

**Cost-Effective Protocols for Enforcing Causal Consistency in
Geo-Replicated Data Store Systems**

by

Ta-Yuan Hsu

B.A. (National Central University)

M.S. (Illinois Institute of Technology)

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Chicago, 2021

Chicago, Illinois

Defense Committee:

Prof. Ajay D. Kshemkalyani, Chair and Advisor

Prof. Chris Kanich, Computer Science

Prof. Balajee Vamanan, Computer Science

Prof. Zhao Zhang

Prof. Zhichun Zhu

Copyright by

Ta-Yuan Hsu

2021

ACKNOWLEDGMENTS

I want to thank my advisor Prof. Ajay Kshemkalyani for guiding and supporting me over the years. His mentorship has gone a long way towards providing me with an engaging and scientifically enriching experience. His patient guidance and insightful advice have lighted the path for me to accomplish my research goals. I would like to express my deepest gratitude to my advisor for his continuous support, motivation, and immense knowledge throughout my life at UIC. I would like to also thank my other committee members, Prof. Balajee Vamanan, Prof. Chris Kanich, Prof. Zhao Zhang, and Prof. Zhichun Zhu for their support and insights towards my research.

Finally, I also want to thank my parents and my roommates for all of the support through this journey during the most difficult time. Their support is what makes this thesis possible.

.

CONTRIBUTION OF AUTHORS

Chapter 1 presents the background of the study, statement of the problem, objectives of the study, and organization of the thesis.

Chapter 2 surveys previous studies which are related to my research.

Chapter 3 is published in (1) having co-authors of my advisor, Prof. Ajay D. Kshemkalyani, and Dr. Min Shen. I am the first author and responsible for all the figures, calculations, and writings in Chapter 3. Prof. Ajay D. Kshemkalyani contributed to the writing of the papers, in addition to the planning and structure of the work. The JAVA peer-to-peer emulation platform was designed and performed by me. Min Shen contributed to the design of the framework and the implementation of the illustrative example.

Chapter 4 is published in (2) having co-author of Prof. Ajay D. Kshemkalyani. I am the first author and responsible for all the figures, calculations, and writings in Chapter 4. Prof. Ajay D. Kshemkalyani contributed to the writing of the papers, in addition to the planning and structure of the work. The JAVA peer-to-peer emulation platform was designed and performed by me.

Chapter 5 is published in (3) having co-author of Prof. Ajay D. Kshemkalyani. I am the first author and responsible for all the figures, algorithm design, and writings in Chapter 5. Prof. Ajay D. Kshemkalyani contributed to the writing of the papers, in addition to the planning and structure of the work. The JAVA CloudSim platform was designed and performed by me. I also contributed to the realistic data workload crawling from Twitter.

CONTRIBUTION OF AUTHORS (Continued)

Chapter 6 presents CaDRoP, Causal consistency in Dynamic Replication Optimized Protocol. I am responsible for algorithm design, simulation implementation, and performance evaluation. The JAVA CloudSim platform was designed and performed by me.

Chapter 7 concludes this thesis work and proposes future work.

TABLE OF CONTENTS

<u>CHAPTER</u>		<u>PAGE</u>
1	INTRODUCTION	1
1.1	Problem Motivation	2
1.1.1	Causal Consistency Algorithms For Partially Replicated Sys- tems	2
1.1.2	Approximate Causal Consistency Algorithm	4
1.1.3	Cost Optimized Replication Protocol	4
1.1.4	Causal+ Consistency for Posts/Comments in Social Net- working Platforms	5
1.2	Contributions	6
1.2.1	Causal Consistency Algorithms For Partially Replicated Sys- tems	6
1.2.2	Approximate Causal Consistency Algorithm	8
1.2.3	Cost Optimized Replication Protocol	9
1.2.4	Causal+ Consistency for Posts/Comments in Social Net- working Platforms	10
1.3	Thesis Outline	11
2	RELATED WORK	12
3	CAUSAL CONSISTENCY PROTOCOLS FOR PARTIALLY/FULLY REPLICATED SYSTEMS	18
3.1	Causally Consistent Memory	18
3.2	Underlying Distributed Communication System	20
3.3	Activation Predicate	23
3.3.1	The \rightarrow_{co} relation	23
3.3.2	Safety	26
3.3.3	Optimal Activation Predicate	26
3.4	Causal Consistency Algorithms	27
3.4.1	Full-Track Algorithm	27
3.4.1.1	Data Structure	28
3.4.1.2	Correctness and Optimality Outlines	29
3.4.1.3	Liveness	31
3.4.1.4	Optimality of the Activation Predicate	31
3.4.2	Opt-Track Algorithm	31
3.4.2.1	Data Structure	37
3.4.2.2	Correctness and Optimality Outlines	40
3.4.2.3	Liveness	41

TABLE OF CONTENTS (Continued)

<u>CHAPTER</u>		<u>PAGE</u>
	3.4.2.4 Optimality of the Activation Predicate	41
	3.4.2.5 Optimality of Log Space and Message Overhead Space	41
	3.4.3 Opt-Track-CRP: Adapting Opt-Track Algorithm to Fully-Replicated Systems	42
	3.4.3.1 Formula	44
	3.4.4 Complexity	45
	3.4.4.1 Full-Track Algorithm	48
	3.4.4.1.1 Message Count Complexity	48
	3.4.4.1.2 Message Space Complexity	48
	3.4.4.1.3 Time Complexity	49
	3.4.4.1.4 Space Complexity	49
	3.4.4.2 Opt-Track Algorithm	49
	3.4.4.2.1 Message Count Complexity	49
	3.4.4.2.2 Message Space Complexity	50
	3.4.4.2.3 Time Complexity	51
	3.4.4.2.4 Space Complexity	51
	3.4.4.3 Opt-Track-CRP Algorithm	52
	3.4.4.3.1 Message Count Complexity	52
	3.4.4.3.2 Message Space Complexity	52
	3.4.4.3.3 Time Complexity	52
	3.4.4.3.4 Space Complexity	53
	3.4.5 Experiments	53
	3.4.5.1 Partial Replication Protocols: Meta-data size	54
	3.4.5.1.1 Scalability as a Function of n	54
	3.4.5.1.2 Impact of Write Rate w_{rate}	55
	3.4.5.2 Full Replication Protocols: Meta-data size	59
	3.4.5.2.1 Scalability as a Function of n	60
	3.4.5.2.2 Impact of Write Rate w_{rate}	62
	3.4.5.3 Partial Replication vs. Full Replication: Message Count	64
	3.4.5.4 Partial Replication vs. Full Replication: Message Space Overhead	64
4	APPROXIMATE CAUSAL CONSISTENCY	69
	4.1 Basic Idea of Approx-Opt-Track	69
	4.2 Approx-Opt-Track	71
	4.3 Credit Instantiations	75
	4.4 Simulation System Model	77
	4.4.1 Process Model	77
	4.4.2 Simulation Parameters	78
	4.4.3 Process Execution	79
	4.4.4 Causal Consistency Verification	81
	4.5 Simulation Results	81

TABLE OF CONTENTS (Continued)

<u>CHAPTER</u>		<u>PAGE</u>
4.5.1	Violation Error Rate (R_e)	83
4.5.2	Average Message Meta-Data Size (m_{ave})	85
4.5.3	Message Meta-Data Size Saving Rate (R_s)	90
4.6	Simulation Evaluation	90
4.6.1	Impact of initial cr on R_e	91
4.6.2	Impact of w_{rate} on R_e	91
4.6.3	Impact of initial cr on m_{ave}	92
4.6.4	Impact of initial cr on R_s	93
4.6.5	Impact of replica factor rate r_f on cr_c	95
4.7	Discussions	96
5	A PROACTIVE, COST-AWARE, OPTIMIZED DATA REPLICATION STRATEGY IN GEO-DISTRIBUTED CLOUD DATA-STORES	99
5.1	System Cost Model	99
5.1.1	Adaptive Cloud Data Provider Architecture	99
5.1.2	ARIMA Configuration	105
5.1.3	Prediction Complexity	106
5.2	Cost Optimization Replica Placement (CORP)	107
5.2.1	CORP Algorithm	107
5.2.2	Cost Optimization Problem	115
5.2.3	CORP + cache.	120
5.3	Performance Evaluation	121
5.3.1	Experimental Setting	121
5.3.2	Results and Discussion	122
5.3.2.0.1	Cost Improvement	127
5.3.2.0.2	Accuracy Analysis	128
5.3.2.0.3	CORP+cache VS. CORP	129
5.3.2.0.4	CORP+cache evaluation	130
6	CADROP: COST OPTIMIZED CONVERGENT CAUSAL CONSISTENCY IN SOCIAL NETWORK SYSTEMS	133
6.1	Definitions and System Model	133
6.1.1	Causal Consistency	133
6.2	System Design	134
6.2.1	Convergent Conflict Handling	137
6.3	Algorithm: CaDRoP	139
6.3.1	The Client Layer	139
6.3.2	The Storage Layer	140
6.3.3	Dynamic Replication Model	143
6.4	Performance Evaluation	149
6.4.1	Results and Discussion	151

TABLE OF CONTENTS (Continued)

<u>CHAPTER</u>		<u>PAGE</u>
	6.4.1.1 CaS VS. CaS+cache	151
	6.4.1.2 CaS+cache VS. CaDRoP+cache	152
	6.4.1.2.1 CaDRoP+cache VS. CaDRoP	153
	6.4.1.2.2 CaDRoP+cache evaluation	153
	6.4.1.3 CoCaCo VS. CaDRoP+cache	154
7	CONCLUSIONS AND FUTURE WORK	162
	APPENDICES	165
	Appendix A	166
	Appendix B	168
	Appendix C	170
	CITED LITERATURE	174

