174

as compared to full replication. This is primarily because partial replication has to track transitive causal dependencies between each pair of processes, which makes the complexity of tracking dependencies higher and the dependency meta-data larger.

Recently, the first partial replication protocol for causal consistency, named the Opt-Track protocol [11], was proposed, by statically binding each client with one of many replica nodes. Opt-Track is adapted from the KS algorithm [12], [13], which aims at reducing the dependency meta-data size and storage cost for causal message ordering in message passing systems.

Simulations to test for the meta-data size of Algorithm Opt-Track in a DSM setting showed that the meta-data size is practically linear in n, where n is the number of client processes/replica nodes [14]. As a special case of Opt-Track, we also proposed in [11], an improvement Opt-Track-CRP for the full-replication case. Simulations [14] showed that Opt-Track-CRP gives about $40\% \sim 60\%$ reduction in the meta-data size compared to the best known CRP protocol, $OptP$, of Baldoni et al. [15]. Although Opt-Track-CRP (for full replication) has lower meta-data overhead than Opt-Track (for partial replication), the net message size (bandwidth consumed, counting the payload file size being written and broadcast/multicast to the replicas) can be lower for Opt-Track for a wide range of file sizes.

In the real world, the characteristics of causal consistency and partial replication are highly suitable to modern social network applications. Data replication costs are an important factor in overall DSM system performance. Partial replication can effectively reduce the replication costs, especially in the multimedia-oriented social networks (e.g., Instagram). Causal consistency improves social user experience since actions appear to everyone in the correct order. Opt-Track provides causal consistency and achieves the cost advantages of partial replication in DSM systems. For large-sized files being written, the net meta-data overhead for Opt-Track is negligible, and the algorithm shows very good performance in terms of the net meta-data overhead as a fraction of the total message size. However, we recognize that for some applications where the data size is very small, such as wall posts in Facebook or Twitter, the size of the meta-data, even for Algorithm Opt-Track, can be a problem. This paper aims to further reduce the size of the meta-data for maintaining causal consistency in partially replicated DSM systems, based on Opt-Track.

1.1 Contributions

(1) We propose the concept of *approximate* causal consistency whereby we can reduce the meta-data at the cost of some possible violations of causal consistency. The amount of violations can be made arbitrarily small by controlling a tunable parameter, which we call *credits*.

(2) We integrate the notion of credits into the Opt-Track algorithm, to give an algorithm Approx-Opt-Track that can fine-tune the amount of causal consistency by trading off the size of meta-data overhead. We show three instantiations of the notion of credits, namely hop count, time-to-live, and metric distance.

(3) For the hop-count instantiation, we quantitatively evaluate the performance of Approx-Opt-Track for implementing causal consistency under partial replication. By controlling initial $credits$, we use simulations to analytically examine the trade-off between initial $credits$ and the size of meta-data. We also study the impacts of varying the number of processes, the replica factor, and the write rate.

With an initial $credits$ small enough ($credits = cr_0$), Approx-Opt-Track is seen to show significant gains over Opt-Track. In particular, for 40 processes, Approx-Opt-Track can lower the meta-data size (i) by around $5\% \sim 20\%$ without causal violations and (ii) from $40\% \sim 60\%$ for upto 0.5 percent causal violations ($credits < cr_0$).

The results (i) and (ii) imply that : a) Approx-Opt-Track is capable of further reducing the meta-data of Opt-Track. b) Approx-Opt-Track can significantly lower the meta-data by sacrificing causal consistency slightly. It appeals for some social network applications, which do not require completely strict causal consistency. For example, in Instagram, most of the comments/replies (update operations) correspond to the image (object creation) poster. Very few replies that do not appear in the correct order do not make the readers misunderstand the context of the comments. Approx-Opt-Track can be expected to improve total replication cost especially for multimedia social network applications.

This is the first study to develop the approximate causal consistency algorithm for partial replication and evaluate its performance. Note that our algorithm could be modeled to also provide eventual consistency by using the "last-writer wins" rule.

Preliminary version of contributions 1) $\sim$ 2) appeared in NDM '15 [16] and that of 3) appeared in ARMS-CC '16 [17].

1.2 Organization

Section 2 describes related work. Section 3 gives the system model of causally consistent memory and the underlying communication system. Section 4 presents the algorithm Approx-Opt-Track that implements causal consistency under partial replication. We have integrated the notion of credits into this algorithm, and explain how it leads to approximate causal consistency. Section 5 shows how the notion of credits can be instantiated. Section 6 presents the communication framework for simulating Approx-Opt-Track based on different initial $credits$ for the hop-count instantiation. Section 7 shows the experimental results. Section 8 illustrates the trade-off between initial $credits$ and the meta-data size. Section 9 gives a discussion and concludes.

2 Related Work

Causal message ordering for message passing systems was defined by Birman and Joseph [18] and they gave the ISIS CBCAST (causal broadcast) protocol to implement causal message ordering. An improved algorithm in the message passing model was given by Schiper et al. [19]. Later, Raynal et al. [20] proposed a simple abstraction and a matrix-based implementation of causal ordering for multicasts and point-to-point messaging. These implementations required $O(n^2)$ meta-data overhead in the average case. Kshemkalyani and Singhal [12], [13] identified the necessary and sufficient

conditions on the meta-data overheads for multicasts, and proposed an optimal algorithm (the KS algorithm) which implements these conditions and requires $O(n)$ meta-data overhead in the amortized case [21], [22].

Ahamad et al. [23] introduced the definition of causal consistency in DSM systems, and gave an algorithm for implementing causal consistency in such systems assuming full replication of the DSM. Baldoni et al. [15] later proposed an application-level protocol, in full replication DSM systems, that improves on the performance of the Ahamad et al. protocol. Their optimality criterion is enforced by the protocol $OptP$, which uses a reliable broadcast primitive. Recently, we proposed a protocol named Opt-Track [11] for partial replication in DSM systems, that adopts the necessary and sufficient conditions of the KS algorithm [12], [13] from message-passing systems. This protocol is the first work on causal consistency under partial replication, and practically has linear ($O(n)$) meta-data overheads in the amortized case [14].

Δ-causal ordering [24], [25] is designed for real-time message passing applications in unreliable networks. This communication abstraction guarantees that message delivery is based on causality order with a limited lifetime, denoted as Δ, after which the message data can no longer be valid. By exploiting transitive dependencies in message transmission, this real-time protocol significantly reduces the dependency meta-data piggybacked on real application file data in the case of peer-to-peer and broadcast communication. Real-time based message communication RTCM [26] can guarantee that multimedia data with real-time validity are efficiently delivered to the application level in causality order. Any message arriving at a destination before its time validity expires will be delivered to the application level. Due to the supports for data with real-time deadlines, the data is deleted after its valid deadline to minimize the dependency information. 'Δ' is different from $credits$ we propose in this study. When $credits$ become aged, the corresponding dependency meta-data is removed. However, in Δ-causal ordering, if the transmission time of a message is beyond Δ, the whole message will be ignored.

Torres-Rojas et al. [27] provided consistent access to objects based on the lifetime approach. The end of the lifetime assigned to an object indicates the time until which this object is valid. By keeping track of the lifetimes of the values stored in shared objects, they showed how to check the mutual consistency of a set of related objects stored at a local site in a synchronized system with efficient and scalable implementations of object sharing across widely distributed users. Crain et al. [28] illustrated a causally consistent protocol for geo-distributed partial replication with dependency vectors. However, it still considered imprecise representation of dependencies, which can result in false dependencies. Orbe [8] provided scalable causal consistency with explicit dependency tracking under full replication.

3 System Model

We use the same model used in [11], [15], [23], [29].

3.1 Causally Consistent Memory

We assume a concurrent system composed of n application processes $ap_1, \ldots, ap_n$ interconnected through a distributed shared memory $\mathcal{Q}$ which is formed of q variables $x_1, x_2, \ldots, x_q$. Each application process ap_i is allowed to either *read* from or *write* to any of the q shared variables. Each variable has an initial value $\perp$. $r_i(x_j)v$ denotes that ap_i invokes a *read* operation on variable x_j which returns value v. $w_i(x_j)u$ denotes that ap_i invokes a *write* operation on variable x_j which writes the value u.

A series of *read* and *write* operations performed in an application process ap_i generates a local history h_i. The set of local histories h_i from all n application processes is denoted by the global history H. When a local operation o_1 precedes another local operation o_2, this means that o_1 precedes o_2 under *program order*, denoted as $o_1 \prec_{po} o_2$. When two operations o_1 and o_2 from distinct processes ap_i and ap_j respectively, are such that $o_1 = w(x)v$ and $o_2 = r(x)v$, this means that *read* operation o_2 retrieves the value written by the *write* operation o_1; o_1 and o_2 are related under the *read-from order*, denoted as $o_1 \prec_{ro} o_2$. Two properties can be summarized as follows:

- 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. Specifically, 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 causal shared 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.

3.2 Underlying Distributed Communication System

To satisfy the causal consistency requirement under the partial replication model, the DSM application is implemented on top of a message passing communication system which is composed of n interconnected sites, each of which hosts an application process ap_i and holds only a subset of variables $x_h \in \mathcal{Q}$. The subset of variables kept on site s_i is denoted as X_i for application process ap_i. If the system replica factor is p and the variables are evenly replicated on all the sites, then the average size of X_i is $\frac{pq}{n}$.

