Performance of Causal Consistency Algorithms for Partially Replicated Systems

Ta Yuan Hsu
Dept. of Electrical and Computer Engineering
University of Illinois at Chicago
Chicago, IL, 60607-7053, USA
thsu4@uic.edu

Ajay D. Kshemkalyani
Dept. of Computer Science
University of Illinois at Chicago
Chicago, IL, 60607-7053, USA
ajay@uic.edu

***Abstract*—Many advances have been made in the design of full replication protocols in distributed systems. Causal consistency in such systems has received great interest. However, most existing works focus on the implementation in full replication because it simplifies designing the algorithm. More recently, interest in full replication has shifted to focus on the development of partial replication protocols which emphasize a better network capacity utilization. In this paper, we present the analytic data to compare the performances of three proposed protocols in partial replication and full replication. We also give simulation results to present the advantage of partial replication over full replication.**

I. Introduction

Distributed systems are commonly designed to improve the deployment of data storages across multiple geographically diverse sites connected by reliable data networks. Due to the independence of hardware maintenance and management, this model for fault tolerance can avoid systematic disasters resulting from operation failures at individual sites [1]. It can also significantly reduce access latency and provide high availability, because replicas are available in close proximity to users in multiple sites. Implementing a consistency model is one of the most important problems for data replication mechanisms. The CAP Theorem [2] states that for a replicated, distributed data store, it can only have at most two of the three features: Consistency, Availability, and Partition tolerance. Besides the above three features, two other desirable features of large-scale distributed data stores are: low Latency (defined as the latency less than the maximum wide-area delay between replicas) and high Scalability [3].

Among several common consistency models in distributed shared memory systems [4], causal consistency is an attractive choice of one of the strongest weaker-than-strong consistency models that satisfies low Latency [3]. More recently, in the past few years, causal consistency has been studied and/or implemented by numerous researchers [5], [6], [7], [8], [9], [10], [3], [11]. 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. Furthermore, there are no experimental or simulation results about the analysis of the performance trade-off between full replication and partial replication.

In [12], three protocols have been proposed for causal consistency. Two of them are designed for partial replication. Another one is designed for full replication. Although partial replication can avoid taking unnecessary network capacity and hardware resources theoretically, it is a challenge to implement partial replication compared with full replication. This is primarily because of the higher complexity and overheads (e.g., additional communication workloads and larger meta-data) of tracking causal dependency between operations. With the increasing usage of multimedia and social networks, where photos and video are uploaded, the average size of the payload increases. Hence, the additional overhead of partial replication is relatively small in comparison with those raw data. Furthermore, for write-intensive workloads, partial replication can offer a direct savings in the number of communication messages without significantly increasing any delay for read operations.

Contributions

This paper is the first work that explores the trade-off between partial replication and full replication analytically. We quantitatively evaluate the performance of the three optimal protocols of [12] for implementing causal consistency under partial replication and under full replication. The three protocols are as follows.

1) The Full-Track protocol is optimal in the sense defined by Baldoni et al. [13]. This protocol primarily reduces the false causality in the partial replica system.
2) The Opt-Track protocol can further optimize the size of the local logs maintained and the amount of dependency information piggybacked on the multicast messages resulting from write operations.
3) The Opt-Track-CRP protocol, derived from Opt-Track, is optimal in a fully replicated system.

We first simulate the Opt-Track and Full-Track protocols within partially replicated systems to compare their performance. We present that Opt-Track outperforms Full-Track in network capacity and shows the advantage in write-intensive workloads. Then, we simulate Opt-Track-CRP and optP [13] within fully replicated systems to compare their efficiency. Opt-Track-CRP 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.

In addition to simulating the performances within partially replicated systems and within fully replicated systems, we also explore the trade-off between partial replication and full replication analytically. We show the advantage of partial replication over full replication.

Organization

Section II outlines the causal memory model. Section III reviews Full-Track, Opt-Track, and Opt-Track-CRP. Section IV presents the communication models for simulating the optimal protocols proposed under partial replication and under full replication. Section V shows all the simulation results and analytically illustrates the performance trade-off between partial replication and full replication. Section VI gives the conclusion.

II. Shared Memory System Model

A. Causally Consistent Shared Memory

The system architecture is based on that proposed by Ahamad et al. [14] and Baldoni et al. [13]. We consider support for n asynchronous application processes ap_1, ap_2, ..., ap_n, interacting through a distributed shared memory Q composed of q variables x_1, x_2, ..., x_q. Application processes run on processors which are distributed across a wide-area network. Each ap_i can execute either a read or a write operation on any of the q variables. When an application process ap_i invokes a read operation, it returns value v stored in variable x_j, denoted as $r_i(x_j)v$. Similarly, when ap_i invokes a write operation to put a new value u in variable x_j, it is denoted as $w_i(x_j)u$. The initial value of each variable is $\perp$.

By executing a sequence 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, it represents that o_1 precedes o_2 under program order, denoted as $o_1 \prec_{po} o_2$. The collection of local histories h_i for all n application processes compose a global history H in which the set of operations is defined as O_H. Operations executed at distinct processes can also be related using the $read\text{-}from\ order$, 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 exist $o_1 = w(x)v$ and $o_2 = r(x)v$ such that $read$ operation o_2 retrieves the value stored by the $write$ operation o_1. As shown in [14],

- 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 variable x and there exists no operation o_1 such that $o_1 \prec_{ro} o_2$, then the return value v is $\perp$.