LIST OF TABLES

<u>TABLE</u>		<u>PAGE</u>
I	Complexity measures of causal memory algorithms.	47
II	Average SM and RM space overhead for Full-Track and Opt-Track (KB)	57
III	Average SM space overhead for Opt-Track-CRP (byte)	61
IV	Total message count for partial replication (Opt-Track) VS. full replication (Opt-Track-CRP)	67
V	Total message overheads for full replication (Opt-Track-CRP) and partial replication (Opt-Track), when $n=40$, $p=12$, $w_{rate}=0.5$, in the worst case.	67
VI	Total message overheads for full replication (Opt-Track-CRP) and partial replication (Opt-Track), when $n=40$, $p=12$, $w_{rate}=0.5$, in practice.	68
VII	Message meta-data structures in partial replication protocols	68
VIII	Critical initial credits for the replica factor rate = 0.3.	83
IX	Critical average message meta-data size m_{ave} (KB).	85
X	Message meta-data size saving rates R_s when R_e is close to or equal to zero.	89
XI	Critical initial credits for the replica factor rate of 0.5.	89
XII	Critical initial credits for the replica factor rate of 0.2.	95
XIII	Impacts of failures/partitions compared to the “no failure” case. . . .	96
XIV	Definition of symbols and parameters used in the model.	119
XV	Cost improvement rates of different replication modes with respect to the standalone cache mode for different Put rates.	128

LIST OF TABLES (Continued)

<u>TABLE</u>		<u>PAGE</u>
XVI	Access number prediction accuracy by various error metrics.	129
XVII	Δ_{saving} : The cost improvement results for different <i>Put</i> rates show that caching has taken an important step to improve the total system costs.	130
XVIII	Δ_{inc} : The performance evaluation of CORP+cache compared to OPT+cache.	131
XIX	$\Delta_{inc'}$: The performance evaluation of CORP compared to OPT in steady states.	131
XX	Definition of symbols and parameters used in the model.	138
XXI	Cost improvement rates in different <i>Put</i> rates and RF values. . . .	158
XXII	Δ_{saving} : The cost improvement results for different <i>Put</i> rates show that caching has taken an important step to improve the total system costs. Δ_{inc} : The performance evaluation of CaDRoP+cache compared to OPT+cache. $\Delta_{inc'}$: The performance evaluation of CaDRoP compared to OPT in steady states.	159
XXIII	The price cost comparisons between CoCaCo and CaDRoP+cache in different <i>Put</i> rates and RF models.	161

LIST OF FIGURES

<u>FIGURE</u>		<u>PAGE</u>
1	False causality.	25
2	Illustration of the two conditions for destination information to be redundant.	32
3	Illustration of why it is important to keep a record even if its destination list becomes empty. For clarity, the <i>apply</i> and <i>return</i> events after each message delivery are omitted.	37
4	In fully replicated systems, the local log will be reset after each write operation. Also, when a write operation is applied, only the write operation itself needs to be remembered. For clarity, the <i>apply</i> events are omitted in this figure.	43
5	Total message meta-data space overhead as a function of n and w_{rate} in partial replication protocols.	55
6	Average message meta-data space overhead as a function of n with lower w_{rate} (0.2) in partial replication protocols.	56
7	Average message meta-data space overhead as a function of n with medium w_{rate} (0.5) in partial replication protocols.	57
8	Average message meta-data space overhead as a function of n with higher w_{rate} (0.8) in partial replication protocols.	58
9	Total message meta-data space overhead as a function of n and w_{rate} in full replication protocols.	60
10	Average message meta-data space overhead as a function of n with lower w_{rate} (0.2) in full replication protocols.	61
11	Average message meta-data space overhead as a function of n with medium w_{rate} (0.5) in full replication protocols.	62
12	Average message meta-data space overhead as a function of n with higher w_{rate} (0.8) in full replication protocols.	63

LIST OF FIGURES (Continued)

<u>FIGURE</u>		<u>PAGE</u>
13	Illustration of causal consistency violation if credits are exhausted. . . .	71
14	Illustration of meta-data reduction when credits are exhausted.	71
15	The Violation Error Rate for $w_{rate} = 0.2$	84
16	The Violation Error Rate for $w_{rate} = 0.5$	84
17	The Violation Error Rate for $w_{rate} = 0.8$	85
18	The Average Meta-Data Size (m_{ave}) for $w_{rate} = 0.2$	86
19	The Average Meta-Data Size (m_{ave}) for $w_{rate} = 0.5$	86
20	The Average Meta-Data Size (m_{ave}) for $w_{rate} = 0.8$	87
21	The Meta-Data Size Saving Rate (R_s) for $w_{rate} = 0.2$	87
22	The Meta-Data Size Saving Rate (R_s) for $w_{rate} = 0.5$	88
23	The Meta-Data Size Saving Rate (R_s) for $w_{rate} = 0.8$	88
24	Examples of failures (a) partition (b) message loss.	96
25	System architecture.	100
26	Architecture for adaptive cloud data provider.	103
27	Diagram for ARIMA workload-based predictor.	105
28	The Transaction Cost	124
29	The Network Transmission Cost	125
30	The Storage Space Cost	126
31	The total system cost.	127
32	The system architecture.	135
33	An example with the convergence property.	156

LIST OF FIGURES (Continued)

<u>FIGURE</u>		<u>PAGE</u>
34	The Transaction Cost	156
35	The Network Transmission Cost	158
36	The Storage Space Cost	159
37	The Total System Cost	160

LIST OF ABBREVIATIONS

Opt	Optimal
IC	Incomplete Causality
ARIMA	Autoregressive Integrated Moving Average
CORP	Cost Optimized Replication Protocol
<i>cr</i>	<i>credits</i>
CaDRoP	Causal consistency in Dynamic Replicated Optimized Protocol.

SUMMARY

In geo-replicated cloud computing systems, it is essential to maintain high data availability, low latency, and fault-tolerance. In order to meet these requirements, replication mechanisms are necessary. Therefore, the cost of replication is one of the most important factors in designing such a system. Theoretically, the more the number of replicas for a data file, the higher the data access availability should be. But, the cost of replication becomes greater at the same time. It is a challenge to develop a solution to achieve the optimal trade-off between the cost of replication and data access availability.

Causal consistency in geo-replicated systems is an interesting consistency model. Most existing works with causal consistency have focused on the full replication for the data. This greatly simplifies the design of the algorithms to implement causal consistency. This is because full replication protocols do not need to track transitive causal dependencies between each pair of processes. However, partial replication protocols have several advantages, such as each write/update operation leads to fewer messages being multicast and smaller storage overheads. Although partial replication can avoid unnecessary network traffic and prevent decreased responsiveness, it poses big challenges to realize partial replication against full replication. This is primarily due to the higher complexity of tracking causal consistency, resulting in additional communication cost and larger dependency meta-data overheads. In this thesis we develop cost-effective protocols for high performance causally consistent geo-replicated key-value data stores in different settings.

SUMMARY (Continued)

First, we propose an optimal partial replication protocol-Opt-Track-for causal consistency in an one-level framework. We also give a special case algorithm-Opt-Track-CRP-for causal consistency in the full-replication case. This work provides the first evidence that explores causal consistency for partially replicated distributed systems. Opt-Track shows better performance on network capacity than full replication.

Next, 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 show that in return for the potential benefit, Approx-Opt-Track provides almost the same guarantees as causal consistency, at a smaller cost.

Then, we address the problem of determining suitable replica placement on-the-fly to increase the availability of data resources and maximize the system utilization. Specifically, it is essential to load the appropriate number of replicas for different data resources at each geo-data cluster in a particular time interval. We propose Cost Optimization Replica Placement (CORP) algorithms to enable state-of-art proactive provisioning of data resources based on an one-step look-ahead workload behavior pattern forecast over the distributed data storage infrastructure using statistical techniques.

Finally, we propose a causal+ consistency protocol, CaDRoP(+cache), to support dynamic replication, and ensure the convergence property for all comments following a post and the causal ordering between posts with explicit causality. We evaluate CaDRoP(+cache) protocol with realistic workloads by different PUT rates in terms of the practical price of Amazon AWS.

SUMMARY (Continued)

The results show that CaDRoP(+cache) incurs much lower cost than the statically replicated data store in another causal+ algorithm. We further evaluate CaDRoP by comparing it with a clairvoyant optimal replication solution.

CHAPTER 1

INTRODUCTION

Reliable distributed shared services are growing in popularity since they can improve the availability and direct services closer to end customers so as to lower access time and drive up customer engagement. Data replication is a critical infrastructure component commonly used in such distributed shared repositories. By replicating data resources across different geographic sites, this model can protect the data to ensure availability in the event of a disaster resulting from correlated failures (4). Furthermore, an effective replication protocol must not only provide efficient access to huge amounts of data, but also reduce access latency in large-scale cloud and geo-replicated distributed systems. With data replication, consistency of data in the face of concurrent reads and updates becomes an essential requirement. Recently, consistency models have widely been utilized in the context of cloud computing with data centers and geo-replicated storages, with product designs from industry, e.g., Google, Amazon, Microsoft, LinkedIn, and Facebook.

There exists a spectrum of consistency models in distributed shared systems (5): linearizability (the strongest), sequential consistency, causal consistency, pipelined RAM, slow memory, and eventual consistency (the weakest). These consistency models represent a fundamental trade-off between cost and convenient semantics for the application programmer. It is well understood between having guaranteed low access latency and making sure that every client sees a single ordering of all operations in the system (strong consistency) (6). This trade-off can

be pretty formalized in the famous CAP theorem. The CAP Theorem introduced by Brewer (7) asserts that for a replicated, distributed data store, it is possible to provide at most two of the three features: Consistency of replicas (C), Availability of Writes (A), and Partition tolerance (P). This theorem supports the major reason behind the increasing prevalence of eventual consistency, popularized by Amazon’s Dynamo (8). This consistency model guarantees the availability to return local results quickly if other remote replicas are unavailable. It states that eventually, all copies of each data item converge to the same value. However, what eventual consistency sacrifices is consistency: replicas hosted at different datacenters may have different results with different order.

What can provide something better than eventual consistency without sacrificing low latency or availability is causal consistency. 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 (9). It specifies that two events that are causally related must appear in the same order. Consider, for example, a user U 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 could not see this photo. It requires extra programming efforts to prevent the inconsistent scenario at the application level.

1.1 Problem Motivation

1.1.1 Causal Consistency Algorithms For Partially Replicated Systems

There has been much research implemented and conducted in the field of causal consistency in the context of large geo-replicated cloud storage in recent years. However, most of studies

assume Complete Replication and Propagation (CRP) based protocols. These protocols assume full replication across all the sites and do not consider the case of partial replication. This is primarily because partial replication makes it difficult to implement causal consistency, due to having to track transitive causal dependencies between each pair of processes. Partial replication protocols have several advantages. Specially, each write operation results in fewer messages being multicast for savings in bandwidth, and the protocols use lesser storage for fewer system-wide replicas for savings in system resources.