To perform the read and write operations in the DSM application, some primitives are provided to communicate messages between different sites, as follows. The `RemoteFetch`(m) primitive is invoked when an application process performing a read operation to read variable x_2 needs to deliver the message m to fetch x_2's value from a randomly selected remote replica site (since x_2 is not locally replicated). Because this is a synchronous primitive, it will not complete until the variable's value is returned. If the

variable to be read is locally replicated, then the application process simply returns the local value. The $\texttt{Send}(m)$ primitive is invoked when an application process performs a write operation $w(x_1)v$ to deliver the message m to all sites that replicate the variable x_1 with an updated value v. The read and write operations performed by the application processes also generate *events* in the underlying message passing system. The following is a list of events:

- *Send event*. The invocation of $\texttt{Send}(m)$ 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 $fetch_i(f)$.
- *Receive event*. The receipt of a message m at site s_i generates event $receive_i(m)$. 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 $receive_i(m)$ 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.

Each time an update message m carrying the information $w_j(x_h)v$ is received at site s_i, a new thread is spawned to check when to apply the update request to the local replica, satisfying causal consistency. The condition whether the update value is ready to be applied is called activation predicate in [15]. This predicate $A(m_{w_j(x_h)v}, e)$, which is $false$ by default initially, becomes $true$ only if the update request $m_{w_j(x_h)v}$ can be applied after the occurrence of local event e. The local thread handling the update request will be blocked until the activation predicate becomes $true$, at which time the thread applies value v to variable x_h's local replica. This apply action will result in the $apply_i(w_j(x_h)v)$ event locally.

3.3 Activation Predicate

Consider the causality order relation, $\rightarrow_{co}$, on *send events* created in the underlying message passing system [15], where the relation $\rightarrow_{co}$ is a partial order on the $send$ events generated by write operations respecting the partial order induced by $\prec_{co}$ on operations. We further modify its definition by adding condition (3) to accommodate the partial replication system. Suppose that $w(x)a$ and $w(y)b$ are two write operations in O_H. Then, for their corresponding send events in the underlying message passing system, $send_i(m_{w(x)a}) \rightarrow_{co} send_j(m_{w(y)b})$ if and only if 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) $i \neq j$ and $\exists l$ such that $apply_l(w(x)a)$ locally precedes $remote\_return_l(r_j(x)a)$, which precedes (as per Lamport's $\rightarrow$ relation [30]) $return_j(x, a)$, which locally precedes $send_j$ $(m_{w(y)b})$
(4) $\exists send_k(m_{w(z)c})$, such that $send_i(m_{w(x)a}) \rightarrow_{co} send_k(m_{w(z)c}) \rightarrow_{co} send_j(m_{w(y)b})$

The relation $\rightarrow_{co}$ is a subset of Lamport's "happened before" relation [30], denoted by $\rightarrow$. If there exists a $\rightarrow_{co}$ relation on two $send$ events, then they are also related by $\rightarrow$. However, the reverse way is not necessarily true. Thus, even if $send_i(m_{w(x)a}) \rightarrow send_j(m_{w(y)b})$ without a return event in between and $i \neq j$, these two send events are concurrent under the $\rightarrow_{co}$ relation. The $\rightarrow_{co}$ relation better represents the causality order in the DSM abstraction to eliminate the "false causality" in the underlying message passing system, where message receive events may causally relate two send events while their corresponding write operations in the shared memory application are concurrent under the $\prec_{co}$ relation. (False causality was identified by Lamport [30].) In [15], the authors have shown that $send_i(m_{w(x)a}) \rightarrow_{co} send_j(m_{w(y)b}) \Leftrightarrow w(x)a \prec_{co} w(y)b$.

The optimal activation predicate is 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 that happened at the site s_i up until e (excluding e).

The causal memory's requirement that a write operation shall not be seen by an application process before any causally preceding write operations is cleanly captured by this activation predicate. This activation predicate $A_{OPT}(m_w, e)$ is optimal because the moment it becomes true is the earliest instant that the update m_w can be applied.

4 Algorithm

4.1 Opt-Track Protocol

As mentioned before, Opt-Track protocol [11] adapts the KS algorithm to a partially replicated causal DSM system. Each site s_i holds a collection of the most recent causal updates, that happened before under the $\rightarrow_{co}$ relation. Each record in the collection contains a set of destinations, each of which represents one replica node of the corresponding update. When a write operation is initiated, the delivered multicast update messages will piggyback the recently stored collection records. On receiving an update message, the receiver site can use the optimal activation predicate A_{OPT} to determine when to apply the update value under causal consistency. Once the apply event is applied, the piggybacked collection of records is associated with the corresponding variable. If a read operation accesses the update variable later, the corresponding associated collection of records will be merged into the local collection of records. Opt-Track will prune redundant destination records by the following conditions to achieve the optimality of dependency metadata size. Here, *implicitly remembering* means that information can be inferred from other later log entries, without storing that information.

- *Propagation and Pruning 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

TABLE 1
Complexity Measures of Opt-Track [11]

Metric	Opt-Track
Message count	$((p-1)+\frac{n-p}{n})w + 2r\frac{(n-p)}{n}$
Message space overhead	$O(n^2pw + nr(n-p))$ amortized $O(npw + r(n-p))$
Time Complexity	write $O(n^2p)$ read $O(n^2)$
Space Complexity	$O(max(n^2, npq))$ amortized $O(max(n, pq))$

explicitly remembered in the causal future of the event $apply_2(w)$. We *implicitly* remember it in the causal future (under the $\rightarrow_{co}$ relation) of event $return_2(x, v)$ (and event $remote\_return_2(r_*(x)v)$), to clean the logs at other sites. This is because m is already delivered to s_2.

- *Propagation and Pruning 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 allows inferring that the update m has already been applied at s_2. We *implicitly* remember in the causal future (under the $\rightarrow_{co}$ relation) of events $return_k(y, v')$ (and events $remote\_return_k(r_*(y)v')$) that m is transitively guaranteed to be delivered to s_2, to clean the logs at other sites.

For message M from source i and timestamped a, a destination set is denoted as $M_{i,a}.Dests$. Log entries are denoted by l and message overhead entries as o. The logs at the sites are cleaned by implicitly tracking messages by Pruning Condition 1 or 2 as follows.

- *Implicit Tracking 1:* $\exists d \in M_{i,a}.Dests$ such that $d \in l_{i,a}.Dests \bigwedge d \notin o_{i,a}.Dests$. Then d can be deleted from $l_{i,a}.Dests$ because it can be inferred that $M_{i,a}$ is delivered to d or is transitively guaranteed to be delivered in causal order. When $l_{i,a}.Dests = \emptyset$, it can be inferred that $M_{i,a}$ is delivered or is transitively guaranteed to be delivered in causal order, to all its destinations.
- *Implicit Tracking 2:* If $a_1 < a_2$ and $l_{i,a_2} \in LOG_j$, then l_{i,a_1} implicitly or explicitly exists in LOG_j. Entries of the form $l_{i,a_1}.Dests = \emptyset$ can be inferred by their absence and should not be stored.

Using *implicit* knowledge about messages delivered and transitively guaranteed to be delivered in causal order, the Opt-Track protocol [11] automatically prunes the meta-data. This pruning is in addition to the pruning done by using *explicitly* maintained information. In the amortized case, the meta-data is manageable and linear in n, rather than quadratic (See Table 1 in [11] and [14]). Four metrics were used in the complexity analysis:

- message count: the total number of messages generated in an execution.

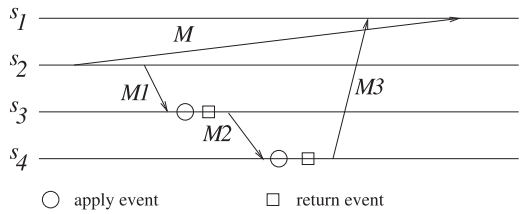

Fig. 1. Illustration of causal consistency violation if credits are exhausted.

- message space overhead: the total size of the meta-data generated in an execution, which is formalized as $\sum_i$*(# type i messages * size of type i messages)*.
- time complexity: the time complexity at each site s_i for executing a write or a read operation.
- space complexity: the space complexity at each site s_i for storing local logs.

The following parameters are used for the DSM system:

- n: the number of sites.
- q: the number of variables.
- p: the replica factor, i.e., the number of replica sites for each variable. (r_f: the replica factor rate is defined as p/n)
- w: the number of write operations performed.
- r: the number of read operations performed.

The complexity of Opt-Track is summarized in Table 1. We further illustrate the overhead of partial replication versus full replication in terms of Opt-Track and Opt-Track-CRP. Suppose that f is the size of an image/data being written and b is the number of bytes in an integer.

Full Replication. The net payload data size for a write operation is $(n-1) \times f$, and $n \times b \times (n-1)$ for the meta-data overheads in the worst case of Opt-Track-CRP protocol [11]. In practice, the space size of the local log depends on the number of entries in the local log, expressed as d, which is only a small finite value. The read cost is zero. In short, the total message space cost arising from one write operation in full replication is $(n-1)f + db(n-1)$.

Partial Replication. In Opt-Track, the net payload data size is $((p-1) + \frac{(n-p)}{n})f$ for a write operation. With $\frac{r}{w}$ reads per write and $\frac{n-p}{n}$ of them to fetch the data from a remote replica, the read cost is $(\frac{r}{w})\frac{(n-p)}{n} f$. The corresponding meta-data overheads are $((p-1) + \frac{(n-p)}{n})n^2b + (\frac{r}{w})\frac{(n-p)}{n}(n^2+1)b$ in the worst case. It was shown in [14] that the amortized log size and message overhead size is approximately $O(n)$, not $O(n^2)$. The amortized total message space cost is $((p-1) + \frac{(n-p)}{n})nb + (\frac{r}{w})\frac{(n-p)}{n}(n+1)b$.