Given both the $program\ order$ and $read\text{-}from\ order$, the $causality\ order$ $\prec_{co}$ can be defined on the operation set O_H. The causality order typically maintains the transitive closure of the union of local histories' program order and the read-from order. 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 three conditions is satisfied:

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

Significantly, the causality order profiles the partial order on the operation set O_H. All the write operations that are related by the causality order have to be seen by each application process in the order defined by the causality relation.

B. Underlying Distributed Communication System

We now describe the abstract design that supports a distributed shared memory for a non-fully replicated system and applies the principle of causal consistency. Our communication system is implemented on top of the underlying reliable distributed asynchronous message passing system with n sites, each of which hosts an application process ap_i connected by FIFO channels. Each site holds only a subset of variables $x_h \in Q$. For application process ap_i, the subset of variables stored on the site s_i is denoted as X_i. If the replica factor of the shared memory system is p and variables are evenly replicated on all the sites, then the average size of X_i is $\frac{pq}{n}$. The execution of an operation at site s_i generates a sequential event. Application process ap_i is in charge of performing the actual operation events at site s_i.

To implement the read and write operations in the distributed shared memory communication abstraction, the underlying message passing system provides several primitives to enable the communication between different sites. For the write operation, whenever application process ap_i executes $w(x_i)v$, it utilizes the reliable **Multicast**(m) primitive to deliver the message m including $w(x_i)v$ to all sites that replicate the variable x_i. For the read operation, there are two different scenarios. First, application process ap_i executing read operation $r(x_2)u$ needs to read the value of x_2 from a remote replicated site when x_2 is not locally replicated in site s_i. In such a case, it invokes the reliable **RemoteFetch**(m) primitive to send the message m including $r(x_2)$ to a pre-designated site replicating x_2 to fetch the value of x_2. This is a synchronous process. It indicates that the fetch primitive will block until it returns the variable's value. Second, if the variable x_2 to be read is locally replicated, then application process ap_i simply returns the corresponding local value.

The read and write operations executed by the application processes formally generate six kinds of $events$ in the underlying communication message passing system. They are generated at each site as shown in the following list.

- $Send\ event$. The invocation of **Multicast**(m) primitive by application process ap_i generates event $send_i(m)$.
- $Fetch\ event$. The invocation of **RemoteFetch**(m) primitive by application process ap_i generates event $fetch_i(m)$.

- $Message\ receipt\ event$. The receipt of a message m at site s_i generates event $receipt_i(m)$. The delivered message m can correspond to either a $send_j(m)$ event or a $fetch_j(m)$ event sent from site s_j.
- $Apply\ event$. When applying the value written by the operation $w_j(x_h)v$ to the local variable x_h replicated at site s_i, an event $apply_i(w_j(x_h)v)$ is generated by ap_i.
- $Remote\ return\ event$. After the occurrence of event $receipt_i(m)$ corresponding to the remote read operation $r_j(x_h)u$ executed by ap_j, an event $remote\_return_i(r_j(x_h)u)$ is generated to transmit variable x_h's value u to site s_j.
- $Return\ event$. Return event $return_i(x_h, v)$ corresponds to the return of variable x_h's value either fetched remotely through a requested $fetch_i(x_h)$ or read from the local replica.

To implement the causal distributed shared memory abstraction, when an update message corresponding to a write operation $w_j(x_h)v$ is received at site s_i, a new thread will be invoked to determine when to locally apply the update access. The condition that the update access is ready to be applied locally is defined as the activation predicate in [13].

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 access will be halted until the activation predicate A 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.

III. Optimal Protocols

We designed two protocols, Full-Track and Opt-Track, implementing causal memories in a partially replicated distributed shared memory system [12], both of which adopt the optimal activation predicate A_{OPT}. A new relation, $\rightarrow_{co}$, defined by Baldoni et al. on $Send\ events$ can be used to demonstrate A_{OPT}. The relation defined by $\rightarrow_{co}$ is actually a subset of Lamport's "happened before" relation [15], denoted by $\rightarrow$. Opt-Track is a message and space optimal protocol for a partially replicated system. Subsequently, as a special case of this algorithm, we derived another optimal protocol - Opt-Track-CRP - for the fully replicated case [12], that is optimal and has lower message size, time, and space complexities than the Baldoni et al. approach [13].

A. Full-Track Protocol [12] for Partially Replicated Systems

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 **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 delivery 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 **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. At each site s_i, the following data structures are maintained:

1) $Write_i[n \times n]$: the $Write$ clock. $Write_i[j, k] = c$ 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 c.
2) $Apply_i[1 \sim n]$: an array of integers. $Apply_i[j] = c$ means that a total number of c updates written by application process ap_j have been applied at site s_i.
3) $LastWriteOn_i\langle$variable id, $Write\rangle$: a hash map of Write clocks. $LastWriteOn\langle h\rangle$ stores the $Write$ clock value associated with the last write operation on variable x_h which is locally replicated at site s_i.

B. Opt-Track Protocol [12] for Partially Replicated Systems

Full-Track protocol 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 [16],[17] (hereafter referred to as the KS algorithm). The KS algorithm aims at reducing the message size and storage cost for causal message ordering abstractions 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 explicitly keeps a record of recently received messages from each other site. Such explicit information is used to track M as long as (i) and (ii) are false.

(i) the message M is delivered to the destination;
(ii) a message has been sent to the destination in the causal future of $send(M)$.