A number of algorithms for achieving causal message ordering have been previously proposed in distributed shared memory (DSM) systems (10; 11; 12). Compared to the causal consistency algorithms, these algorithms are for message-passing systems where application processes communicate with each other via sending and receiving messages. None of these causal message ordering algorithms assume that messages get broadcast each time application processes communicate with each other. This is similar to partially replicated distributed shared memory systems, where an individual application process writing a variable does not write to all sites in the system. In both cases, the changes in one application process do not get propagated to the entire system. These algorithms provide a good starting point for the design of our algorithms. Furthermore, there is no study analytically comparing partial replication with full replication in message space overheads and message counts for causal consistency.

Our goal is to devise an optimal protocol for achieving causal consistency in such systems where the data is only partially replicated to achieve optimality in terms of the activation predicate and optimizing the space overheads of dependency meta-data.

1.1.2 Approximate Causal Consistency Algorithm

In the real world, the characteristics of causal consistency and partial replication are highly useful for modern social networking applications. Data replication costs are an important factor in overall storage system performance. Partial replication can effectively reduce the replication costs, especially in multimedia-oriented social networks (e.g., Instagram). Causal consistency improves social user experience since actions appear to everyone in the correct order. For large-sized files being written, the net meta-data overhead is negligible 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 text posts in Facebook or Twitter, the size of the meta-data can be a problem.

The second problem is to consider the text type of data in social networks. We aim to further reduce the size of the meta-data for maintaining causal consistency in partially replicated systems, based on the solution to the previous problem.

1.1.3 Cost Optimized Replication Protocol

The delivery of cloud computing resources as a utility provides users with flexible services in a transparent manner, with cheap costs and more customized operations. One may pay for such demand resources to cloud data providers (CDPs) as per their usage like a utility. As outsourcing continues to rise in popularity, the widespread adoption of cloud-based data services raises the challenges of provisioning appropriate replicas of data resources situated at different geographical locations. As the number of replicas of a target data item increases, the overheads of loading and maintaining these replicas become higher and expensive. Thus, a fine balance

of the advantages and overheads of replication is needed. However, static replication of data resources in dynamic environments having time-varying workloads is ineffective for optimizing system utilization. For example, the operating behaviors of social media applications vary based on different time intervals (e.g., Monday vs. Sunday). Adaptive data replication strategies could potentially resolve the above issue by enabling dynamic provisioning of data resources to applications based upon the patterns of workload traces. The challenge of dynamic replica provisioning is the determination of the following three aspects. The first is to select which data items need to be replicated. The second is to select how many replicas should be loaded in the whole cloud system. The third is to decide the locations where the replicas should be placed.

The third problem is to design a proactive provisioning replication scheme across multiple cloud data providers (CDP). CDP can dynamically deploy required data replicas in suitable geo-locations for serving the predicted requests in the near future.

1.1.4 Causal+ Consistency for Posts/Comments in Social Networking Platforms

–

Causal consistency preserves intuitive causal ascription. This feature is crucial in social networks (e.g., privacy policies). It improves user experience, because, with it, events appear to each user in the correct order. However, the ordinary causal consistency does not support the convergence property on concurrent updating operations. Thus, updating conflicts may result in replicas having different values for the same key in key-value store systems. Casual+ consistency requires that data replicas converge to the same state under concurrent updates

from different clients. Additionally, dynamic replication and placement can enhance the system availability in time-varying environments. A number of causal+ consistency protocols have been proposed for partially replicated key-value store systems. They are all based on static replication strategies. None of them can simultaneously satisfy the following requirements in social media systems. 1) All the reply comments (values) under a post (a key) follow causal consistency. 2) Each individual post may be replicated to other different data storage nodes. When concurrent updates to a post are made in different replicas, the comments of each replica-post (with the same key) can be ordered in the same sequence. 3) Each post would be dynamically replicated to geographically-diverse data store nodes in different time periods according to user requirements.

The fourth problem is to devise a cost-optimized protocol that ensures causal+ consistency within a datacenter or across datacenters for posts and comments in a partially geo-replicated storage and satisfies the above three requirements.

1.2 Contributions

The contributions in this work are as follows.

1.2.1 Causal Consistency Algorithms For Partially Replicated Systems

We devise causal consistency protocols in partially replicated store systems to make the following contributions:

1. Full-Track protocol is optimal in the sense defined by Baldoni et al. (13), viz., It can update the local copy as soon as possible while respecting causal consistency. This reduces the false causality in the system.

2. Full-Track can be made further optimal in terms of the size of the local logs maintained and the amount of control information piggybacked on the update messages, by achieving minimality. The resulting algorithm which optimally minimizes the size of meta-data is Algorithm Opt-Track.
3. As a special case of protocol Opt-Track, we present Opt-Track-CRP, that is optimal in a fully replicated distributed shared memory system. This algorithm is optimal not only in the sense of Baldoni et al. but also in the amount of control information used in local logs and on update messages, which is considerably less than for algorithm Opt-Track, making it highly scalable. The algorithm is significantly more efficient than the Baldoni et al. protocol *optP* (13) for the complete replication case.

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. We present that Opt-Track-CRP also significantly outperforms *OptP* in scalability and network capacity utilization. we also explore the trade-off between partial replication and full replication analytically. We show the advantage of partial replication over full replication. Opt-Track is applicable to large-scale DSM systems, and in particular to those accommodating replications of medium or large-sized data files (> 100 KB). An example of a real world network which can benefit from our results is the multimedia object oriented social network ‘Instagram’ which is a photo-sharing social community where the average file size is 2 MB.

1.2.2 Approximate Causal Consistency Algorithm

To solve the problem of reducing the dependency meta-data overheads with small payload data sizes, we make the following 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 protocol, to give a protocol Approx-Opt-Track that can fine-tune the amount of causal consistency by trading off the size of meta-data overhead.
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, Approx-Opt-Track is seen to show significant gains over Opt-Track. This implies 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.

1.2.3 Cost Optimized Replication Protocol

To solve the problem of the replica placement decision strategy, we make the following contributions:

1. We propose Cost Optimization Replica Placement (CORP) algorithm and design a proactive provisioning replication scheme across multiple CDPs. According to current data resource allocation and historical changes in workload patterns, our replication framework, composed of CDPs, is designed to employ the autoregressive integrated moving average (ARIMA) model to concretely predict how many data access requests are made in the near future. CDP can dynamically deploy required data replicas in suitable geo-locations for serving the predicted requests.
2. Since caching has potential for performance benefits, we also extend CORP as CORP (+cache) for different data-intensive workloads.
3. We further propose the optimal placement solution to evaluate CORP (+cache) in steady states.

We conduct an evaluation of cost-effectiveness of our CORP algorithms via trace-driven CloudSim simulator toolkit and realistic workload traces from Twitter. Results show that

CORP with cache mechanism can highly reduce the total system cost in comparison to the stand-alone caching strategy and static partial replication.

1.2.4 Causal+ Consistency for Posts/Comments in Social Networking Platforms

To solve the problem of enforcing causal+ consistency for the comments under each post and between posts on social network applications, we make the following contributions:

1. We propose a new cost-optimized protocol that ensures causal+ for all replying comments corresponding to an object (a post) with a unique key or for different objects with explicit happens-before relationships in social applications by a key-values store system.
2. Because of the potential of dynamic replication strategies for system cost savings, we adapt the consistency protocol CaDRoP for our proposed replication algorithm CORP in optimizing the total system costs.
3. Due to the potential of caching in performance benefits, we also extend CaDRoP as CaDRoP+cache to reduce network transmission costs.

We conduct an evaluation of cost-effectiveness of CaDRoP(+cache) algorithms via trace-driven CloudSim simulator toolkit and realistic workload traces from Twitter in terms of the prices set on Amazon Web Service (AWS) as of 2019. Results show that the total system cost can be highly reduced by CaDRoP+cache in a dynamic replication strategy (3) in comparison to the same protocol and CoCaCo(14) in different static replication models. We further evaluate CaDRoP by comparing it with a clairvoyant optimal replication solution. The findings

indicate that with cache, CaDRoP only incurs around 8% extra cost on average. Without cache, CaDRoP brings less than 5% extra cost in steady states.

1.3 Thesis Outline

Chapter 1 introduces the background and presents the overall problem statements and hypotheses of this research, and outlines the research contributions. Chapter 2 provides the related work for the thesis. Chapter 3 discusses the problem of implementing causal consistency protocols in partially replicated distributed shared memory systems and presents the algorithms to do so. Chapter 4 discusses the problem of the dependency meta-data overheads with small payload data sizes and presents the algorithms to do so. Chapter 5 discusses the problem of the replica placement decision strategy and give the corresponding algorithm. Chapter 6 discusses the problem of implementing causal+ consistency protocols in geo-replicated data stores for social network applications. The conclusion and future directions of this work are given in Chapter 7.

Portions of Chapter 3 have been previously published in Elsevier Journal of Future Generation Computer Systems (1). Portions of Chapter 4 have been previously published in IEEE Transactions on Parallel and Distributed Systems (2). Portions of Chapter 5 have been previously published in the proceedings of IEEE/ACM 12th International Conference on Utility and Cloud Computing (3). Portions of Chapter 6 have been submitted under reviewing.

CHAPTER 2

RELATED WORK

Causal consistency in distributed shared memory systems was proposed by Ahamad et al. (15) and studied by Petersen et al. (16) and Belaramani et al. (17). Later, Baldoni et al. (13) gave an improved broadcast based protocol *OptP* to implement causal memory. Their implementation is optimal in the sense that the protocol can update the local copy as soon as possible, while respecting causal consistency. Specifically, additional delays due to the inability of Lamport’s “happened before” relation (18) to map in a one-to-one way, cause-effect relations at the application level into relations at the implementation level (a phenomenon called *false causality*) are eliminated. False causality was identified by Lamport (18). 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 by Mahajan et al. (19).