Example. Assume that one word holds 4 B, f = 50 B (the average text tweet size), and each message corresponds to one write operation. In Figs. 1 or 2, p is 2; n is 4; d is 2. The

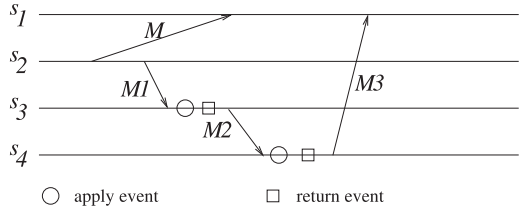

Fig. 2. Illustration of meta-data reduction when credits are exhausted.

total transmitted data in Opt-Track is around 300 B. However, considering full replication, the total transmitted data in Opt-Track-CRP is around 700 B. In the next section and Section 5, we show the details of how to reduce the meta-data overheads of Opt-Track further.

4.2 Basic Idea of Approx-Opt-Track

We can further reduce the size of meta-data by deleting older dependencies rather than carry them around and store them in logs. With very high probability, the older the dependencies are, the more they are likely to be immediately satisfied as the corresponding messages are more likely to be delivered. We introduce the notion of *credits* associated with each meta-data unit of information. When a dependency is created, it is allocated a certain number of initial credits. For every read and write operation, we decrement the available credits by some used-up credits, and when the available number of credits reaches zero, the dependency becomes "old enough" and can be deleted. By setting the initial credits to ∞, we get the original Opt-Track algorithm. By setting them to a smaller finite value, we can prune meta-data information about older dependencies by risking that those dependencies might not be satisfied, rather than wait for the pruning mechanisms of Opt-Track to prune them. *Credits* is a parameter that lets us approximate causal consistency to the accuracy desired.

Consider the timing diagram in Fig. 1. The messages shown indicate those sent due to write operations to update the remote replica. The causality chain induced by write operations corresponding to $M1$, $M2$, and $M3$, and the intervening *apply* and *return* events, ends in $M3$ being sent to site s_1. Normally in Opt-Track, the meta-data on $M3$ contains the dependency that "M is sent to s_1", and will prevent $M3$ from being delivered to s_1 before M is delivered. However, if the credits get expired along this causality chain, then $M3$ will not carry the meta-data dependency that "M is sent to s_1" and hence $M3$ will be delivered by violating causal consistency at s_1. If credits are decremented slowly enough, then with very high probability, $M3$ will carry the meta-data information about M and causal consistency is not violated.

In another scenario, consider the timing diagram in Fig. 2. It is the same as in Fig. 1, with the exception that message M is delivered to s_1 within a reasonable (i.e., an expected) amount of time. Assume that the credits about the dependency "M is sent to s_1" get exhausted when $M2$ is delivered to s_4 along the causality chain $\langle M1, M2\rangle$. The dependency is thus deleted at s_4 and is not carried in the meta-data sent along with message $M3$. This results in reduced meta-data on $M3$. This does not cause any violation of causal consistency when the reduced meta-data is delivered to s_1, because M has been delivered to s_1.

4.3 Approx-Opt-Track

Algorithm 1 shows the steps performed by Approx-Opt-Track. The following data structures are needed for each site.

(1) $clock_i$: local counter for write operations performed at site s_i by application process ap_i.

(2) $Apply_i[1 \dots n]$: an integer array (initially set to 0s). $Apply_i[j] = y$ means that y updates written by application process ap_j have been applied at site s_i.

(3) $LOG_i = \{\langle j, clock_j, Dests, cr\rangle\}$: the local log (initially set to empty). Each entry $\langle \bullet \rangle$ specifies a write operation, for which $Dests$ is the destination set, in the causal past. Only necessary destination information is stored. cr is the remaining amount of credits allowed before the entry ages out.

(4) $LastWriteOn_i\langle \text{variable id}, LOG\rangle$: a hash map of LOGs. $LastWriteOn_i\langle h\rangle$ contains the piggybacked LOG for the most recent update applied at site s_i to locally replicated variable x_h.

Algorithm 1. Approx-Opt-Track Algorithm, which is a Modification of Algorithm Opt-Track [11] (Code at site s_i)

WRITE(x_h, v):
1: $clock_i + +$;
2: $cr :=$ initial credits;
3: **for** *all* $l \in LOG_i$ **do**
4: $\quad l.cr := l.cr -$ used credits;
5: $\quad$ **if** $l.cr \leq 0 \wedge l.Dests \neq \emptyset$ **then** delete l;
6: **for** *all sites* $s_j (j \neq i)$ *that replicate* x_h **do**
7: $\quad L_w := LOG_i$;
8: $\quad$ **for** *all* $o \in L_w$ **do**
9: $\quad\quad$ **if** $s_j \notin o.Dests$ **then** $o.Dests := o.Dests \setminus x_h.replicas$;
10: $\quad\quad$ **else** $o.Dests := o.Dests \setminus x_h.replicas \cup \{s_j\}$;
11: $\quad$ **for** *all* $o_{z,clock_z} \in L_w$ **do**
12: $\quad\quad$ **if** $o_{z,clock_z}.Dests = \emptyset \wedge (\exists o'_{z,clock'_z} \in L_w | clock_z < clock'_z)$ **then** remove $o_{z,clock_z}$ from L_w;
13: $\quad$ Send $m(x_h, v, i, clock_i, x_h.replicas, cr, L_w)$ to site s_j;
14: **for** *all* $l \in LOG_i$ **do**
15: $\quad l.Dests := l.Dests \setminus x_h.replicas$;
16: PURGE;
17: $LOG_i := LOG_i \cup \{\langle i, clock_i, x_h.replicas \setminus \{s_i\}, cr\rangle\}$;
18: **if** x_h *is locally replicated* **then**
19: $\quad x_h := v$;
20: $\quad Apply_i[i] + +$;
21: $\quad LastWriteOn_i\langle h\rangle := LOG_i$;
READ(x_h):
22: **if** x_h *is not locally replicated* **then**
23: $\quad$ RemoteFetch$[f(x_h)]$ from randomly selected site s_j that replicates x_h to get x_h and $LastWriteOn_j\langle h\rangle$;
24: $\quad$ MERGE$(LOG_i, LastWriteOn_j\langle h\rangle)$;
25: **else** MERGE$(LOG_i, LastWriteOn_i\langle h\rangle)$;
26: PURGE;
27: return x_h;
On receiving $m(x_h, v, j, clock_j, x_h.replicas, c, L_w)$ from site s_j:
28: **for** *all* $o_{z,clock_z} \in L_w$ **do**
29: $\quad$ **if** $s_i \in o_{z,clock_z}.Dests$ **then** wait until $clock_z \leq Apply_i[z]$;
30: **for** *all* $o \in L_w$ **do**
31: $\quad o.cr := o.cr -$ used credits;
32: $\quad$ **if** $o.cr \leq 0 \wedge o.Dests \neq \emptyset$ **then** delete o;
33: $x_h := v$;
34: $Apply_i[j] := clock_j$;
35: $L_w := L_w \cup \{\langle j, clock_j, x_h.replicas, c -$ used credits$\rangle\}$;
36: **for** *all* $o_{z,clock_z} \in L_w$ **do**
37: $\quad o_{z,clock_z}.Dests := o_{z,clock_z}.Dests \setminus \{s_i\}$;
38: $LastWriteOn_i\langle h\rangle := L_w$;
On receiving $f(x_h)$ from site s_j:
39: return x_h and $LastWriteOn_i\langle h\rangle$ to s_j;

The data structures are the same as in algorithm Opt-Track, with the addition of the credits parameter cr in each entry in LOG_i. Algorithm 1 implements the optimality mechanisms described in algorithm Opt-Track [11].

For a write operation, it will send different log meta-data L_w to different replica sites. Lines (6)-(13) formulate the meta-data for each replica site and minimize its space overhead. Lines (8)-(10) and lines (14)-(15) can prune the destination sets by Propagation and Pruning Condition 2. Lines (36)-(37) prune the redundant information by Propagation and Pruning Condition 1. Lines (28)-(29) are used to implement the optimal activation predicate A_{OPT}.

Algorithm 2 shows two functions used in Algorithm Approx-Opt-Track (Algorithm 1). Function PURGE removes the records with $\emptyset$ destination sets, per sender process. When reading variable x_h, function MERGE combines the piggybacked log of the corresponding write to x_h and the local log LOG_i. In this function, new dependencies are added to LOG_i and old dependencies in LOG_i are pruned, based on the information in the piggybacked data L_w. The merging procedure realizes the optimality techniques of Implicit Tracking1.