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.

Although the KS algorithm is dedicated to message passing systems, its essential idea of deleting unnecessary dependency information still applies to distributed shared memory systems. We can adapt the KS algorithm to a partially replicated distributed 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 utilized 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 implicit conditions.

1) When an update m derived from 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 kept in the causal future of the event $apply_2(w)$. It implies that m has been delivered to s_2, to clean the logs at other sites in the causal future of event $return_2(x, v)$.
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 reality, 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 associated with a message m'' sent causally later to s_2 can infer the update m has already been applied at s_2. It implies that m has been delivered to s_2, to clean the logs at other sites in the causal future of events $return_k(y, v')$.

Such implicit information can be used to learn when explicit information (i) or (ii) becomes true and can be deleted. 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) $Apply_i[1 \sim n]$: an array of integers. $Apply_i[j]$ = c means that a total number of c 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 $LOGs$. $LastWriteOn\langle h\rangle$ stores the piggybacked LOG from the most recent update applied at site s_i for locally replicated variable x_h.

C. Opt-Track-CRP Protocol [12] : Adapting Opt-Track Protocol to Fully-Replicated Systems

Opt-Track protocol can be directly applied to fully replicated causal distributed shared memory systems. Furthermore, since in the full replication case, every write operation will be sent to exactly the entire set of sites, 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 the second implicit condition. Also, when a write operation is received, all the write operations it piggybacks also have the receiver as part of their destinations.

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 $clock_i$ is the local write operation counter at site id 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 protocol's scalability under full replication.

Practically, Opt-Track protocol's scalability can be further improved in the fully replicated scenario. Under the partially replicated system, it is important to keep entries with empty destination list as long as they represent the most recent updates applied from some site. This 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. Under the fully replicated scenario, we can also decrease this cost. 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 a write operation w' updating the variable x_h is applied at site s_i, only w' itself needs to be stored in $LastWriteOn_i\langle x_h\rangle$.

For maintaining local logs, 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 incurs 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 retrieve variables that are updated by the same application process, only the entry associated with the very last read operation needs to be kept 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 distributed shared memory system is write-intensive, then the local log will be reset at the frequency of write operations generated at each site. It means that each site simply cannot invoke enough read operations to maintain the local log to reach a number of n entries. Even if the application is read-intensive, only a limited subset of all the sites 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.

For the fully replicated causal distributed shared memory, each site still maintains the same data structures as in Opt-Track. 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 $\langle i, clock_i \rangle$. The average message size in Opt-Track-CRP is significantly less than that in [13], as we will show in Section V.

IV. Simulation System Model

We consider an asynchronous distributed system. Formally, a distributed system is composed of a finite set of asynchronous processes running on multiple sites which are distributed over a wide area and interconnected through a communication network. To simplify and without loss of generality, we assume that there is only one process on each site. Each site has a local memory and can communicate by asynchronous message passing through TCP channels of the underlying network. The communication network is reliable and transmits messages in FIFO order without omissions or duplications.

A. Process Model

1) Partial Replication: A process consists of two major subsystems viz., the *application subsystem* and *message receipt subsystem*. The application subsystem is responsible for the functionality of scheduling operation events (write/read) including two major functions, viz., $Write$ and $Read$. The message receipt subsystem is responsible for responding to remote request service, including two major functions, viz., *Applying Multicast* and *Responding Fetch*. The application subsystem executes write and read events. It also maintains a floating point local clock to generate operation patterns based on a temporal schedule. For a write operation $w(x_h)v$, the application subsystem delivers the message $m(w(x_h)v)$ and the corresponding meta-data – local log L_w (in **Opt-Track protocol**) or local $Write$ clock (in **Full-Track protocol**) to other replicas. For a read operation $r(x_h)$, the application subsystem returns the local variable x_h's value or sends a fetch message $fetch(x_h)$ to a predesignated site to get the remote variable x_h's value as well as the corresponding meta-data $LastWriteOn\langle h \rangle$.

The message receipt subsystem monitors two kinds of incoming messages. First, for a multicast message $m(w(x_h)v)$, the message receipt subsystem determines when to apply a new value v to the variable x_h in causal consistency and then update the meta-data $LastWriteOn\langle h \rangle$. Second, for a remote fetch message $m(fetch(x_h))$, it simply replies with the local value of the variable x_h and the corresponding meta-data $LastWriteOn\langle h \rangle$ to the requesting site.

Message structure. A message is the fundamental entity that delivers information from a sender process to one or more receiver processes. The structures of different kinds of messages typically follow the data structures shown in Section III. In partial replication protocols, there are three distinct messages. Their structures are as shown in the Table I. SM corresponds to a multicast message generated by $send\ event$ to deliver the information of updating variable's value. FM is a fetch message caused by a $fetch\ event$. RM represents a remote return message to respond to a remote read operation.

TABLE I
Message Meta-Data Structures in Partial Replication Protocols

	Full-Track	Opt-Track
SM (Multicast)	x_h,v,$Write$	x_h,v,$Site_{id}$,$clock$,L_w
FM (Fetch)	x_h,v	x_h,v
RM (Remote Return)	v,$LastWriteOn\langle h \rangle$	v,$LastWriteOn\langle h \rangle$

The Full-Track protocol imposes an $n \times n$ $Write$ matrix structure as part of the piggybacked meta-data in SM and RM messages. The Opt-Track protocol utilizes a list log to maintain causal ordering information in SM and RM messages.