Lazy replication (20) is a client-server framework to provide causal consistency using vector clocks, where the size of the vector is equal to the number of replicas. A client can issue updates and queries to any replica, and replicas exchange gossip messages to keep their data up-to-date. Causal+ consistency (CC+) has gained interest as a highly attractive consistency model. The vast majority of existing CC+ protocols assume full replication. They are outlined next. COPS (6) implements a causally consistent key-value store system. It computes a list of dependencies whenever an update occurs, and the update operation is not performed until updates in the

dependencies are applied. The transitivity rule of the causality relationship is used to prune the size of the dependency list. Eiger (21), an improvement over COPS, provides scalable causal consistency for the complex column-family data model, as well as non-blocking algorithms for read-only and write-only transactions. The Orbe (22) key-value storage system provides two different protocols to provide causal consistency – the DM protocol uses two-dimensional matrices to track the dependencies, and the DM-Clock protocol uses loosely synchronized physical clocks to support read-only transactions. The GentleRain (23) causally consistent key-value store uses a periodic aggregation protocol to determine whether updates can be made visible in accordance with causal consistency. Rather than using explicit dependency check messages, it tracks causal consistency by attaching to updates, scalar timestamps derived from loosely synchronized physical clocks. Cure (24) uses a stabilization approach for making update operations visible while respecting causal consistency. ChainReaction (25) achieves causal consistency on top of chain replication. In order to track the state of every access at the client side, it exploits a global sequencer service at each datacenter to order update operations and read-only transactions. However, the sequencer service is a potential performance bottleneck, because it increases the latency of update operations by one round-trip network delay within the datacenter. Bolt-on (26) provides causal consistency on top of eventually consistent data stores. A shim-layer is inserted between the data store layer and the application layer to ensure causal ordering. Bolt-on relies on the application semantics to track explicit causality relationships but requires the application developer to handle causal relationships among application operations. Other CC+ protocols include OCCULT (27), OCC (28), Rots (29), Spartan (30), CSNOW (31), and

Eunomia (32). Those protocols support one-shot read-only transactions. OCCULT(27) can block reads on a data node when waiting for a snapshot to be installed.

A few CC+ protocols support partial replication, such as Saturn (33), C³ (34), Karma (35), CoCaCo (14), LazyP (20), and the one by Xiang et al. (36). These CC+ protocols realize single-object read and write operations. SwiftSwiftcloud (37) provides efficient reads and writes using an occasionally stale but causally consistent client-side local cache mechanism. The size of the meta-data is proportional to the number of data centers used to store data. Paris (38) provides fresher data to the clients and tolerate some degree of data staleness. However, those protocols implement CC+ for the single value associated with the requesting object. Crain et al. (39) outlined a causally consistent protocol for geo-distributed partial replication with dependency vectors. The main idea used is that the sender, instead of the receiver, checks the dependencies when propagating updates among data centers. However, it does not support the convergence property and still considers imprecise representation of dependencies, which can result in false dependencies.

Replication mechanisms have been particularly influential in distributed data storage systems. Cost optimization of cloud storage services has seen growing importance of pricing differences. In this section, we investigate the related work in the above two broad areas. The challenge of dynamic resource management and allocation in distributed systems and cloud environments has been dealt with via several *reactive* approaches. Evaluated from different kinds of users' access patterns, six replication strategies are proposed for hierarchically distributed data grids in an initial work on dynamic data replication (40). The users' dynamic and distributed

nature has been used in (41) to design suitable replica placement strategies in the grid environment. A dynamic data replication mechanism called Latest Access Largest Weight (LALW) is proposed in (42). LALW calculates the access frequency of each file requested divided by average access frequency of all files. Each data file is assigned different weights at different times in its lifetime. The most recently requested data files are given higher weights. In this way, the overload of the network can be reduced effectively. A dynamic distributed cloud data replication algorithm CDRM is proposed in (43). CDRM is a cost-effective framework for replication designed on the HDFS platform in a cloud storage system. Based on node capacity and workload changes to calculate the popularity of a data file, CDRM can dynamically determine the replica placement in the cloud environment. Dynamic data replication strategy (D2RS) in hierarchical cloud environments to improve system availability is proposed in (44). Combined with a checkpoint strategy, the above algorithm is extended as a new algorithm called Dynamic Adaptive Fault-tolerance (DAFT) in (45). It can dynamically provision and deliver computing resources in a transparent manner to achieve higher availability, performance, and reliability. Based on D2RS, the concept of knapsack has been used in (46) to optimize the cost of data replication between data centers without impacting the data availability. Because of pricing differences among different resources across cloud data stores, a lightweight heuristic solution to minimize monetary cost of the application in hot-spot or cold-spot objects is proposed in (47). It utilizes dynamic and linear programming techniques to identify object replica location, migration time, and remote access request redirection. Based on the scale of applications, and dynamic workloads, the models proposed in (48) optimize request response time during normal

operations to meet SLA (scalability, latency, and availability) requirements. A hierarchical data replication strategy (HDRS) proposed in (49) can dynamically store replicas, based on the access load and labeling technique, to offer high data availability and low bandwidth consumption, while satisfying QoS requirements and storage capacity constraints. A two-way replication strategy (TWR) with P2P-like features, where a multi-tier sibling-tree architecture is used, is proposed in (50). TWR intends to place the replicas of more popular data files near the users, and that of less popular data files away from the users. The data files with higher-than-average access frequency are replicated at the parent node of the client node generating the most requests, and the data files with lower access frequency are migrated to the grandparent node. A distributed traffic-based replication strategy RFH (resilient, fault-tolerant and high efficient) for high availability of data stored in the cloud is proposed in (51). Based on the access traffic load of data nodes, the replicas of a data file are placed at the place facing maximum traffic to access this data file and the least blocking probability to meet end-user requirements. A list-based replication strategy in data grid systems that introduces the concept of priority list to specify the forwarding policies is proposed in (52). A dynamic reliability-driven instantaneous fault-tolerant scheduling scheme (DRFACS) with active replication mechanism to handle enormous resource failures is proposed in (53). DRFACS intends to increase reliability by dynamically scheduling non-periodic and non-preemptive real-time tasks to enhance the QoS throughout the scheduling process. Similarly, in (54), a dynamic self-organized, fault-tolerant and scalable replication strategy is proposed for the resource allocation optimization decision in order to reduce the query processing overheads and to meet the availability for different query rates

and storage space requirements in a cost-effective manner. The framework of Differentiated Replication (DiR) to select suitable replication strategies on the basis of users' requirements and system resource utilization is used in (55).

The weakness of reactive replication approaches is their inability to anticipate the unexpected changes in workload. Since the workload changes in cloud environments follow patterns that may vary time to time, *proactive* methods can overcome the above deficiency by pre-deciding the correct amount of resources before the expected increase or decrease of demand. An approach to the problem of workload prediction based on identifying similar past occurrences of the current short-term workload history in public clouds is proposed in (56). An event-aware strategy to more effectively predict workload bursts by exploiting prior knowledge associated with scheduled events is proposed in (57). Artificial neural networks (ANN) and linear regression were used to develop prediction-based resource measurement and provisioning strategies in (58). A cloud workload prediction module for SaaS providers based on the ARIMA model is proposed in (59). This model can meet the QoS with a cost-effective amount of resources by estimating the future requirements.

CHAPTER 3

CAUSAL CONSISTENCY PROTOCOLS FOR PARTIALLY/FULLY REPLICATED SYSTEMS

This chapter is based on our previous publications (60; 61; 1). In this chapter, we discuss the problem of implementing causal consistency in partially replicated data stores, presenting the algorithms, and illustrating the experimental results. The rest of this chapter is organized as follows. Section 3.1 presents the fundamental definition of our causally consistent memory model. Section 3.2 presents the underlying communication system. Section 3.3 describes the activation predicate determination. Section 3.4 shows three causal consistency algorithms we proposed for partial replication data stores.

3.1 Causally Consistent Memory

The system model is based on that proposed by Ahamad et al. (15) and Baldoni et al. (13). We consider a system which consists of n application processes ap_1, ap_2, \dots, ap_n interacting through a shared memory \mathcal{Q} composed of q variables x_1, x_2, \dots, x_q . Each application process ap_i can perform either a *read* or a *write* operation on any of the q variables. A *read* operation performed by ap_i on variable x_j which returns value v is denoted as $r_i(x_j)v$. Similarly, a *write* operation performed by ap_i on variable x_j which writes the value u is denoted as $w_i(x_j)u$. Each variable has an initial value \perp .

By performing a series of *read* and *write* operations, an application process ap_i generates a local history h_i . If a local operation o_1 precedes another operation o_2 , we say o_1 precedes o_2 under *program order*, denoted as $o_1 \prec_{po} o_2$. The set of local histories h_i from all n application processes form the global history H . Operations performed at distinct processes can also be related using the *read-from order*, denoted as \prec_{ro} . Two operations o_1 and o_2 from distinct processes ap_i and ap_j respectively have the relationship $o_1 \prec_{ro} o_2$ if there are variable x and value v such that $o_1 = w(x)v$ and $o_2 = r(x)v$, meaning that *read* operation o_2 retrieves the value written by the *write* operation o_1 . It is shown in (15) that

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

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

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

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

Given a history H , S is a *serialization* of H if S is a sequence containing exactly the operations of H such that each read operation of a variable x returns the value written by the most recent precedent write on x in S . A serialization respects a given order if, for any two operations o_1 and o_2 in S , o_1 precedes o_2 in that order implies that o_1 precedes o_2 in S . Let H_{i+w} be the history containing all the operations in h_i and all write operations of H .

Definition 1. (*Causally consistent history*) *A history is causally consistent if for each application process ap_i , there is a serialization S_i of H_{i+w} that respects the causality order \prec_{co} .*

Definition 2. (*Causal memory*) *A memory is causal if it admits only causally consistent histories.*

3.2 Underlying Distributed Communication System

The DSM abstraction and its causal consistency model is implemented by a memory consistency system (MCS) on top of the underlying distributed message passing system which also consists of n sites connected by FIFO channels. The distributed system is asynchronous. We assume that all messaging primitives are reliable. Message transfer delay is arbitrary but finite, and there is no bound on the relative process speeds. Each site s_i hosts an application process ap_i . The local MCS process at s_i is denoted mp_i .

Since we assume a partially replicated system, each site holds only a subset of variables $x_h \in \mathcal{Q}$. For application process ap_i , we denote the subset of variables kept on the site s_i as X_i . We assume the replication factor of the DSM system is p and the variables are evenly replicated on all the sites. This assumption is justified from a statistical viewpoint. It follows that the average size of X_i is $\frac{pq}{n}$.

If a variable x_h is not locally replicated, then a read operation fetches the value from one of the replica sites of x_h . The replica site could be chosen randomly from one of the sites that replicate x_h . (Alternatively, a policy decision could be made to fetch the variable from the closest replica.) For this, we assume a static system and that complete knowledge about the partial replication scheme is known to all systems/replicas. Thus, for both a read and a write operation on a variable, each site is assumed to know the replica set of that variable.