Algorithm 2. Procedures Used in Algorithm 1, Approx-Opt-Track Algorithm (Code at Site s_i)

PURGE:
1: **for** *all* $l_{z,t_z} \in LOG_i$ **do**
2: **if** $l_{z,t_z}.Dests = \emptyset \wedge (\exists l'_{z,t'_z} \in LOG_i | t_z < t'_z)$ **then**
3: remove l_{z,t_z} from LOG_i;
MERGE (LOG_i, L_w):
4: **for** *all* $l \in LOG_i$ **do**
5: $l.cr :=$ $l.cr -$ used credits;
6: **for** *all* $o \in L_w$ **do**
7: $o.cr :=$ $o.cr -$ used credits;
8: **for** *all* $o_{z,t} \in L_w$ and $l_{s,t'} \in LOG_i$ *such that* $s = z$ **do**
9: **if** $t < t' \wedge l_{s,t} \notin LOG_i$ **then** mark $o_{z,t}$ for deletion;
10: **if** $t' < t \wedge o_{z,t'} \notin L_w$ **then** mark $l_{s,t'}$ for deletion;
11: delete marked entries;
12: **if** $t = t'$ **then**
13: $l_{s,t'}.Dests := l_{s,t'}.Dests \cap o_{z,t}.Dests$;
14: $l_{s,t'}.cr := \min(l_{s,t'}.cr, o_{z,t}.cr)$;
15: delete $o_{z,t}$ from L_w;
16: $LOG_i := LOG_i \cup L_w$;
17: **for** *all* $l \in LOG_i$ **do**
18: **if** $l.cr \leq 0 \wedge l.Dests \neq \emptyset$ **then** delete l;

Notice that in the PURGE function, and in lines (11)-(12) of the WRITE procedure, 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 is required for implicit tracking of messages delivered and guaranteed to be delivered in causal order using Implicit Tracking 2, as explained in [11], [12], [13]. Such entries should not be deleted even if their credits allocation becomes zero. Thus in line 5, line 32, of the main algorithm, and in line 18 of MERGE, we delete an entry with exhausted credits only if its destination list is non-empty.

5 Credit Instantiations

Using the hop count instantiation, credits of a meta-data entry denote the hop count available before the entry ages out and is deleted. A message is said to traverse one hop when it traverses along a logical channel between any pair of processes (sites). We make some notes about this instantiation of Algorithm Approx-Opt-Track.

(1) Line 2: initial assignment of the hop count for a new dependency created by a write operation.
(2) Lines (3)-(5): These lines are no-ops because there is no message transfer.
(3) Lines (30)-(32): In line 31, the hop count is decremented by one for each entry in the piggybacked meta-data received.
(4) Line 35: The hop count is decremented by one for the new dependency just formed by the received message.
(5) Lines (4)-(7) in MERGE: The entries in LOG_i do not experience any decrease in hop count, while the entries in L_w have the hop count decremented if the data was remotely fetched by the read operation that triggered the MERGE.
(6) Line 14 in MERGE: The hop count is set to the minimum of the hop counts of the entries being merged.
(7) Lines (17)-(18) in MERGE: LOG_i entries whose hop count is zero are deleted.

Example. We illustrate Approx-Opt-Track with hop count credits by using Figs. 1 and 2, and quantitatively show how much improvement one can gain in running Approx-Opt-Track against Opt-Track. Initially, a hop count credit x is assigned to the meta-data "M is sent to s_1" (denoted m_d). After receiving and applying M_1 at s_2, the credits will be decremented by one. When M_2 is delivered to and applied at s_4, if the credit value $x - 2$ is not greater than zero, m_d will be removed from the corresponding record. Thus, if x was initialized to 2, when M_3 is delivered to s_1 without piggybacking m_d, applying M_3 at s_1 violates causal consistency in Fig. 1, but not in Fig. 2. By controlling x, Approx-Opt-Track can carry less amount of dependency meta-data. We expect that if the initial allocation of x is made as a high single-digit, by the time x reaches zero and the meta-data entry is deleted, the message about which the meta-data is deleted would already have reached its destination (with very high probability). Consider Fig. 2. In Opt-Track, there are 6 dependency records delivered ($1(M_1) + 2(M_2) + 3(M_3)$) and 5 dependency records ($2(s_3) + 3(s_4)$) saved in local storages by the two apply events. In Approx-Opt-Track, when x is set to 2, there are 5 dependency records delivered ($1(M_1) + 2(M_2) + 2(M_3)$) and 4 dependency records ($2(s_3) + 2(s_4)$) saved in local storages by the two apply events. Approx-Opt-Track can not only reduce transmitted data and storage overheads as compared to Opt-Track but can also maintain causal consistency. Our simulation results focus on the relationship between adequate x (no causal violation) and n (the number of processes) and the trade-off between adequate x and how much meta-data can be reduced further.

'Physical time lapse' and 'metric distance traversed' can be used as instantiations of credits. Note that we assume that the network is symmetric and homogeneous. In this case, the TTL and physical distance are not good metrics or they need to be modified to reflect the real trade-off between causal violation and $credits$. For simplicity, we consider the metric of the hop count $credits$.

6 Simulation System Model

We describe the experimental methodology used to evaluate the performance of the proposed Approx-Opt-Track

algorithm in an asynchronous distributed system, with respect to the instantiation of $credits$, by hop count. The system is composed of a finite number of interconnected sites. Each site has only one asynchronous process with a local memory, for simplicity. All the processes can communicate by asynchronous message passing through TCP channels of the underlying network, where messages are delivered in FIFO order with no omissions or duplications.

6.1 Process Model

Two major subsystems in a process are the application subsystem and the message receipt subsystem. The purpose of the application subsystem (AS) is to facilitate scheduling operation events (write/read). AS not only maintains a floating point local clock to generate event patterns based on a temporal schedule, but also is responsible for handling $Write$ and $Read$ functions. The message receipt subsystem (MRS) takes charge of responding to remote request service. MRS mainly consists of $ApplyingMulticast$ and $RespondingFetch$ services.

The simulation system core is based on Approx-Opt-Track. For a write operation $w(x_h)v$, AS invokes a $send$ event to deliver the message $m(w(x_h)v)$ with the corresponding meta-data—local log L_w and hop count $credits$ to other replicas. For a read operation $r(x_h)$, AS returns the local value of variable x_h or invokes a $fetch$ event to deliver the message $fetch(x_h)$ to a predesignated site to retrieve the remote variable x_h's value as well as the corresponding meta-data $LastWriteOn\langle h\rangle$ and hop count $credits$. MRS can recognize and allow the receipt of two distinct types of incoming messages. First, if the incoming message m contains a write operation $w(x_h)v$, MRS will determine when to apply the new value v to the variable x_h under causal consistency and then update the meta-data log $LastWriteOn\langle h\rangle$ and hop count $credits$. Second, on receiving a $remote\ fetch\ message\ m(fetch(x_h))$, it invokes a $remote\ return$ event to reply with the local value of the variable x_h, the corresponding meta-data log $LastWriteOn\langle h\rangle$, and hop count $credits$ to the requesting site.

6.2 Simulation Parameters

The system parameters whose effects we examine on the performance of Approx-Opt-Track are as follows:

- Number of Processes (n): The performance of every DSM implementation highly depends on the node count, or core count, and the underlying hardware running the simulation (i.e., memory allocation of the benchmark device). It is hence necessary to simulate a DSM system over a wide range of n. On an Intel Core 2 Duo workstation with a JDK8 virtual machine, we can simulate up to 40 processes.
- Number of Variables (q): q is usually unbound in a real system. Subject to the memory limitation, q in the benchmark experiment is one hundred.
- Replica Factor Rate (r_f): The ratio of the number of replicas to n.
- Write Rate (w_{rate}): It is defined as the ratio of the number of write operations to the total number of operations. Binding by a variety of write rates, we can study performance for read-intensive and write-intensive application workloads.
- Hop Count Credits (cr): Credits of a meta-data entry denote the hop count available before the entry ages out and is removed.
- Message Count (m_c): The total number of messages generated by all the processes.
- Message Meta-Data Size (m_s): The total size of all the meta-data transmitted over all the processes.

6.3 Process Execution

For simplicity, we consider a homogeneous and fully connected network, in which all the processes are symmetric. The operation events are triggered based on a event schedule randomly generated in advance. Subject to the underlying hardware, the time interval T_e between two events at a process is given from 5 to 2,005 ms. The propagation time T_t is from 100 to 3,000 ms. Consider the real-world network communication situation. T_t is initially given as $5\ \mathrm{ms} \sim 300\ \mathrm{ms}$ and T_e is set to be $10\ \mathrm{ms} \sim 200\ \mathrm{ms}$. We simulated multi-processes on a standalone workstation. They use the TCP protocol to transmit messages, where each transmission establishes a TCP "short" connection, getting closed after delivering the message. This TCP port cannot be used immediately. It is released after some delay. If T_t and T_e are so short as to cause sockets to be leaked, with time, this exhausts all available TCP ports. A compromise solution is to increase the number of ephemeral network ports and make sure the rate at which the connections are created does not throttle the kernel memory. There are no formally documented approaches available to increase non-paged kernel memory in a standalone machine. With not enough hop count credits, causal violation would depend on the combination of T_e and T_t. In order to avoid connection exceptions, we need to lower the connection rate in our simulation. To seek the next-best thing, we formulate isomorphic communication patterns with the above larger T_e and T_t. The range of T_e is finally set from 5 through 2,005 ms and that of T_t from 100 through 3,000 ms. The simulation results of these two different time ranges in smaller numbers of processes are not obviously distinct.

The processes in the distributed system execute concurrently. Simulating each process as an independent process at a site invoked inter-process communication. When a process gets initialized, it first invokes the MRS. Then, the system executes $Scheduled\text{-}ExecutorService$ in JDK to drive the AS which extends $TimerTask$ class—a JDK scheduling service to dispose of the scheduling operation events. In the simulation, the system relies on TCP channels to deliver messages. An AS stops generating operation events once it runs out of all the scheduling events and flags its status as finished. The simulation is done when all the ASs have their status set to 'finished'.