2) Full Replication: Similarly, a process also consists of two major subsystems viz., the *application subsystem* and *message receipt subsystem*, in the full replication protocols. The functionalities of the two subsystems are very similar to those of subsystems in partial replication protocols. The application process consists of $Write$ and $Read$ functions. Specially, the function $Read(x_h)$ only needs to merge the local piggybacked log $LastWriteOn\langle h \rangle$ into the local log LOG and then return the local value of x_h. Besides, the message receipt subsystem only handles the function of *Applying Multicast* to determine when to apply a write update.

Message structure. There is only one message type – SM message corresponding to a write operation for multicasting – in Opt-Track-CRP. SM message is represented by $m(x_h, v, Site_{id}, clock, LOG)$.

B. Simulation Parameters

The system parameters whose effects we examine on the performance of the Opt-Track partial replication protocol and the Opt-Track-CRP full replication protocol are as follows:

- **Number of Processes** (n): In reality, a desirable causal consistency algorithm must show good performance for a large number of processes. It may be necessary to simulate any causal consistency algorithm over a wide range of the number of processes. The limitation of the number of processes in the system depends on the processor model and memory allocation of the benchmark device running the simulation. On an Intel Core 2 Duo computer with 16 GB DDR2 and the simulation framework being implemented in JDK8, we can simulate up to 40 processes.
- **Number of Variables** (q): It is the number of variables in the distributed shared memory system. In a super cloud storage, q is unbound. In other words, it is also necessary to set q to be a large number. Due to the memory limitation, q we used in the benchmark experiment is one hundred.

- **Replication Factor** (p): It is defined as the number of sites at which each variable is replicated. In full replication protocols, $p = n$. In partial replication protocols, we set p equal to $0.3 \times n$.
- **Write Number** (w): The number of write operations performed in the distributed system.
- **Read Number** (r): The number of read operations performed in the distributed system.
- **Write Rate** (w_{rate}): It is defined as $\frac{w}{w+r}$. Binding by a variety of write rates, we can study performance for read-intensive and write-intensive application workloads.
- **Log Size** (d): The number of write operations stored in local log (used only in the analysis of Opt-Track-CRP).
- **Message Count** (m_c): The total number of messages generated by all the processes.
- **Message Size** (m_s): The total size of all the messages generated by all the processes. It is the most important performance indicator for the benchmark simulation.

C. Process Execution

All the processes in the system are symmetric and generate operation events (write event or read event) according to a event schedule planned in advance. The event schedule is randomly generated. The time interval between two events is given from $5ms$ to $2005ms$. The processes in the distributed system execute concurrently. However, simulating each process as an independent process at a site invoked interprocess communication.

When a process gets initialized, it first invokes the message receipt subsystem. Then, the system executes $Scheduled\text{-}ExecutorService$ in JDK to drive the application subsystem 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. While TCP exploits $slow\ start$ to retransmit the lost packets, which can lead to high value of transmission time, it guarantees that messages can be successfully transmitted without any loss.

An application subsystem stops generating operation events once it runs out of all the scheduling events and flags its status as finished. The simulation stops when all the application subsystems have their status set to 'finished'.

V. Simulation Results and Discussion

The implementations of the four causal consistency replicated protocols – Full-Track, Opt-Track, Opt-Track-CRP identified in Section III, and optP in [13] – were realized in the framework presented in Section IV. The communication framework and the core algorithms were implemented in JDK 8. The performance metrics used are as follows:

- The ratio of the total message cost for Full-Track vs. Opt-Track and Opt-Track-CRP vs. optP.
- the average size of the messages transmitted in different causal consistency replicated protocols.

We report two experiments for each protocol, in each of which we vary one of the two parameters n and w_{rate}, respectively. For each combination of parameters in each experiment, multiple runs were performed for each protocol. The experimental results of all the runs did not have more than one percent variation. Thus, only the mean of the multiple runs is represented for each combination.

Each simulation execution runs $600n$ operation events totally. Experimental data was stored after the first 15 % operation events to eliminate the side effect in startup.

A. Partial Replication Protocols

In the Full-Track protocol, each write operation will only incur $(p-1) + \frac{n-p}{n}$ number of SM messages. Some read operations might need to communicate with a remote site. Assume that the variables are evenly replicated across the entire system. Then, an additional $2r\frac{(n-p)}{n}$ number of FM and RM messages will be generated. In total, the message count complexity of the Full-Track protocol is $((p-1) + \frac{n-p}{n})w + 2r\frac{(n-p)}{n}$. Since the size of FM is a constant byte count c, the total message size in Full-Track protocol almost depends on the size of SM and RM. Furthermore, the Full-Track protocol needs to maintain a local $Write$ clock matrix, as shown in Section III-A, of size n^2. This $Write$ clock needs to be piggybacked with each SM or RM message. Hence, the total message size complexity of the Full-Track protocol is $O(n^2pw + nr(n-p))$.

The Opt-Track protocol is an optimization on top of the Full-Track protocol. As Opt-Track runs the same message pattern for both the read and write operations as the Full-Track protocol, its message count complexity is also the same, being $((p-1) + \frac{n-p}{n})w + 2r\frac{(n-p)}{n}$. Instead of maintaining a matrix clock, the Opt-Track protocol keeps a log of only the necessary write operations that happened in the causal past. Although the upper bound on the size of the log is $O(n^2)$, Chandra et al. [18] showed through extensive simulations that the amortized log size is almost $O(n)$. It means that the local log at each site will only incur an amortized storage cost of $O(n)$. Thus, the amortized message size complexity of Opt-Track is only $O(npw + r(n-p))$.