To facilitate the read and write operations in the DSM abstraction, the underlying message passing system provides several primitives to enable the reliable communication between different sites. For the write operation, each time an application process ap_i performs $w(x_1)v$, the local MCS process invokes the **Multicast**(m) primitive to deliver the message m containing $w(x_1)v$ to all sites that replicate the variable x_1 . For the read operation, there is a possibility that an application process ap_i performing read operation $r(x_2)u$ needs to read x_2 's value from a remote site since x_2 is not locally replicated. In such a case, the local MCS process invokes the **RemoteFetch**(m) primitive to deliver the message m containing $r(x_2)u$ to a random site replicating x_2 to fetch its value u . This is a synchronous primitive, meaning that it will block

until returning the variable's value. If the variable to be read is locally replicated, then the application process is simply returned the local value.

The read and write operations performed by the application processes cause *events* to be generated in the underlying message passing system. More specifically, each MCS process mp_i generates a set of events E_i . $E = \langle E_i, \dots, E_n \rangle$ is the global set of events, ordered by Lamport's "happened before" relation \rightarrow (18). The distributed computation \hat{E} is the partial order induced on E by \rightarrow . The set of messages in \hat{E} is denoted $M_{\hat{E}}$.

The following types of events are generated at each site:

- *Send event.* The invocation of **Multicast**(m) primitive by MCS process mp_i generates event $send_i(m)$.
- *Fetch event.* The invocation of **RemoteFetch**(m) primitive by MCS process mp_i generates event $fetch_i(f(x))$. Here, $f(x)$ denotes the variable being fetched.
- *Message receipt event.* The receipt of a message m at site s_i generates event $receipt_i(m)$.

The message m can correspond to either a $send_j(m)$ event or a $fetch_j(f(x))$ event.

- *Apply event.* When applying the value written by the operation $w_j(x_h)v$ to variable x_h 's local replica at site s_i , an event $apply_i(w_j(x_h)v)$ is generated.
- *Remote return event.* After the occurrence of event $receipt_i(m)$ corresponding to the remote read operation $r_j(x_h)u$ performed by ap_j , an event $remote_return_i(r_j(x_h)u)$ is generated which transmits x_h 's value u to site s_j .

- *Return event.* Event $return_i(x_h, v)$ corresponds to the return of x_h 's value v either fetched remotely through a previous $fetch_i(f(x))$ event or read from the local replica.

To implement the causal memory in the DSM abstraction, each time an update message m corresponding to a write operation $w_j(x_h)v$ is received at site s_i , a new thread is spawned to check when to locally apply the update. The condition that the update is ready to be applied locally is called, as in (13), the activation predicate. This predicate, $A(m_{w_j(x_h)v}, e)$, is initially set to *false* and becomes *true* only when the update $m_{w_j(x_h)v}$ can be applied after the occurrence of local event e . The thread handling the local application of the update will be blocked until the activation predicate becomes *true*, at which time the thread writes value v to variable x_h 's local replica. This will generate the $apply_i(w_j(x_h)v)$ event locally. Thus, the key to implement the causal memory is the activation predicate.

3.3 Activation Predicate

3.3.1 The \rightarrow_{co} relation

To formulate the activation predicate, Baldoni et al. (13) defined a new relation, \rightarrow_{co} , on *send events* in \hat{E} generated in the underlying message passing system. We modify their definition by adding condition (3) to accommodate the partial replication scenario.

Definition 3. (\rightarrow_{co} on send events) Let $w(x)a$ and $w(y)b$ be two write operations in O_H . Then, for their corresponding send events in the underlying message passing system, $send_i(m_{w(x)a}) \rightarrow_{co} send_j(m_{w(y)b})$ iff one of the following conditions holds:

1. $i = j$ and $send_i(m_{w(x)a})$ locally precedes $send_j(m_{w(y)b})$

2. $i \neq j$ and $\text{return}_j(x, a)$ locally precedes $\text{send}_j(m_{w(y)b})$
3. $i \neq j$ and $\text{apply}_i(w(x)a)$ locally precedes $\text{remote_return}_l(r_j(x)a)$ which precedes (as per Lamport's \rightarrow relation) $\text{return}_j(x, a)$ which locally precedes $\text{send}_j(m_{w(y)b})$
4. $\exists \text{send}_k(m_{w(z)c})$, such that $\text{send}_i(m_{w(x)a}) \rightarrow_{co} \text{send}_k(m_{w(z)c}) \rightarrow_{co} \text{send}_j(m_{w(y)b})$

Notice that the relation defined by \rightarrow_{co} is actually a subset of Lamport's "happened before" relation (18), denoted by \rightarrow . If two send events are related by \rightarrow_{co} , then they are also related by \rightarrow . However, the other way is not necessarily true. Even though $\text{send}_i(m_{w(x)a}) \rightarrow \text{send}_j(m_{w(y)b})$, if there is no return event that occurred and $i \neq j$, these two send events are concurrent under the \rightarrow_{co} relation. The difference between these two relations is essential under the context of causal memory. The \rightarrow_{co} relation better represents the causality order in the DSM abstraction as it prunes the "false causality" introduced in the underlying message passing system, where message receipt events may causally relate two send events while their corresponding write operations in the shared memory abstraction are concurrent under the \prec_{co} relation. In simple terms, false causality arises under the \rightarrow relation when a remotely generated value has arrived but not been read, before a new value is generated locally. This is illustrated in Figure 1. Consider the computation in a partial replication system. Assume that there is no $\text{read}_2(x_2)$ operation and no corresponding $\text{return}_2(x_2)$ event. When s_3 receives $m(w(x_3)v_3)$ at time t_{31} , by using the \rightarrow relation, the apply event cannot be applied until t_{33} . This is because there is another message $m(w(x_1)v_1)$, which has not yet been received at s_3 such that $\text{send}_1(m(w(x_1)v_1)) \rightarrow \text{send}_2(m(w(x_3)v_3))$. However, this is not optimal because $\text{send}_1(m(w(x_1)v_1)) \not\rightarrow_{co} \text{send}_2(m(w(x_3)v_3))$. At t_{31} , $A_{opt}(m(w(x_3)v_3))$ will become true accord-

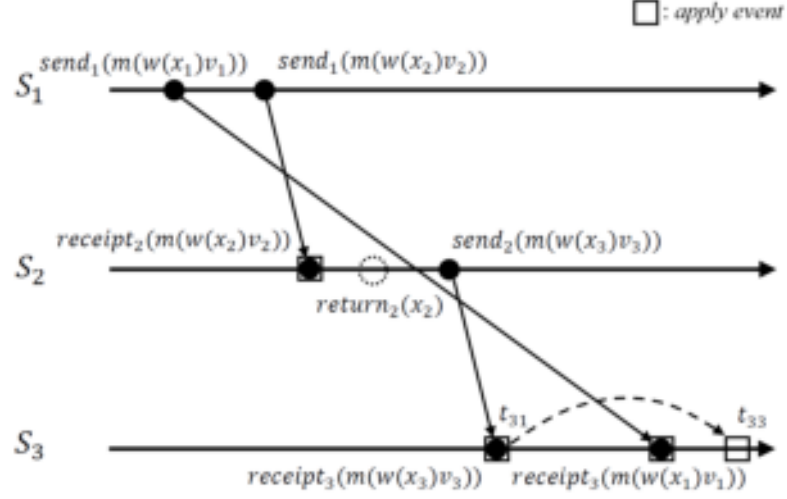


Figure 1. False causality.

ing to the \rightarrow_{co} relation (unless the $return_2(x_2)$ event corresponding to a $read_2(x_2)$ operation exists). Apply event $apply(m(w(x_3)v_3))$ can be immediately applied without any delay. Using the \rightarrow relation introduces this delay phenomenon, called false causality. This false causality is removed by using the \rightarrow_{co} relation.

Actually, it is easy to show the following property, along the lines of (13).

Property 1. $send_i(m_{w(x)a}) \rightarrow_{co} send_j(m_{w(y)b}) \Leftrightarrow w(x)a \prec_{co} w(y)b$.

3.3.2 Safety

By Property 1, a protocol is safe with respect to \prec_{co} if and only if the order on local update applications at each MCS process is compliant with the order induced by \rightarrow_{co} on send events of updates. Thus,

Definition 4. (*Safety*) Let \hat{E} be a distributed computation generated by a protocol P . P is safe if and only if: $\forall m_w, m_{w'} \in M_{\hat{E}}, \text{send}_j(m_w) \rightarrow_{co} \text{send}_k(m_{w'})$ implies that at all common destinations s_i of the two updates, $\text{apply}_i(w)$ locally precedes $\text{apply}_i(w')$.

An activation predicate of a safe protocol has to stop the application of any update message m_w that arrives out of order with respect to \rightarrow_{co} .

3.3.3 Optimal Activation Predicate

With the \rightarrow_{co} relation defined, Baldoni et al. gave an optimal activation predicate in (13) as follows:

Definition 5. (*Optimal Activation predicate*)

$$A_{OPT}(m_w, e) \equiv \nexists m_{w'} : (\text{send}_j(m_{w'}) \rightarrow_{co} \text{send}_k(m_w) \wedge \text{apply}_i(w') \notin E_i|_e)$$

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

A protocol is optimal in terms of the activation predicate if its activation predicate is false only if there exists an update message $m_{w'}$ such that $\text{send}_j(m_{w'}) \rightarrow_{co} \text{send}_k(m_w)$ and $m_{w'}$ has not yet been applied locally at s_i . The key to design the optimal activation predicate is to track dependencies under the \rightarrow_{co} relation and not the Lamport's \rightarrow relation.

This activation predicate cleanly represents the causal memory's requirement: a write operation shall not be seen by an application process before any causally preceding write operations. It is optimal because the moment this activation predicate $A_{OPT}(m_w, e)$ becomes true is the earliest instant that the update m_w can be applied. The activation predicate used in the original paper describing causal memory (15) uses the happened before relation. This activation predicate is given below as A_{ORG} . As shown by Baldoni et al. (13), it does not achieve optimality.

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

3.4 Causal Consistency Algorithms

In this section, we present two algorithms — Full-Track and Opt-Track — implementing causal memories in a partially replicated distributed shared memory system, both of which adopt the optimal activation predicate A_{OPT} . Opt-Track is a message and space optimal algorithm for a partially replicated system. Subsequently, as a special case of this algorithm, we derive an optimal algorithm — Opt-Track-CRP — for the fully replicated case, that is optimal and has lower message size, time, and space complexities than the Baldoni et al. algorithm (13).