6.4 Causal Consistency Verification

In this paper, we undertake a comprehensive study of the trade-off relations among the initial allocation of $credits$, the dependency meta-data size, and the causal consistency correctness. It is a critical issue to detect how many receiving operations violate causal consistency in a simulation. In Algorithm 1, theoretically, when the hop count reaches zero, the dependency meta-data entry is deleted. It means that not all the meta-data entries (explicit information of destinations) are kept before the destination information becomes redundant. Thus, while A_{OPT} in lines (28)-(29)

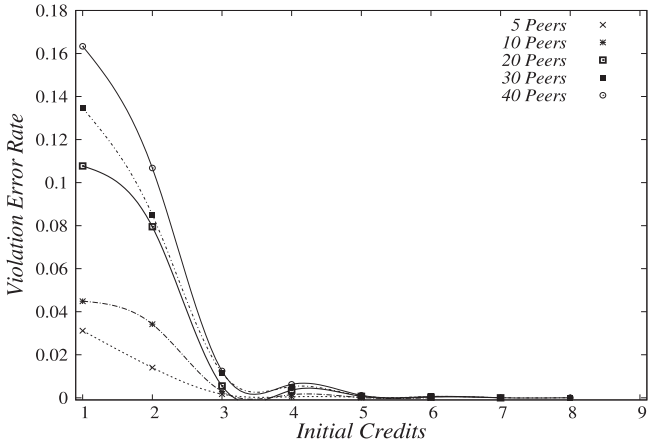

Fig. 3. The violation error rate for w_{rate} = 0.2.

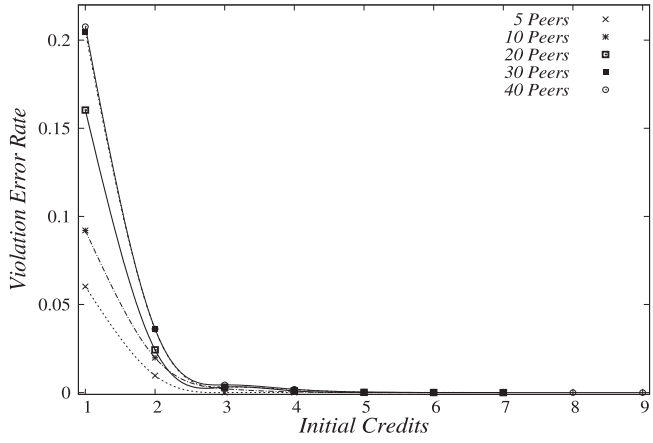

Fig. 4. The violation error rate for w_{rate} = 0.5.

(Algorithm 1) becomes true, the instant that an update m_w is applied may violate causal consistency.

Practically, the meta-data entry whose cr is equal to zero would be marked rather than deleted in line 32 (Algorithm 1) and line 18 (MERGE, Algorithm 2). If A_{OPT} ($clock_z \leq Apply_i[z]$) becomes true and at least one $o.cr \leq 0$ (the meta-entry is marked), this receiving operation applied would violate causal consistency. We realize this verification mechanism at the application level by counting the number of messages applied by the remote replicated sites with a violation of causal consistency, denoted as n_e.

7 Simulation Results

We present results of simulations performed to study the trade-off among initial credits cr, message meta-data size m_s, and causal consistency accuracy rate in the empirical evaluation. The performance metrics used are as follows:

- The causal consistency violation error rate, R_e.
- The average size of the message meta-data transmitted for different initial credits and write rates, m_{ave}.
- The message meta-data size saving rate, R_s.

In order to clarify the relative contribution of these metrics, multivariate analyses were conducted. Choosing a different set of parameters (r_f, w_{rate}, n, and cr) will result in a different evaluation. Our evaluations realize four evaluation loops, each of which corresponds to one parameter. The loop structure is as follows: First, we select a r_f. It was set to be 0.3, 0.2, and 0.5. The first loop evaluates the impact of r_f. Second, the w_{rate} was set to be 0.2 (lower write rate), 0.5 (medium write rate), and 0.8 (higher write rate), respectively. The second loop evaluates the impact of different w_{rate}. Third, n was varied from 5 up to 40. This tests the scalability of Approx-Opt-Track. Finally, the innermost loop varied the initial hop count credit cr. It was specified from one to a critical value cr_0, with which there is no message transmission violating causal consistency in the corresponding simulation. Not only does this illustrate how Approx-Opt-Track further reduces the meta-data overheads but it also verifies the trade-off between meta-data saving rates and causal violation degrees.

Before running an evaluation (corresponding to a set of parameters), we randomly generate a task schedule set T_s. Because this is a simulation-based study, randomness is introduced in the readings. For each experimental evaluation, three runs were performed over the same T_s. The variations for all the simulation results are less than 2 percent. The mean of numerical results performed from three runs is represented for each combination of the four parameters. Each simulation execution runs $600n$ operation events totally. Experimental data was stored after the first 15 percent operation events to eliminate the side effect of startup.

7.1 Violation Error Rate (R_e)

In Section 6.4, we defined n_e as the number of messages applied by the remote replicated sites with a violation of causal consistency. We define R_e as the ratio of n_e to the total number of transmitted messages m_c. The results for R_e versus different initial hop count credits are shown in Figs. 3, 4, and 5. Each of them corresponds to a different w_{rate}.

With increasing cr (less than 4), R_e rapidly decreases. For the same initial cr and w_{rate}, the larger the n, the higher the R_e. The larger the w_{rate}, the lower the R_e. Table 2 highlights the results for two types of critical initial credits (cr_c) when R_e is around 0.5 percent (exactly, $0.4\% \sim 0.6\%$) and R_e = 0 (no causal consistency violation). When n is larger, the critical initial cr is basically also larger.

7.2 Average Message Meta-Data Size (m_{ave})

Figs. 6, 7, and 8 visualize the experimental data of m_{ave}. With increasing initial credit cr, m_{ave} linearly increases. The findings also indicate that m_{ave} becomes smaller with a

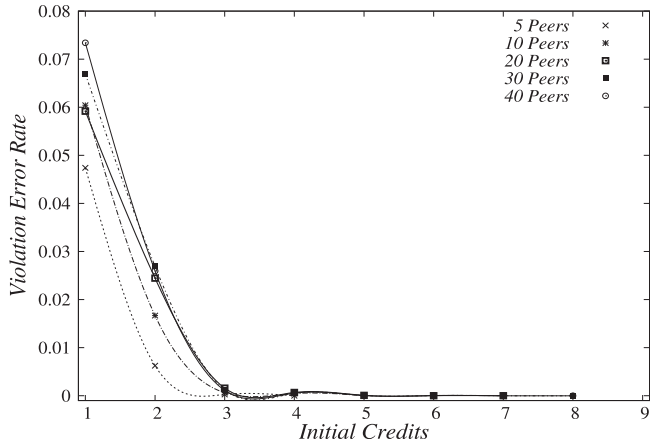

Fig. 5. The violation error rate for w_{rate} = 0.8.

TABLE 2
Critical Initial Credits for the Replica Factor Rate = 0.3

	w_{rate}	the number of processes				
		5	10	20	30	40
$R_e \sim 0.5\%$	0.2	3	3	3	4	4
	0.5	3	3	3	3	3
	0.8	3	3	4	4	4
$R_e = 0$	0.2	5	6	7	8	8
	0.5	3	5	7	7	9
	0.8	4	5	7	8	8

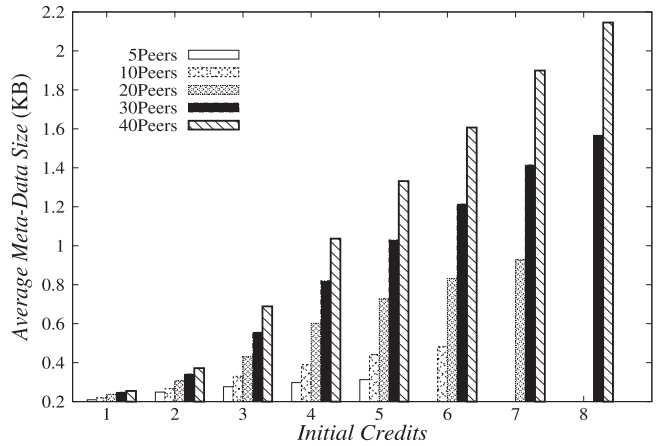

Fig. 6. The average meta-data size (m_{ave}) for w_{rate} = 0.2.

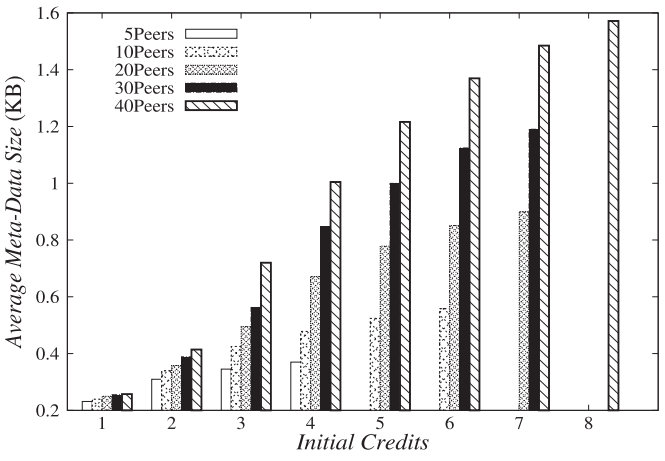

Fig. 7. The average meta-data size (m_{ave}) for w_{rate} = 0.5.