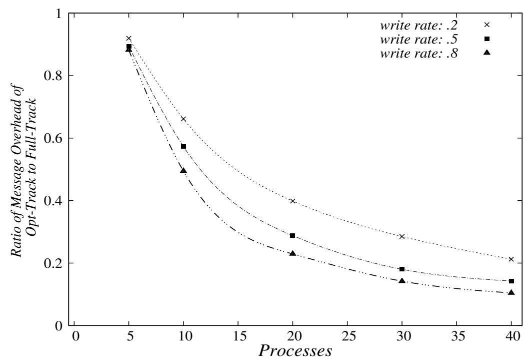

Fig. 1. Total message meta-data space overhead as a function of n and w_{rate} in partial replication protocols.

1) Scalability as a function of n: The number of processes was varied from 5 up to 40. The w_{rate} is set to be 0.2 (lower write rate), 0.5 (medium write rate), and 0.8 (higher write rate), respectively. The results for the ratio of message space overhead (meta-data size) of Opt-Track to Full-Track are shown in Fig. 1. With increasing n, the space overhead ratio rapidly decreases. For the case of 40 processes, for all the simulations of Opt-Track, the overheads are only around $10 \sim 20$ percent those of Full-Track on different write rates. For the case of 5 processes, the overheads reported for Opt-Track for different write rates are around 90 percent of the ones of Full-Track, but the overhead of Full-Track itself is low for such a parameter setting. It can also be seen from Fig. 1 that a higher write rate magnifies the difference of the message space overhead between Opt-Track and Full-Track.

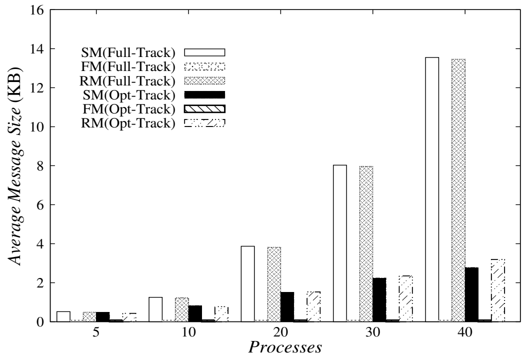

Fig. 2. Average message meta-data space overhead as a function of n with lower w_{rate} (0.2) in partial replication protocols.

2) Impact of write rate w_{rate}: The results for the average message space overhead are shown in Figs. 2, 3, and 4 according to different write rates, respectively. As discussed before, the average message overheads of FM in Opt-Track and Full-Track protocols are constant, very small, and the same, regardless of write rates. In Full-Track protocol, the average message space overheads of SM and RM quadratically increase with n based on our previous discussion. However, the increasingly lower overheads of SM and RM in Opt-Track protocol can be seen from the results. Their overheads appear almost linear in n. This observation can be explained as follows: Although more explicit information of type "s_i is a destination of message m" needs to be maintained in the logs,

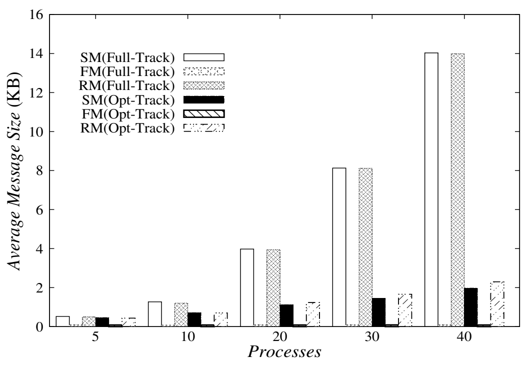

Fig. 3. Average message meta-data space overhead as a function of n with medium w_{rate} (0.5) in partial replication protocols.

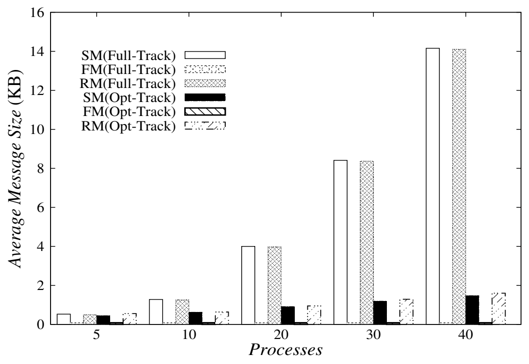

Fig. 4. Average message meta-data space overhead as a function of n with higher w_{rate} (0.8) in partial replication protocols.

TABLE II
AVERAGE SM AND RM SPACE OVERHEAD FOR FULL-TRACK AND OPT-TRACK (KB)

		w_{rate}	the number of processes				
			5	10	20	30	40
Opt-Track	SM	0.2	0.489	0.828	1.512	2.241	2.783
		0.5	0.464	0.715	1.125	1.442	1.976
		0.8	0.450	0.627	0.914	1.194	1.475
	RM	0.2	0.432	0.774	1.530	2.351	3.184
		0.5	0.436	0.702	1.235	1.656	2.197
		0.8	0.555	0.632	0.948	1.288	1.599
Full-Track	SM	0.2	0.518	1.252	3.870	8.028	13.547
		0.5	0.522	1.271	3.975	8.127	14.033
		0.8	0.524	1.275	3.988	8.410	14.157
	RM	0.2	0.493	1.220	3.817	7.959	13.461
		0.5	0.497	1.205	3.941	8.117	13.983
		0.8	0.499	1.250	3.966	8.369	14.099

each log also involves more implicit information. In [12], it showed that additional implicit information provides incentive for the Propagation Constraints to merge and prune the logs when SM or RM messages are received. The observation from Figs. 2 to 4 demonstrates the scalability of Opt-Track under partial replication.