3.4.1 Full-Track Algorithm

Since the system is partially replicated, each application process performing a write operation will only write to a subset of all the sites in the system. Thus, for an application ap_i and

a site s_j , not all write operations performed by ap_i will be seen by s_j . This makes it necessary to distinguish the destinations of ap_i 's write operations. The activation predicate A_{OPT} can be implemented by 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 transmission generates a causal relationship between two processes. However, under the \rightarrow_{co} relation, it is reading the value that was written previously by another application process that generates a causal relationship between two processes. Thus, the *Write* clock piggybacked on messages generated by the **Multicast**(m) primitives should not be merged with the local *Write* clock at the message reception or at the apply event. It should be delayed until a later read operation which reads the value that comes with the message and generates the corresponding return event.

3.4.1.1 Data Structure

We now give the formal descriptions in Algorithm 1. At each site s_i , the following data structures are maintained:

1. $Write_i[1 \dots n, 1 \dots n]$: the *Write* clock (initially set to 0s). $Write_i[j, k] = a$ means that the number of updates sent by application process ap_j to site s_k that causally happened before under the \rightarrow_{co} relation is a .
2. $Apply_i[1 \dots n]$: an array of integers (initially set to 0s). $Apply_i[j] = a$ means that a total number of a updates written by application process ap_j have been applied at site s_i .
3. $LastWriteOn_i\langle \text{variable id}, Write \rangle$: a hash map of *Write* clocks. $LastWriteOn_i\langle h \rangle$ stores the *Write* clock value associated with the last write operation on variable x_h which is locally replicated at site s_i .

3.4.1.2 Correctness and Optimality Outlines

Let $send_j(m_w) \rightarrow_{co} send_k(m_{w'})$ and both messages be sent to site s_i . By Definition 4, it needs to be shown that $apply_i(w)$ locally precedes $apply_i(w')$.

Let m_w update variable x . We apply the 4 cases of the \rightarrow_{co} relation (Definition 3) to our scenario.

1. If $j = k$, $Write_k[k, i]$ at $send_j(m_w)$ is less than $Write_k[k, i]$ at $send_k(m_{w'})$ due to line (2).
2. If $j \neq k$ and condition (2) holds (corresponding to a local read at s_k), $LastWriteOn_k\langle x \rangle$.
 $Write[j, i]$ gets copied into $Write_k[j, i]$ in line (12) using the max operation. This value is less than or equal to $Write_k[j, i]$ at $send_k(m_{w'})$.
3. If $j \neq k$ and condition (3) holds (corresponding to a remote read from s_l), $LastWriteOn_l\langle x \rangle$.
 $Write[j, i]$ is fetched to k and gets copied into $Write_k[j, i]$ in line (10) using the max operation. This value is less than or equal to $Write_k[j, i]$ at $send_k(m_{w'})$.

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

```

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

READ( $x_h$ ):
8 if  $x_h$  is not locally replicated then
9   RemoteFetch[ $f(x_h)$ ] from any site  $s_j$  that replicates  $x_h$  to get  $x_h$  and  $LastWriteOn_j\langle h \rangle$ ;
10   $\forall k, l \in [1 \dots n], Write_i[k, l] := \max(Write_i[k, l], LastWriteOn_j\langle h \rangle.Write[k, l])$ ;
11 else
12   $\forall k, l \in [1 \dots n], Write_i[k, l] := \max(Write_i[k, l], LastWriteOn_i\langle h \rangle.Write[k, l])$ ;
13 return  $x_h$ ;

On receiving  $m(x_h, v, W)$  from site  $s_j$ :
14 wait until ( $\forall k \neq j, Apply_i[k] \geq W[k, i] \wedge Apply_i[j] = W[j, i] - 1$ );
15  $x_h := v$ ;
16  $Apply_i[j] ++$ ;
17  $LastWriteOn_i\langle h \rangle := W$ ;

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

```

4. By applying transitivity involving any combination of the above three cases, $Write_j[j, i]$ at $send_j(m_w)$ is less than or equal to $Write_k[j, i]$ at $send_k(m_{w'})$.

In all cases, $Write_j[j, i]$ at $send_j(m_w)$ is less than or equal to $Write_k[j, i]$ at $send_k(m_{w'})$. The activation predicate implemented at line (14) and FIFO channels ensure that $apply_i(w)$ locally precedes $apply_i(w')$ at s_i .

3.4.1.3 Liveness

It is easy to see that all write operations $w_i(x)a$ invoked by an application process ap_i are eventually applied at all replicas of x . When the write update arrives at a replica site s_j , s_j can delay it only until the activation predicate in line (14) is satisfied. Assuming reliable channels, processes executing a step in a finite time, and the OS scheduler being fair, the update will be applied in a finite number of steps.

3.4.1.4 Optimality of the Activation Predicate

We need to show that the activation predicate becomes true at the earliest possible instant for each message update received. We can see from this algorithm that, instead of merging the piggybacked *Write* clock at message reception, it is delayed until a later read operation at line (10) (remote read) and (12) (local read). This implements tracking causality under the \rightarrow_{co} relation, which is the key to implementing the activation predicate optimally. Thus, for any update m , the duration from event $receipt_i(m)$ to $apply_i(m)$ is minimal because only those dependencies that are essential to be satisfied under \rightarrow_{co} are guaranteed to be satisfied before the activation predicate becomes true. The activation predicate A_{OPT} is implemented at line (14).

3.4.2 Opt-Track Algorithm

Algorithm Full-Track achieves optimality in terms of the activation predicate. However, in other aspects, it can still be further optimized. We notice that, each message corresponding to a write operation piggybacks an $O(n^2)$ matrix, and the same storage cost is also incurred at each site s_i . Kshemkalyani and Singhal proposed the necessary and sufficient conditions on the

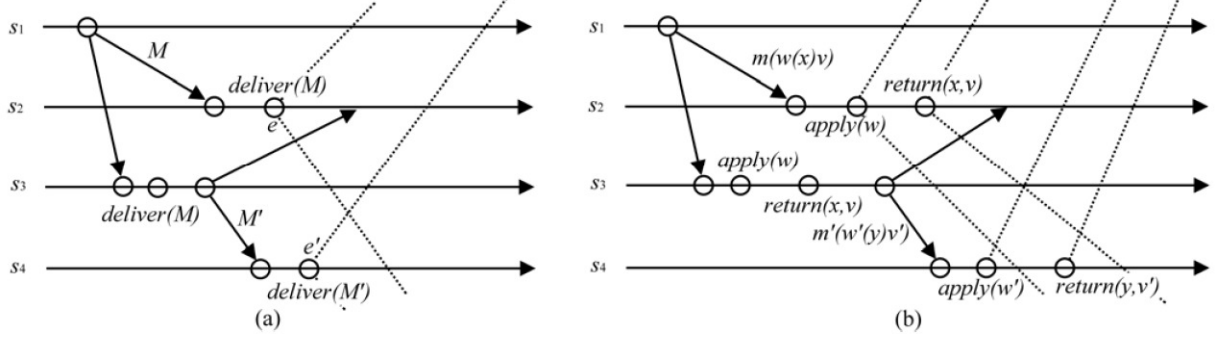


Figure 2. Illustration of the two conditions for destination information to be redundant. (a) For causal message ordering algorithms, the information is “ s_2 is a destination of M ”. The causal future of the relevant message delivery events are shown in dotted lines. (b) For causal memory algorithms, the information is “ s_2 is a destination of m ”. The causal future of the relevant *apply* events and the causal future (under the \rightarrow_{co} relation) of the relevant *return* events are shown in dotted lines.

information for causal message ordering and designed an algorithm implementing these optimality conditions (11; 12) (hereafter referred to as the KS algorithm). The KS algorithm aims at reducing the message size and storage cost for causal message ordering algorithms in message passing systems. The ideas behind the KS algorithm exploit the transitive dependency of causal deliveries of messages and encode the information being tracked. In the KS algorithm, each site keeps a record of recently received messages from each other site. The list of destinations of a message is also kept in each record (the KS algorithm assumes multicast communication) and is progressively pruned, as described below. With each outgoing message, these records are also piggybacked. The KS algorithm achieves another optimality, in the sense that no redundant

destination information is recorded. There are two situations when the destination information can become redundant. These are illustrated in Figure 2(a).

1. When message M is delivered at site s_2 (we denote this event as e), then the information that s_2 is part of message M 's destination no longer needs to be remembered in the causal future of e . This is because the delivery of M at s_2 is guaranteed at events in the causal future of e .

In addition, we *implicitly* remember in the causal future of e that M has been delivered to s_2 , to clean the logs at other sites.

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

In addition, we *implicitly* remember in the causal future of events of type e' that M is transitively guaranteed to be delivered to s_2 , to clean the logs at other sites.

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

Although the KS algorithm is for message passing systems, its ideas of deleting unnecessary dependency information still apply to distributed shared memory systems. We can adapt the KS algorithm to a partially replicated DSM system to implement causal memory there. Now, each site s_i will maintain a record of the most recent updates received from every site, that causally happened before under the \rightarrow_{co} relation. Each such record also keeps a list of destinations representing the set of replicas receiving the corresponding update. When performing a write operation, the outgoing update messages will piggyback the currently stored records. When receiving an update message, the optimal activation predicate A_{OPT} is used to determine when to apply the update. Once the update is applied, the piggybacked records will be associated with the updated variable. When a later read operation is performed on the updated variable, the records associated with the variable will be merged into the locally stored records to reflect the causal dependency between the read and write operations. Similar to the KS algorithm, we can prune redundant destination information using the following conditions. These are illustrated in Figure 2(b).

- **Propagation Condition 1:** When an update m corresponding to write operation $w(x)v$ is applied at site s_2 , then the information that s_2 is part of the update m 's destinations no longer needs to be remembered in the causal future of the event $apply_2(w)$.