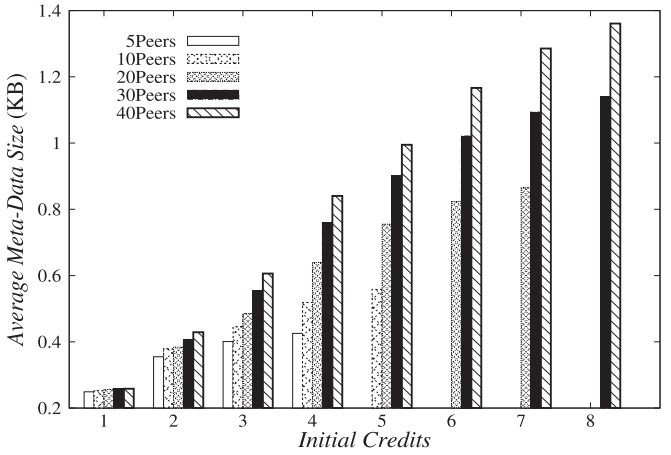

Fig. 8. The Average Meta-Data Size (m_{ave}) for w_{rate} = 0.8.

TABLE 3
Critical Average Message Meta-Data Size m_{ave} (KB)

R_e	w_{rate}	the number of processes				
		5	10	20	30	40
$\sim 0.5\%$	0.2	0.277	0.330	0.430	0.820	1.037
	0.5	0.345	0.425	0.495	0.562	0.720
	0.8	0.401	0.445	0.640	0.759	0.840
0	0.2	0.312	0.481	0.927	1.566	2.146
	0.5	0.345	0.524	0.899	1.190	1.572
	0.8	0.426	0.558	0.864	1.140	1.361

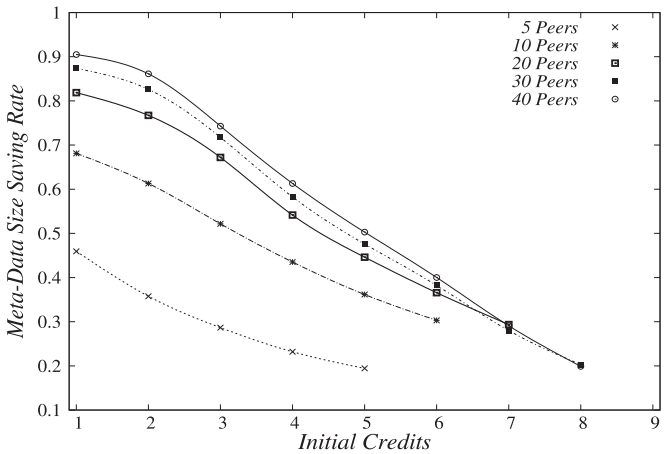

Fig. 9. The Meta-Data Size Saving Rate (R_s) for w_{rate} = 0.2.

higher w_{rate} in more peers. Table 3 lists the critical m_{ave} corresponding to the numerical data in Table 2.

7.3 Message Meta-Data Size Saving Rate (R_s)

Note that Approx-Opt-Track with $cr = \infty$ is equivalent to Opt-Track in [11]. We define R_s as follows:

$$R_s = 1 - \frac{m_s(cr \neq \infty)}{m_s(\text{Opt} - \text{Track})}. \quad (1)$$

Figs. 9, 10, and 11 reflect the results for R_s versus different initial credits in different (low, medium, and high) write rates, respectively. Note that although cr is unbounded in Approx-Opt-Track, it does not make sense for simulation with $cr > cr_c$, in which cases, there is no message delivery violating causal consistency.

With increasing n, R_s increases. Corresponding to the same n and initial credit cr, the higher the w_{rate}, the lower the R_s seems to be. Table 4 lists the critical R_s corresponding to the numerical data in Table 2. It can be seen that R_s is significantly negatively related to w_{rate}.

8 Simulation Evaluation

We expect that if the initial allocation of hop count cr is a small finite value but enough, it not only reduces message meta-data size, but also maintains the desired causal consistency accuracy. In other words, it is expected with very high probability that when cr reaches zero so as to delete

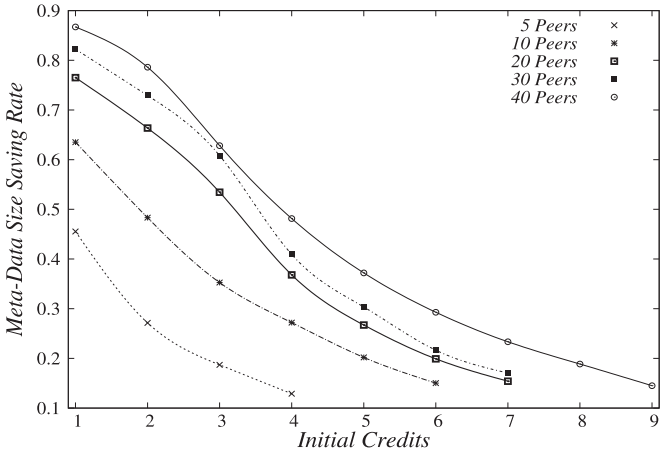

Fig. 10. The Meta-Data Size Saving Rate (R_s) for w_{rate} = 0.5.

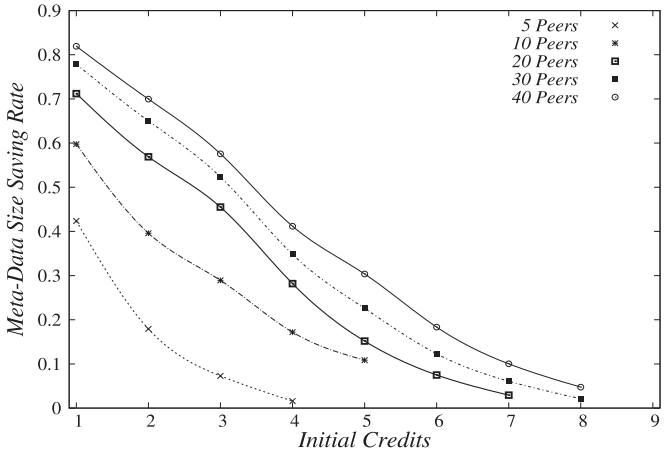

Fig. 11. The Meta-Data Size Saving Rate (R_s) for w_{rate} = 0.8.

the corresponding entry, the message associated with it has reached its destination.

8.1 Impact of Initial cr on R_e

With increasing the initial cr, R_e rapidly decreases especially when $cr < 4$. The smaller the initial cr, the earlier the meta-data entry is deleted. Thus, when an entry is removed earlier, it is more likely that the message associated with the deleted entry might not reach its destination. It causes that the corresponding dependency may not be satisfied, resulting in higher R_e. Table 2 shows the major and minor critical initial credits — cr_0 (R_e = 0) and $cr_{\sim 0.5\%}$ (R_e = $0.4\% \sim 0.6\%$) — for different numbers of processes.

For the minor critical initial $credits$, $cr_{\sim 0.5\%}$ seems not to significantly increase as the number of processes n. It implies that by setting initial $credits$ to a small finite value but enough value, most of the dependencies associated with the meta-data will become aged and can be removed without risking causal violations after being transmitted across a few hops, even in a large number of processes. On the other hand, in R_e = 0 (no causal violations), the resulting correlation coefficients of cr_0 and n for different w_{rate} are around $0.94 \sim 0.95$. It means that the more the number of processes n, the larger the initial cr is to avoid causal violations.

8.2 Impact of w_{rate} on R_e

This section evaluates how different write rates influence the violation error rates R_e across a variety of process numbers and initial credits. Figs. 3, 4, and 5 show the results of R_e among different w_{rate} in smaller initial credits over different process numbers. We observe that w_{rate} does not have an apparent impact on R_e when $cr > 4$. However, we can see the variation of R_e with w_{rate} when initial $cr < 4$. For $cr > 1$, the higher the w_{rate}, the lower the R_e. Causal consistency follows *read-from order* $\prec_{ro}$. Two operations o_1 and o_2 have the relationship $o_1 \prec_{ro} o_2$ if there exists $o_1 = w(x)v$ (write a value v into variable x) and $o_2 = r(x)$ (read the value from variable x) such that operation o_2 retrieves the value stored by operation o_1. When the initial cr is a smaller value, dependencies might not be satisfied with higher probability. The higher the read rate r_{rate} (i.e., the lower the w_{rate}), the more likely *read-from* relation occurs and higher the R_e. Table 2 summarizes the critical values of cr_0 and $cr_{\sim 0.5\%}$ from the results of R_e in Figs. 3, 4, and 5.

8.3 Impact of Initial cr on m_{ave}

Figs. 6, 7, and 8, each of which corresponds to a certain write rate, illustrate experimental results for average message meta-data size m_{ave} for different initial credits by varying n. We can see that m_s is linearly proportional to initial cr. That is because we only carried out the experiments under initial $credits \leq$ critical cr_0. We call this situation *Incomplete causality* (*IC*). Note that we only run the simulations in *IC* with initial $credits \leq cr_0$. For example, in Fig. 6, the bars start with five variants and end up, for the $credits$ 8, with only 2 (for n = 30 and 40). '8' is larger than the critical $credits$ for $n = 5 \sim 20$. In other words, when the initial $credits$ is up to 8, it guarantees that there is no message apply event violating causal consistency. It does not make sense to run the simulation in this situation.

For any combination of n and initial cr, m_s decreases as w_{rate} increases. This is due to fewer $MERGE$ and more $PURGE$ operations in Opt-Track [11] or Approx-Opt-Track. A read operation will invoke the $MERGE$ function to merge the piggybacked log of the corresponding write to that variable with the local log LOG. Thus, a higher read rate may increase the likelihood that the size of explicit information becomes larger. Furthermore, although a write operation results in the increase of explicit information, it comes with the $PURGE$ function to prune the redundant information, so that the size of LOG could be decreased. Therefore, a higher write rate with a corresponding lower read rate causes fewer $MERGE$ and more $PURGE$ operations generated.