Furthermore, under the same number of processes, we also compare the average SM and RM message sizes in different write rates. (The FM message size is an invariant constant count that is independent of n and w_{rate}. In Full-Track and Opt-Track, their FM sizes are the same since they use the same data structure for FM message.) The analytic data is listed in Table II according to Figs. 2 to 4. The average SM and RM overheads decrease as the write rate increases in Opt-Track Protocol. According to algorithm Opt-Track in [12], the reason can be explained as follows. A read operation will invoke a $MERGE$ 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 LOG is enlarged. Furthermore, although a write operation results in the increase of explicit information, it comes with a $PURGE$ 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 results in fewer $MERGE$ and more $PURGE$ operations generated. The simulation results show that the Opt-Track

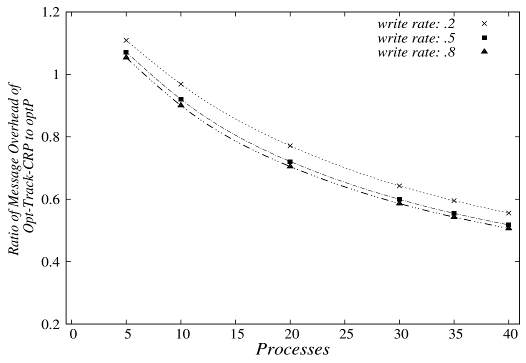

Fig. 5. Total message meta-data space overhead as a function of n and w_{rate} in full replication protocols.

has a better utilization of network capacity in write-intensive workloads than in read-intensive ones. On the other hand, in Full-Track, although the average SM and RM overheads increase as the write rate does, the increase percentage is only $3\% \sim 1\%$.

From the above analysis, it can be concluded that the implementation of the Opt-Track protocol has a better network capacity utilization and better scalability than Full-Track. These improvements increase in a higher write-intensive workload.

Note that our simulating numerical data obtained for Opt-Track protocol does not completely follow the data structures defined in Section III-B. In partial replication protocols, we use a matrix clock ($Write$) in Full-Track and a linked list log (LOG) in Opt-Track to track the causal relation. These two data structures occupy the majority of their corresponding meta-data respectively. Thus, they dictate the trade-off between Full-Track and Opt-Track. Our simulation platform is based on JAVA program. $Write$ clock can be realized by a primitive JAVA integer class matrix. On the other hand, if one wants to realize LOG log format as shown in Section III-B, it is necessary to create a user-defined JAVA class list. However, a user-defined class has some additional overhead against a primitive class in JAVA. Instead of using a user-defined list where each entry contains $\langle j, clock_j, Dests\rangle$, we used three primitive class lists to maintain $\langle j\rangle, \langle clock_j\rangle, \langle Dests\rangle$ in our simulation.

B. Full Replication Protocols

The Opt-Track-CRP protocol is a special case of the Opt-Track protocol dedicated to the full replication scenario. Each write operation incurs a total of $n-1$ SM messages, while the read operation will always read from the local replica and thus generate no messages. In total, the message count complexity of the Opt-Track-CRP protocol is $(n-1)w$. Since Opt-Track-CRP does not need to store the destination list for each record of a write operation, there are no n entries in the local log at each site. This means that the size of the local log is only determined by the number d of entries in the local log. In practice, d is a constant value. Thus the message overhead for each SM corresponding to a write operation is $O(d)$. The

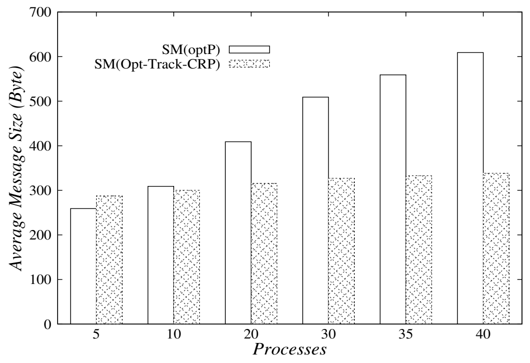

Fig. 6. Average message meta-data space overhead as a function of n with lower w_{rate} (0.2) in full replication protocols.

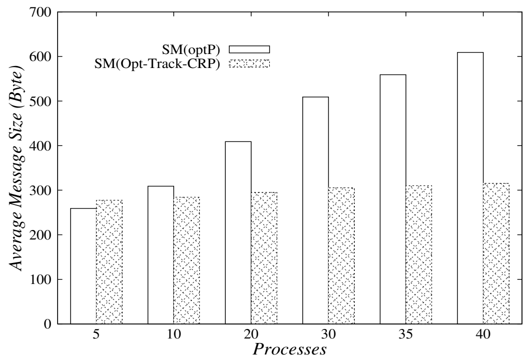

Fig. 7. Average message meta-data space overhead as a function of n with medium w_{rate} (0.5) in full replication protocols.