- **Pruning Condition 1:** In addition, we *implicitly* remember in the causal future (under the \rightarrow_{co} relation) of event $return_2(x, v)$ (and events $remote_return_2(r_*(x)v)$) that m has been delivered to s_2 , to clean the logs at other sites.
- **Propagation Condition 2:** For two updates $m_{w(x)v}$ and $m'_{w'(y)v'}$ such that $send(m) \rightarrow_{co} send(m')$ and both updates are sent to site s_2 , the information that s_2 is part of update m 's destinations is irrelevant in the causal future of the event $apply(w')$ at all sites s_k receiving update m' . (In fact, it is redundant in the causal future of $send(m')$, other than m' sent to s_2 .) This is because, by transitivity, applying update m' at s_2 in causal order with respect to a message m'' sent causally later to s_2 will infer the update m has already been applied at s_2 .
- **Pruning Condition 2:** In addition, 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.

The logs at the sites are cleaned as follows. The algorithm explicitly tracks (source, timestamp, Dests) per multicast message M in the log and on the message overhead. The shorthand we use for this information is $M_{i,a}.Dests$ to indicate source i and local timestamp a . Log entries are denoted by l and message overhead entries are denoted by o . The algorithm implicitly tracks messages that are delivered (Pruning Condition 1) or transitively guaranteed to be delivered in causal order (Pruning Condition 2) as follows.

- **Implicit Tracking 1:** $\exists d \in M_{i,a}.Dests$ such that $d \in l_{i,a}.Dests \wedge d \notin o_{i,a}.Dests$. 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. (Likewise with the roles of l and o reversed.) 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. Observe that entries of such format will accumulate. Such entries can be discarded and implicitly inferred as follows.

- **Implicit Tracking 2:** If $a_1 < a_2$ and $l_{i,a_2} \in LOG_j$, then l_{i,a_1} is implicitly or explicitly in LOG_j . Entries of the form $l_{i,a_1}.Dests = \emptyset$ can be inferred by their absence and should not be stored.

Implicit Tracking 1 and Implicit Tracking 2 in a combined form implement Pruning Condition 1 and Pruning Condition 2. When doing Implicit Tracking 1 and Implicit Tracking 2, if explicitly stored information is encountered, it must be deleted as soon as it comes into the causal future of the implicitly tracked information.

Notice that even if the destination list in a message M 's record becomes \emptyset at a certain event e in site s_i , that record still needs to be kept until a later message from message M 's sender is delivered at s_i . This is because although M 's destination list becomes \emptyset at s_i , it might still be non-empty at other sites. Thus, by piggybacking M 's record with an empty destination list, we can prune M 's destination list at other sites in the causal future of event e (using Implicit Tracking 1). This is illustrated in Figure 3. Notice that, after the deliveries of message M_2 and M_3 , the destination list in the message M_1 's record at site s_4 becomes empty. If we delete M_1 's record at this time, then when message M_4 is later sent to site s_3 , there is no way for s_3 to know that it can delete site s_2 from its local record of message M_1 's destination

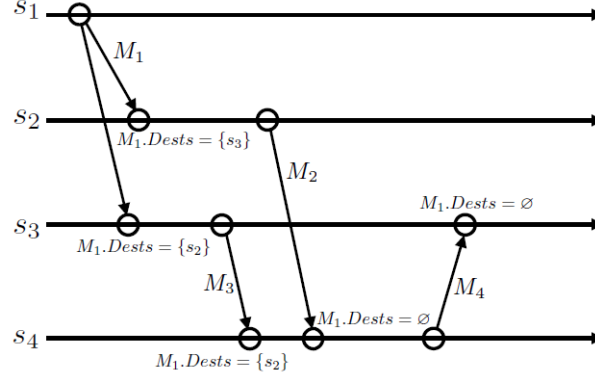


Figure 3. Illustration of why it is important to keep a record even if its destination list becomes empty. For clarity, the *apply* and *return* events after each message delivery are omitted.

list. M_1 's record can be deleted at site s_4 after another message from s_1 is delivered at s_4 . This technique is important for achieving optimality of no redundant destination information of already delivered messages (Pruning Condition 1) and of messages transitively guaranteed to be delivered in causal order (Pruning Condition 2). However, note that such records with \emptyset destination lists may accumulate. Only the latest such record per sender needs to be maintained; the presence of other such records can be implicitly inferred (using Implicit Tracking 2).

3.4.2.1 Data Structure

We give the formal algorithm in Algorithm 2. The following data structures are maintained at each site:

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

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

In a write operation, rather than multicast the same message to all replica sites, the meta-data per replica site is tailored in lines (2)-(9) to minimize its space overhead. L_w is the working variable used to modify the local LOG_i to send to each replica site.

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

Algorithm 3 gives the procedures used by Algorithm Opt-Track (Algorithm 2). Function **PURGE** removes old records with \emptyset destination lists, per sender process (Implicit Tracking 2). On a read operation of variable x_h , function **MERGE** merges the piggybacked log of the corresponding write to x_h with the local log LOG_i . In this function, new dependencies get added to LOG_i

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

```

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

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

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

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

```

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

PURGE:

```

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

```

MERGE(LOG_i, L_w):

```

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

```

and existing dependencies in LOG_i are pruned, based on the information in the piggybacked data L_w . The merging implements the optimality techniques of Implicit Tracking 1, described before.

3.4.2.2 Correctness and Optimality Outlines

Let $send_j(m_w) \rightarrow_{co} send_k(m_{w'})$ and both messages be sent to site s_i . By Definition 4, it needs to be shown that $apply_i(w)$ locally precedes $apply_i(w')$.

On event $send_j(m_w)$, the entry that s_i is a destination is added in the local LOG_j (line 13) as well as in the logs of all other replicas except s_i (lines (28)-(30)). This information gets propagated in the various local logs and the message overheads until Propagation Conditions 1 and 2 become applicable, viz., until m_w is delivered to s_i or is transitively guaranteed to be delivered in causal order to s_i . In the latter case, there is a causally more recent entry in the

logs with destination s_i . If LOG_k contains information about this more recent entry or about m_w itself at event $send_k(m_{w'})$, that information will be piggybacked on $m_{w'}$ sent to s_i . The activation predicate at lines (24)-(25) ensures that the update due to m_w or due to this more recent entry (as the case might be) is applied before $apply_i(w')$.

3.4.2.3 Liveness

This proof is similar to that for the Full-Track algorithm.

3.4.2.4 Optimality of the Activation Predicate

This proof is similar to that for the Full-Track algorithm.

3.4.2.5 Optimality of Log Space and Message Overhead Space

The log sizes are the minimal possible at any point in time. This is because the Propagation Conditions 1 and 2 are implemented in conjunction with the Pruning Conditions 1 and 2 to enforce pruning of all information that is not necessary for safety. Pruning Conditions 1 and 2 are implemented by Implicit Tracking 1 and 2 to remove entries from the logs. Implicit Tracking 1 and 2 are implemented by procedure **MERGE**, and procedure **PURGE** deletes the old records with null destination lists to maintain logs of minimal size.

The local log gets piggybacked as the message overhead. In addition, when preparing the message overhead before sending the message, lines (4)-(6) and lines (7)-(8) ensure that by removing transitive dependencies and null entries, respectively, the message overhead space is minimal.

At the expense of slightly larger message overhead, we can distribute the *Write* processing in lines (3)-(8) of Algorithm 2 to the receivers' sites after line (27). Instead of the loop in

line (4), send the *LOG*; and on its receipt, for each entry o in L_w , subtract $x_h.replicas$ from $o.Dests$. This reduces the time complexity of a write operation from $O(n^2p)$ to $O(n^2)$.

3.4.3 Opt-Track-CRP: Adapting Opt-Track Algorithm to Fully-Replicated Systems

Algorithm Opt-Track can be directly applied to fully replicated DSM systems. Furthermore, since in the full replication case, every write operation will be sent to exactly the same set of sites, namely all of them, there is no need to keep a list of the destination information with each write operation. Each time a write operation is sent, all the write operations it piggybacks as its dependencies will share the same set of destinations as the one being sent, and their destination list will be pruned by Propagation Condition 2. Also, when a write operation is received, all the write operations it piggybacks also have the receiver as part of their destinations. So, when checking for the activation predicate at lines (24)-(25) in Algorithm 2, all piggybacked write operations need to be checked. With these additional properties in the full replication scenario, we can represent each individual write operation using only a pair $\langle i, clock_i \rangle$, where i is the site id and $clock_i$ is the local write operation counter at site s_i when the write operation is issued. In this way, we bring the cost of representing a write operation from potentially $O(n)$ down to $O(1)$. This improves the algorithm's scalability when the shared memory is fully replicated.

In fact, Algorithm 2's scalability can be further improved in the fully replicated scenario. In the partially replicated case, keeping entries with empty destination list as long as they represent the most recent applied updates from some site is important, as it ensures the optimality that no redundant destination information is transmitted. However, this will also require each site

to almost always maintain a total of n entries. In the fully replicated case, we can also decrease this cost. We observe that, once a site s_3 issues a write operation $w'(x_2)u$, it no longer needs to remember any previous write operations, such as $w(x_1)v$, stored in the local log. This is because all the write operations stored in the local log share the same destination list as w' . Thus, by making sure the most recent write operation is applied in causal order, all the previous write operations sent to all sites are guaranteed to be also applied in causal order. Similarly, after the activation predicate becomes true and the write operation w' is applied at site s_1 , only w' itself needs to be remembered in $LastWriteOn_1(2)$. This is illustrated in Figure 4.

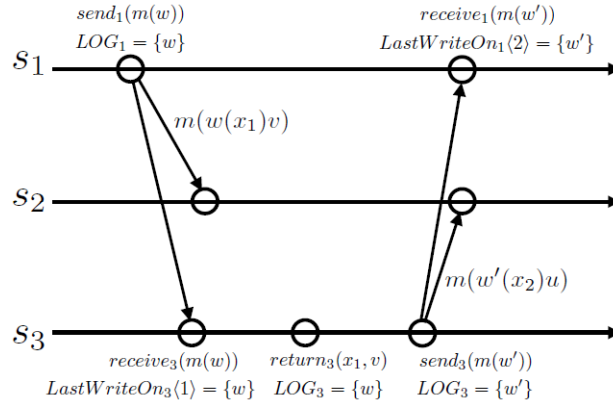


Figure 4. In fully replicated systems, the local log will be reset after each write operation. Also, when a write operation is applied, only the write operation itself needs to be remembered. For clarity, the *apply* events are omitted in this figure.