Table 3 lists the analytic data about m_{ave} in cr_0 and $cr_{\sim 0.5\%}$. For the case of 10 processes, $m_s(10)$ is around $0.48 \sim$

TABLE 4
Message Meta-Data Size Saving Rates R_s when R_e is Close to or Equal to Zero

R_e	w_{rate}	the number of processes				
		5	10	20	30	40
$\sim 0.5\%$	0.2	0.287	0.521	0.672	0.582	0.613
	0.5	0.187	0.352	0.534	0.608	0.628
	0.8	0.073	0.289	0.282	0.348	0.412
0	0.2	0.194	0.303	0.294	0.203	0.198
	0.5	0.187	0.202	0.154	0.171	0.145
	0.8	0.016	0.108	0.029	0.021	0.047

TABLE 5
Critical Initial Credits for the Replica Factor Rate of 0.5

	w_{rate}	the number of processes				
		5	10	20	30	40
$R_e \sim 0.5\%$	0.2	3	3	3	4	4
	0.5	3	3	3	3	3
	0.8	2	2	3	3	4
$R_e = 0$	0.2	4	5	6	7	7
	0.5	3	5	6	7	7
	0.8	3	4	5	6	7

TABLE 6
Critical Initial Credits for the Replica Factor Rate of 0.2

	w_{rate}	the number of processes				
		5	10	20	30	40
$R_e \sim 0.5\%$	0.2	1	3	4	4	4
	0.5	3	3	3	4	4
	0.8	3	3	3	3	3
$R_e = 0$	0.2	5	6	8	9	9
	0.5	5	6	7	8	8
	0.8	4	6	7	8	9

0.56 KB for $R_e = 0$. For the case of 20 processes, $m_s(20)$ is around 0.86 $\sim$ 0.93 KB for $R_e = 0$. For the case of 40 processes, $m_s(40)$ is around 1.36 $\sim$ 2.15 KB for $R_e = 0$. Consider $w_{rate} = 0.5$ and $w_{rate} = 0.8$. Using cross-comparison analyses, $m_s(40)/\ m_s(20)/\ m_s(40)$ is less than $m_s\ (10) \times 4/\ m_s(10) \times 2/\ m_s(20) \times 2$. The results reflect the better scalability of Approx-Opt-Track for higher w_{rate} under no risk of violating causal consistency.

8.4 Impact of Initial cr on R_s

This section reports the effectiveness of Approx-Opt-Track to reduce the meta-data overheads under causal consistency. As mentioned before, the meta-data for dependencies could be reduced at the cost of some violations of causal consistency. We intend to study the exact nature of the trade-off between R_s and R_e in *IC*. We expect to find a separation point, where the initial $credits = cr_0$, with a finite initial cr small enough to separate *IC* from *CC* — *Complete Causality*. It can not only reduce the meta-data size, but also have the system fully follow causal consistency.

According to Equation (1), R_s depends on m_s. R_s decreases as m_s increases, which positively depends on initial credits cr. Figs. 9, 10, and 11 present the linear relationships between R_s and initial cr for different w_{rate}. Consistent with the results in Figs. 6, 7, and 8, with larger initial $credits$, m_s will increase (i.e., R_s will decrease). Furthermore, the larger the value of n, the higher the R_s would be. For example, when $n = 5$ with cr being 2, the values of R_s are around 0.2 $\sim$ 0.4 corresponding to different w_{rate}. As n increases, R_s increases, too. When $n = 40$ with cr still being 2, the values of R_s are close to 0.7 $\sim$ 0.9. It shows that Approx-Opt-Track can reduce more meta-data overheads in a larger n under the *IC* state.

For the same number of processes, the curves of R_s versus cr shift downward as w_{rate} increases. It implies that, in the *IC* state, m_s increases more slowly than m_s ($cr = \infty$) does as r_{rate} rises. This is because there are more $MERGE$ operations to delete meta-data entries in higher r_{rate} (lower w_{rate}).

Table 4 summarizes the details of numerical data about R_s in the major and minor critical initial credits. For the case of 40 processes, R_s is around 40% $\sim$ 60% at a very slight cost of violation of causal consistency ($R_e \sim 0.5\%$). R_s reaches around 5% $\sim$ 20% without violating causality order in terms of different write rates. This evidence proves that if the initial allocation of cr is made as a small digit, the message about which the corresponding meta-data entry is deleted would already have reached its destination with very high probability. On the other hand, the simulation results of R_s for R_e being zero in Table 4 reflect the effect of Approx-Opt-Track in the real world on causal consistency. Although it reduces 4.7 percent of the total meta-data in a higher w_{rate}, when $n = 40$, the values of R_s are around 14.5 and 19.8 percent in the lower and medium w_{rate}. From the above comprehensive analyses, it can be concluded that Approx-Opt-Track provides a better network capacity utilization than Opt-Track without causing additional causal violations.

8.5 Impact of Replica Factor Rate r_f on cr_c

This section illustrates the impact of different replica factor rates on cr_0 and $cr_{\sim 0.5\%}$. Tables 5 and 6 present the results of cr_0 and $cr_{\sim 0.5\%}$ for different numbers of processes on the higher replica factor rate ($r_f = 0.5$) and the lower replica factor rate ($r_f = 0.2$). The values of $cr_{\sim 0.5\%}$ for the lower r_f and the higher r_f seem to be almost consistent with that of $cr_{\sim 0.5\%}$ with r_f being 0.3 for each process number for different w_{rate}. Again, as stated above, most of the dependencies will become aged after transmitting the associated meta-data across a few hops. Comparing among the values of cr_0 for the three r_f rates varied from 0.2 to 0.5, it is found that the larger the r_f rate, the smaller the value of cr_0. We believe that the reason is similar to that of the impact of w_{rate} on R_e, as described in Section 8.2. The lower the value of r_f, the fewer the write operations, which means that the read rate is higher. This makes R_e higher. It implies that it requires a slightly larger cr to maintain causal consistency with a lower r_f.

9 Conclusion and Discussion

We considered the problem of providing causal consistency protocols in partially replicated systems. In this paper, we proposed algorithm Approx-Opt-Track and showed theoretically its potential to improve the meta-data size of Opt-Track [11]. By controlling a parameter called $credits$, we can trade-off the level of potential inaccuracy by the size of meta-data. We then considered the performance of the instantiation of the credits by hop count, in detail. There is a trade-off between m_s (or initial cr) and R_e. This paper conducted a performance trade-off analysis of R_e, m_s, and R_s using multi-scale simulations.

The simulation results showed that by controlling initial cr, we can leverage the potential causal consistency inaccuracy to further improve the meta-data overhead. By setting a small finite initial cr, most of the dependencies turn out to be aged after being transmitted across a few hops. For various numbers of processes, varying from 5 to 40, the minor critical initial $credits$ ($cr_{\sim 0.5\%}$) are around 3 $\sim$ 4 with a very

TABLE 7
Impacts of Failures/Partitions Compared to the "no Failure" Case

	Without Fault Tolerance	With Fault Tolerance
enough credits	(A); R_e is the same as no failures ($R_e = 0$; system hangs)	(B); R_e is the same as no failures ($R_e = 0$; no system hangs)
not enough credits	(C); R_e decreases compared to no failures	(D); R_e increases compared to no failures

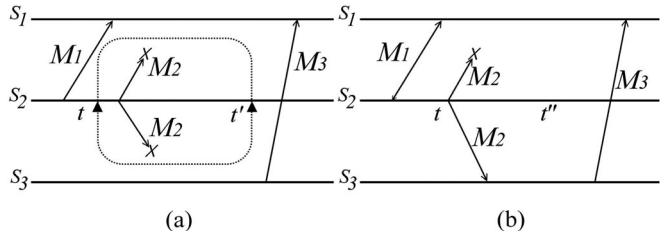

Fig. 12. Examples of failures (a) partition (b) message loss.

low R_e, which is close to 0.5 percent. It concludes that if the initial allocation of cr is made as a finite single-digit, by the time the cr reaches zero, the message corresponding to the meta-data would reach its destination with very high probability.

With a finite initial cr small enough, Approx-Opt-Track was also seen to show significant gains over Opt-Track. In particular, as for 40 processes, Approx-Opt-Track can lower the meta-data size by around $5\% \sim 20\%$ without causal violations and from $40\% \sim 60\%$ for upto 0.5 percent causal violations. Thus, we showed that Approx-Opt-Track can provide less overhead than Opt-Track at little or no cost of violating causal consistency.

The introduction of the concept of credits and their manipulation in the algorithm Approx-Opt-Track does not increase the complexity measures beyond the corresponding values of Opt-Track (see Table 1). The notion of credits can also be used in conjunction with other causal consistency algorithms, including full replication and partial replication algorithms, besides Opt-Track.

In real systems, failures may happen often. Failures/partitions would result in different impacts on violation errors for Approx-Opt-Track in different conditions compared to the "no failure" situations. Table 7 summarizes the impacts of failures/partitions for four different cases for Approx-Opt-Track. First, cases (A) and (B) illustrate the impacts when the initial hop count $credits$ are enough (i.e., Approx-Opt-Track is identical to Opt-Track here). Second, cases (C) and (D) clarify the impacts when the initial hop count $credits$ are not enough.