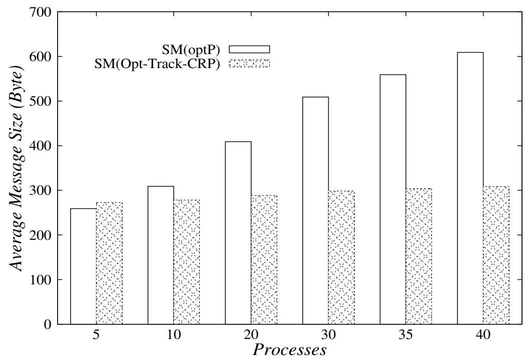

Fig. 8. Average message meta-data space overhead as a function of n with higher w_{rate} (0.8) in full replication protocols.

total message space overhead complexity of the Opt-Track-CRP is $O(nwd)$. The optP proposed by Baldoni et al. [13] is a complete replication based causal consistency protocol. However, optP incurs a higher overhead in the message space complexity. The reason is that optP requires to maintain a $Write$ clock of size n. Since optP has the same message pattern for the write and read operations as Opt-Track-CRP, the message count complexity of the optP is also $O(nw)$. The total message space overhead is $O(n^2w)$.

1) Scalability as a function of n: The results for the ratio of message space overhead of Opt-Track-CRP to optP are

TABLE III
AVERAGE SM SPACE OVERHEAD FOR OPT-TRACK-CRP (BYTE)

n	w_{rate}=.2	w_{rate}=.5	w_{rate}=.8	optP
5	287.3	277.5	272.9	259
10	300.3	284.3	278.2	309
20	315.5	294.9	288.3	409
30	327.1	305.2	298.4	509
35	332.8	310.1	303.4	559
40	338.4	315.3	308.4	609

TABLE IV
TOTAL MESSAGE COUNT FOR PARTIAL REPLICATION (OPT-TRACK) VS. FULL REPLICATION (OPT-TRACK-CRP)

n	Full Replication			Partial Replication		
	(0.2)	(0.5)	(0.8)	(0.2)	(0.5)	(0.8)
5	2,036	4,960	8,004	3,208	3,463	3,764
10	8,910	22,266	35,892	8,297	10,234	12,156
20	38,057	95,114	151,905	22,808	35,668	48,128
30	86,826	217,181	347,304	42,600	75,679	108,810
40	156,156	390,039	624,390	69,405	130,572	192,883

shown in Fig. 5. With increasing n, the ratio of total SM space overhead of Opt-Track-CRP vs. optP decreases. For the case of 5 processes, the total SM overheads for Opt-Track-CRP are consistently higher than around 10% of those for optP on a variety of write rates. For the case of 10 processes, the SM space overhead for Opt-Track-CRP is still close to that for optP in a lower write rate 0.2. But their space overhead ratio is down to 90 percent in a higher write rate 0.8. When the number of processes is up to 40, the SM space overheads for Opt-Track-CRP are around 50% to 55% for different write rates.

2) Impact of write rate w_{rate}: As with partial replication protocols, a higher write rate makes the total message space overhead ratio of Opt-Track-CRP vs. optP smaller. The results for the average SM space overhead are shown in Figs. 6, 7, and 8 in terms of different write rates. As mentioned before, the average SM space complexity of Opt-Track-CRP is $O(d)$ but that of optP is $O(n)$. According to Figs. 6 to 8, Table III presents the analytic data. Obviously, the average SM space overhead of optP only depends on the number of processes n, irrespective of w_{rate}. However, under the same number of processes, the SM space overheads of Opt-Track-CRP decrease slightly with increasing w_{rate}. This can be explained as follows: In Opt-Track-CRP protocol, a write operation does not make the local log size larger than one and change the remote log size at a receiving site. But a read operation might incur a growth in the local log size when it often reads different variables updated via other remote sites. Therefore, lower write rate (corresponding to higher read rate) would cause higher meta-data overhead than higher write rate (corresponding to lower read rate). In other words, Opt-Track-CRP protocol has a better utilization of network capacity in write-intensive workloads than in read-intensive ones.

From the experimental analysis in full replication, we can conclude that Opt-Track-CRP protocol has a better scalability and utilization than optP, especially in write-intensive workloads.

C. Discussion

Compared with the existing causal distributed shared memory protocols, our suite of protocols [12] has the additional ability to implement causal consistency in partially replicated distributed shared memory systems. Further, the protocols in [19], [20], [21] do not provide scalability as they use a form of log serialization and exchange to implement causal consistency.

The advantage of implementing partial replication compared with full replication lies in multiple aspects. First, this could reduce the number of messages for updating or accessing remote variables. Although the read operation may incur additional messages, the overall number of messages can still be lower than that of full replication if the replica factor is low. Hadoop HDFS and MapReduce is one such example. The HDFS framework typically chooses a small number as the replica factor even when the size of the system is large. Furthermore, the MapReduce framework is dedicated to data locality optimization to allocate tasks that read only from the local replicas. In such a system, partial replication generates much less messages than full replication.

Based on the message count formulas of Opt-Track (partial replicated protocol) and Opt-Track-CRP (full replication protocol) shown in the previous subsection, partial replication gives a lower message count than full replication if

$$((p-1)+\frac{(n-p)}{n})w+2r\frac{(n-p)}{n} < (n-1)w \Rightarrow w > 2\frac{r}{n-1} \quad (1)$$

w_{rate} is defined as $\frac{w}{w+r}$. Then, this equation can be expressed as follows:

$$w_{rate} > \frac{2}{1+n} \quad (2)$$

Clearly, this formulates the necessary condition for which partial replication has a lower message count.