Case (A). Fig. 12a shows a partition failure under which S_2 splits into an isolated subnetwork from t to t'. A multicast message M_2 cannot be delivered to the destinations. When S_1 receives M_3, M_3 can be immediately applied at S_1 without a causal violation for Approx-Opt-Track (because $send_2(M_1)$ and $send_3(M_3)$ become concurrent). Fig. 12b presents a message loss case where S_1 fails to receive M_2 from S_2. Even if S_1 receives M_3, M_3 will not be applied at S_1 until M_2 is received and applied at S_1 (because sending M_2 causally happens before sending M_3, applying M_3 should causally happen after applying M_2). However, without fault tolerance, S_1 does not receive M_2. Thus, M_3 cannot be applied in S_1 (forever). In case (A), R_e will be zero (i.e., failures will not lead to higher violation errors, compared to the same situation without failure). However, some scenarios cause system hangs.

Case (B). With fault tolerance, if any message loss occurs, it will be resent. In Fig. 12a, S_2 will resend the multicast message M_2 after t'. Based on when M_2 is received at S_3, Approx-Opt-Track can guarantee that there is no violation error for applying M_3 at S_1, by using activation predicate. In Fig. 12b, M_3 will be applied causally after receiving M_2 for the same reason as in Fig. 12a. In case (B), R_e is still zero (i.e., failures will not lead to higher violation errors, compared to the same situation without failure).

Case (C). Without fault tolerance, R_e may decrease. In Fig. 12b, if the $credits$ of the meta-data "M_2 is sent to S_1" (m_d) become zero, m_d will be removed at S_3. Thus, M_3 does not piggyback m_d. Because S_1 does not receive M_2 and M_3 will be applied when receiving M_3, this will not lead to a violation error. However, if no failure, the above violation error might be possible. Therefore, case (C) causes R_e to decrease.

Case (D). With fault tolerance, R_e may increase. In Fig. 12b, if M_2 sent to S_1 at t fails, it will be resent at t''. Because M_3 does not piggyback the meta-data m_d, M_3 will be applied immediately when receiving M_3. The condition of violating causal consistency for applying M_3 depends on whether $apply_1(M_3)$ happens causally before $apply_1(M_2)$. Without loss of generality, sending M_2 at t'' to S_1 has a higher probability of receiving it causally after applying M_3 compared to sending M_2 at t. Thus, fault tolerance may cause R_e to increase.

Our final discussion studies the characteristics of violation errors when a node becomes partitioned 'forever' after sending messages to a subset of destinations. See Fig. 12b, S_2 becomes partitioned forever after sending M_2 to (S_1 and) S_3. In case (B), M_3 cannot be applied after receiving M_3, although M_2 sent to S_1 is missed, and M_2 will not be resent in future. This does not lead to a violation error. Because S_2 becomes partitioned forever, 'with fault tolerance' is equivalent to 'without fault tolerance'. Therefore, case (B) degenerates to case (A). In case (D), M_3 will be applied when receiving M_3, since the dependency on M_2 sent to S_1 is forgotten. This does not lead to a violation error, either. Based on the same reason above, case (D) degenerates to case (C). Although they both do not lead to violation errors in this partition issue, case (A) is not fully equivalent to case (C). In case (A), it is impossible to apply M_3 after receiving M_3. The thread for applying M_3 will hang. However, M_3 will be applied after receiving M_3 (no thread hangs) in case (C). Therefore, long-lasting network partitions make case (B) and (D) degenerate to case (A) and case (C), respectively.

REFERENCES

[1] G. DeCandia, et al., "Dynamo: Amazon's highly available key-value store," *SIGOPS Operating Syst. Rev.*, vol. 41, no. 6, pp. 205–220, Oct. 2007.

[2] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Operating Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[3] 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. 23rd ACM Symp. Operating Syst. Principles*, 2011, pp. 401–416.
[4] A. D. Kshemkalyani and M. Singhal, *Distributed Computing: Principles, Algorithms, and Systems*, 1st ed. New York, NY, USA: Cambridge Univ. Press, 2011.
[5] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002.
[6] S. Almeida, J. A. Leitão, and L. Rodrigues, "ChainReaction: A causal+ consistent datastore based on chain replication," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, 2013, pp. 85–98.
[7] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger semantics for low-latency geo-replicated storage," in *Proc. 10th USENIX Conf. Netw. Syst. Des. Implementation*, 2013, pp. 313–328.
[8] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel, "Orbe: Scalable causal consistency using dependency matrices and physical clocks," in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, pp. 11:1–11:14.
[9] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, "GentleRain: Cheap and scalable causal consistency with physical clocks," in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 4:1–4:13.
[10] P. Mahajan, L. Alvisi, and M. Dahlin, "Consistency, availability, convergence," Comput. Sci. Dept., Univ. Texas at Austin, Austin, TX, USA, Tech. Rep. TR-11–22, May 2011.
[11] M. Shen, A. D. Kshemkalyani, and T. Y. Hsu, "Causal consistency for geo-replicated cloud storage under partial replication," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2015, pp. 509–518.
[12] A. D. Kshemkalyani and M. Singhal, "An optimal algorithm for generalized causal message ordering," in *Proc. 15th Annu. ACM Symp. Principles Distrib. Comput.*, 1996, p. 87.
[13] A. D. Kshemkalyani and M. Singhal, "Necessary and sufficient conditions on information for causal message ordering and their optimal implementation," *Distrib. Comput.*, vol. 11, no. 2, pp. 91–111, 1998.
[14] T. Y. Hsu and A. D. Kshemkalyani, "Performance of causal consistency algorithms for partially replicated systems," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2016, pp. 525–534.
[15] R. Baldoni, A. Milani, and S. T. Piergiovanni, "Optimal propagation-based protocols implementing causal memories," *Distrib. Comput.*, vol. 18, no. 6, pp. 461–474, 2006.
[16] A. D. Kshemkalyani and T.-Y. Hsu, "Approximate causal consistency for partially replicated geo-replicated cloud storage," in *Proc. 5th Int. Workshop Netw.-Aware Data Manage.*, 2015, pp. 3:1–3:8.
[17] T. Y. Hsu and A. D. Kshemkalyani, "Performance of approximate causal consistency for partially replicated systems," in *Proc. Workshop Adaptive Resource Manage. Scheduling Cloud Comput.*, 2016, pp. 7–13.
[18] K. P. Birman and T. A. Joseph, "Reliable communication in the presence of failures," *ACM Trans. Comput. Syst.*, vol. 5, no. 1, pp. 47–76, Jan. 1987.
[19] A. Schiper, J. Eggli, and A. Sandoz, "A new algorithm to implement causal ordering," in *Proc. 3rd Int. Workshop Distrib. Algorithms*, 1989, pp. 219–232.
[20] M. Raynal, A. Schiper, and S. Toueg, "The causal ordering abstraction and a simple way to implement it," *Inf. Process. Lett.*, vol. 39, no. 6, pp. 343–350, Oct. 1991.
[21] P. Chandra, P. Gambhire, and A. D. Kshemkalyani, "Performance of the optimal causal multicast algorithm: A statistical analysis," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 1, pp. 40–52, Jan. 2004.
[22] P. Chandra and A. D. Kshemkalyani, "Causal multicast in mobile networks," in *Proc. 12th Int. Workshop Model. Anal. Simul. Comput. Telecommun. Syst.*, 2004, pp. 213–220.
[23] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto, "Causal memory: Definitions, implementation and programming," *Distrib. Comput.*, vol. 9, no. 1, pp. 37–49, 1995.
[24] R. Baldoni, R. Prakash, M. Raynal, and M. Singhal, "Efficient delta-causal broadcasting," *Int. J. Comput. Syst. Sci. Eng.*, vol. 13, pp. 263–271, 1998.
[25] R. Baldoni, A. Mostefaoui, and M. Raynal, "Causal delivery of messages with real-time data in unreliable networks," *Real-Time Syst.*, vol. 10, no. 3, pp. 245–262, May 1996.
[26] F. Adelstein and M. Singhal, "Realtime causal message ordering in multimedia systems," *Telecommun. Syst.*, vol. 7, no. 1, pp. 59–74, 1997.
[27] F. J. Torres-Rojas, M. Ahamad, and M. Raynal, "Lifetime based consistency protocols for distributed objects," in *Proc. 12th Int. Symp. Distrib. Comput.*, 1998, pp. 378–392.
[28] T. Crain and M. Shapiro, "Designing a causally consistent protocol for geo-distributed partial replication," in *Proc. 1st Workshop Principles Practice Consistency Distrib. Data*, 2015, pp. 6:1–6:4.
[29] M. Shen, A. D. Kshemkalyani, and T. Y. Hsu, "OPCAM: Optimal algorithms implementing causal memories in shared memory systems," in *Proc. Int. Conf. Distrib. Comput. Networking*, 2015, pp. 16:1–16:4.
[30] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

Ta-Yuan Hsu is working toward the PhD degree in the Department of Electrical and Computer Engineering, University of Illinois at Chicago. His research interests include distributed algorithms, social networks, and causal memory. He is a student member of the IEEE.

Ajay D. Kshemkalyani received the BTech degree in computer science from the Indian Institute of Technology, Bombay, in 1987, and the PhD degree in computer science from the Ohio State University, in 1991, respectively. He is currently a professor with the Department of Computer Science, University of Illinois at Chicago. His research interests include distributed computing, computer networks, and concurrent systems. In 1999, he received the US National Science Foundation Career Award. He has served on the editorial board of the Elsevier journal *Computer Networks*, and the *IEEE Transactions on Parallel and Distributed Systems*. He has coauthored a book entitled *Distributed Computing: Principles, Algorithms, and Systems* (Cambridge University Press, 2011). He is a distinguished scientist of the ACM and a senior member of the IEEE.