Table IV shows the results of running the same operation event scheduling using Opt-Track-CRP and Opt-Track, respectively. It presents the total message counts with different write rates in full replication and partial replication. Except for when n=5 and w_{rate}=0.2, the message counts for partial replication are always less than the ones for full replication. The results in Table IV are in line with the necessary condition – equation (2).

Partial replication can also help to reduce the total size of messages transmitted within the system. Although the two partial replication protocols proposed might have a higher message size complexity compared with their counterparts for full replication, this complexity measurement is only for the control meta-data 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. In 2012, Johnson et al. [22] addressed that the average web page size in 2011 was 679KB, which is apparently much larger than the average meta-data size of Opt-Track. 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, 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.

VI. Conclusion

We considered the problem of providing causal consistency protocols in large-scale storage systems under partial replication and full replication. Two optimal protocols – Opt-Track under partial replication and Opt-Track-CRP under full replication – have been proposed and proven theoretically their optimality. However, there is no performance data available, and an analytical analysis or comparison of their performance. Hence, this paper conducted a performance analysis of the message space and message count complexity of them under a wide range of system conditions using simulations.

The simulations considered two partial replication protocols (Full-Track and Opt-Track) and two full replicated protocols (Opt-Track-CRP and optP), and examined the performance by varying the write rate and the number of processes. Opt-Track was seen to show significant gains over Full-Track in partial replication. In full replication, the results also supported that Opt-Track-CRP performed better than optP in scalability and network capacity utilization. In particular, as the size of the system increased to 40 processes, the two optimal protocols performed very well and have lower meta-data overheads under high write-intensive workloads. This paper is also the first such work that explored the trade-off between partial replication and full replication analytically. We showed the advantage of partial replication and provided the conditions under which partial replication can provide less overhead (transmission and storage) than full replication.

References

[1] I. Iliadis, D. Sotnikov, P. Ta-Shma, and V. Venkatesan, "Reliability of geo-replicated cloud storage systems," in *Dependable Computing (PRDC), 2014 IEEE 20th Pacific Rim International Symposium on*, Nov 2014, pp. 169–179.

[2] 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.

[3] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual consistency," *Commun. ACM*, vol. 57, no. 5, pp. 61–68, 2014.

[4] A. D. Kshemkalyani and M. Singhal, *Distributed Computing: Principles, Algorithms, and Systems*, 1st ed. New York, NY, USA: Cambridge University Press, 2011.

[5] S. Almeida, J. a. Leitão, and L. Rodrigues, "Chainreaction: A causal+ consistent datastore based on chain replication," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 85–98.

[6] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "The potential dangers of causal consistency and an explicit solution," in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC '12. New York, NY, USA: ACM, 2012, pp. 22:1–22:7.

[7] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Bolt-on causal consistency," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 761–772.

[8] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel, "Orbe: scalable causal consistency using dependency matrices and physical clocks," in *ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013*, 2013, pp. 11:1–11:14. [Online]. Available: http://doi.acm.org/10.1145/2523616.2523628

[9] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, "Gentlerain: Cheap and scalable causal consistency with physical clocks," in *Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, November 03 - 05, 2014*, 2014, pp. 4:1–4:13. [Online]. Available: http://doi.acm.org/10.1145/2670979.2670983

[10] K. Lady, M. Kim, and B. D. Noble, "Declared causality in wide-area replicated storage," in *33rd IEEE International Symposium on Reliable Distributed Systems Workshops, SRDS Workshops 2014, Nara, Japan, October 6-9, 2014*, 2014, pp. 2–7.

[11] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger semantics for low-latency geo-replicated storage," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 313–328.

[12] M. Shen, A. D. Kshemkalyani, and T. Y. Hsu, "Causal consistency for geo-replicated cloud storage under partial replication." in *IPDPS Workshops*. IEEE, 2015, pp. 509–518.

[13] R. Baldoni, A. Milani, and S. Tucci-piergiovanni, "Optimal propagation based protocols implementing causal memories," *Distributed Computing*, vol. 18, no. 6, pp. 461–474, 2006.

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

[15] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[16] A. Kshemkalyani and M. Singhal, "Necessary and sufficient conditions on information for causal message ordering and their optimal implementation," *Distributed Computing*, vol. 11, no. 2, pp. 91–111, Apr. 1998.

[17] A. D. Kshemkalyani and M. Singhal, "An optimal algorithm for generalized causal message ordering," in *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '96. New York, NY, USA: ACM, 1996, pp. 87–.

[18] 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, 2004.

[19] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng, "PRACTI replication," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2006.

[20] P. Mahajan, L. Alvisi, and M. Dahlin, "Consistency, availability, convergence," Computer Science Department, University of Texas at Austin, Tech. Rep. TR-11-22, May 2011.

[21] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers, "Flexible update propagation for weakly consistent replication," *SIGOPS Oper. Syst. Rev.*, vol. 31, no. 5, pp. 288–301, Oct. 1997.

[22] D. L. Johnson, E. M. Belding, and G. van Stam, "Network traffic locality in a rural african village," in *Proceedings of the Fifth International Conference on Information and Communication Technologies and Development*, ser. ICTD '12. New York, NY, USA: ACM, 2012, pp. 268–277.