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

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

3.4.3.1 Formula

We give the formal algorithm of a special case of Algorithm 2, optimized for the fully replicated DSMs. The algorithm is listed in Algorithm 4. Each site still maintains the same data structures as in Algorithm 2, the only difference lies in that there is no need to maintain the destination list for each write operation in the local log, and hence the format of the log

entries becomes the pair $\langle i, clock_i \rangle$. Algorithm 4 assumes a highly simplified form. However, it is very systematically derived by adapting Algorithm 2 to the fully replicated case. Algorithm 4 is significantly better than the algorithm in (13) in multiple respects.

Function MERGE works similarly to that in Opt-Track. There are two merge cases in Opt-Track-CRP. First, when any log entry $l_{s,t}$ with clock time t for site $s = j$ in LOG is older than the piggybacked log entry $\langle j, clock_j \rangle$ (i.e., $t < clock_j$), it implies that this entry information is out of date compared with $\langle j, clock_j \rangle$. It also means that there is no entry in LOG whose time clock is greater than $clock_j$. If so, this entry $l_{s,t}$ will be removed from LOG and entry $\langle j, clock_j \rangle$ needs to be united with LOG . Note that, in full replication, it is impossible that there are multiple entries with the same site id and different clock time marks (i.e., $LOG = \{\langle j, clock_j \rangle, \langle j, clock'_j \rangle \dots\}$). Second, if there is no log entry $l_{s,t}$ in which site id s is equal to j , then, the piggybacked log entry $\langle j, clock_j \rangle$ will also be merged with LOG .

3.4.4 Complexity

In this section, we analyse the complexity of the algorithms proposed. Four metrics are used in the complexity analysis:

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

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

```

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

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

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

  MERGE( $LOG_i, \langle j, clock_j \rangle$ ):
14   $unionflag := 1$ ;
15  for all  $l_{s,t} \in LOG_i$  such that  $s = j$  do
16   if  $t < clock_j$  then
17    delete  $l_{s,t}$  from  $LOG_i$ ;
18   else
19     $unionflag := 0$ ;
20  if  $unionflag$  then  $LOG_i := LOG_i \cup \{\langle j, clock_j \rangle\}$ ;

```

- space complexity: the space complexity at each site s_i for storing local logs and the

$LastWriteOn$ log.

The following parameters are used in the analysis:

- n : the number of sites in the system
- q : the number of variables in the DSM system

TABLE I. Complexity measures of causal memory algorithms.

Metric	Full-Track	Opt-Track	Opt-Track-CRP	optP (13)
Message count	$((p-1) + \frac{n-p}{n})w + 2r\frac{(n-p)}{n}$	$((p-1) + \frac{n-p}{n})w + 2r\frac{(n-p)}{n}$	$(n-1)w$	$(n-1)w$
Message size	$O(n^2pw + nr(n-p))$	$O(n^2pw + nr(n-p))$ amortized $O(npw + r(n-p))$	$O(nwd)$	$O(n^2w)$
Time Complexity	write $O(n^2)$ read $O(n^2)$	write $O(n^2p)$ read $O(n^2)$	write $O(d)$ read $O(1)$	write $O(n)$ read $O(n)$
Space Complexity	$O(\max(n^2, npq))$	$O(\max(n^2, npq))$ amortized $O(\max(n, pq))$	$O(\max(d, q))$	$O(nq)$

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

Table I summarizes the results.

3.4.4.1 Full-Track Algorithm

3.4.4.1.1 Message Count Complexity

The Full-Track algorithm assumes partial replication, thus each write operation will only incur $(p-1) + \frac{n-p}{n}$ number of messages, accumulating $((p-1) + \frac{n-p}{n})w$ number of messages across all write operations. However, since now read operations might need to read from remote site, assuming the variables are evenly replicated across the entire system and the read operations read variables in a truly random manner, then an additional $\frac{2r(n-p)}{n}$ number of messages will be generated by the read operations. In total, the message count complexity of the Full-Track algorithm is $((p-1) + \frac{n-p}{n})w + \frac{2r(n-p)}{n}$.

3.4.4.1.2 Message Space Complexity

The Full-Track algorithm maintains a local *Write* clock which is a matrix of size n^2 . Since this clock is piggybacked with each message containing a write operation and each message that returns a variable's value to a remote site on a remote read operation, each message generated by a send event and the remote return event in the Full-Track algorithm has a size of $O(n^2)$,

(whereas each message generated by a fetch event has a small and constant byte count). Thus, the total message size complexity of the Full-Track algorithm is $npw(n-1) + nr(n-p)$, which is $O(n^2pw + nr(n-p))$.

3.4.4.1.3 Time Complexity

Each write operation updates the local *Write* clock for each of the p replicas before invoking the **Multicast** primitive. This incurs an $O(p)$ time complexity. There is an added cost of $O(n^2)$ for copying the $Write_i$ matrix into the $LastWriteOn_i$ log.

For the read operation, merging the *Write* clock associated with the variable to be read with the local clock incurs an $O(n^2)$ time complexity.

3.4.4.1.4 Space Complexity

This is composed of the cost to maintain the local *Write* clock and the cost to maintain the $LastWriteOn$ log. The local *Write* clock takes $O(n^2)$ space. Each entry in the $LastWriteOn$ log that is associated with a locally replicated variable is of size $O(n^2)$. Assuming the variables are evenly distributed across the entire system, each site will replicate a total of $\frac{pq}{n}$ variables. Thus, the $LastWriteOn$ log incurs an $O(npq)$ space complexity. The Full-Track algorithm's space complexity is $O(\max(n^2, npq))$.

3.4.4.2 Opt-Track Algorithm

3.4.4.2.1 Message Count Complexity

As the Opt-Track algorithm produces the same message pattern for both the read and write operations as the Full-Track algorithm, its message count complexity is also the same, being $((p-1) + \frac{n-p}{n})w + \frac{2r(n-p)}{n}$.

3.4.4.2.2 Message Space Complexity

Different from the Full-Track algorithm, the Opt-Track algorithm does not maintain a matrix at each site. Instead, it keeps a log of only the necessary write operations from all that happened in the causal past, as in the KS algorithm. Each record of a write operation also maintains its destination list, which contains up to p sites. The optimality of the Opt-Track algorithm guarantees that only the necessary write operations and destination information are kept, and piggybacked. Similar to the Full-Track algorithm, each message generated by a send event and a remote return event contributes to the main meta-data overhead. The messages generated by a fetch event are small and of constant byte size.

In the KS algorithm, the upper bound on the size of the log and the message overhead is $O(n^2)$ (12). This has also been shown using an adversarial argument (62), viz., if you cannot decide the distribution of replicas, then partial replication incurs the same costs as does full replication as each process has to manage information on all the replicas. However, Chandra et al. (63; 64) showed through extensive simulations that the amortized log size and message overhead size of the KS algorithm is approximately $O(n)$. This is because the optimality conditions implemented ensure that only necessary destination information is kept in the log and purged as soon as possible. This also applies to the Opt-Track algorithm because the same optimization techniques are used. Since the log is piggybacked with each message containing a write operation and each message that returns a variable's value to a remote site, the total message size complexity of the Opt-Track algorithm is $O(n^2pw + nr(n - p))$. However, this is

only the asymptotic upper bound. The amortized message size complexity of the Opt-Track algorithm is approximately $O(npw + r(n - p))$.

3.4.4.2.3 Time Complexity

For a write operation, for each replica the algorithm prunes the destination information stored in the local log accordingly before piggybacking it with the message. This will incur an $O(n^2p)$ time complexity. This subsumes the $O(n^2)$ time complexity incurred when a message is received. In Section 3.4.2.5, we pointed out a way to bring down the $O(n^2p)$ complexity to $O(n^2)$, at the expense of slightly larger message overhead.

For the read operation, the MERGE operation will merge the log associated with the variable to be read with the local log. If the local logs are maintained in such a way that all the entries $l_{j,t_j} \in LOG_i$ are stored in the ascending order of j and t_j , then the MERGE operation can be completed with only one pass through both logs. Thus, the incurred time complexity of a read operation is $O(n^2)$.

3.4.4.2.4 Space Complexity

This is composed of the cost to maintain the local log (corresponding to the *Write* clock) and the cost to maintain the *LastWriteOn* log. The local log (LOG_i) takes $O(n^2)$ space. Since for each of the $\frac{qp}{n}$ locally replicated variables, the Opt-Track algorithm needs to store the log (of size $O(n^2)$) piggybacked with the most recent write operation updating that particular variable, the size of the *LastWriteOn* log is $O(npq)$. Thus, the space complexity is $O(\max(n^2, npq))$. Still, this is only the asymptotic upper bound. The amortized space complexity will be approximately

$O(\max(n, pq))$, since the amortized size of the local log is approximately $O(n)$ on average, instead of $O(n^2)$ (63; 64).

3.4.4.3 Opt-Track-CRP Algorithm

3.4.4.3.1 Message Count Complexity

Each write operation incurs a total of $n - 1$ messages, however the read operation will always read from the local copy and thus generate no messages. In total, the message count complexity of the Opt-Track-CRP algorithm is $(n - 1)w$.

3.4.4.3.2 Message Space Complexity

Being a special case of the Opt-Track algorithm, the Opt-Track-CRP algorithm does not keep the destination list for each record of a write operation, nor does it always maintain n entries in the local log at each site (as discussed in Section 3.4.3). This means the size of a write operation's record is only $O(1)$ and the size of the local log is only determined by the number of entries in the local log, which is denoted as d in this section. As discussed in Section 3.4.3, in practice d is only a small constant number. Thus the size of the log piggybacked with each message containing a write operation is $O(d)$ and the total message size complexity of the Opt-Track-CRP algorithm is $O(nwd)$.

3.4.4.3.3 Time Complexity

The Opt-Track-CRP algorithm also has a very small time complexity, because the size of the local log is very small.

For the write operation, the algorithm rewrites the local log and thus incurs only an $O(1)$ time complexity. The processing on receiving a write broadcast checks for each piggybacked log entry, which is the size of the sender's log, and takes $O(d)$ time.

For the read operation, the **MERGE** operation merges the log associated with the variable to be read with the local log. As the logs stored in the *LastWriteOn* log always contains only one write operation, the **MERGE** operation can be completed within $O(1)$ time.

3.4.4.3.4 Space Complexity

This is composed of the size of the *LastWriteOn* log and the size of the local log. The *LastWriteOn* log contains q logs, each containing a single write operation; thus its size is $O(q)$. For the local log, it contains d entries of write operations, thus having a size of $O(d)$. The space complexity of the Opt-Track-CRP algorithm is thus $O(\max(d, q))$.

3.4.5 Experiments

The implementations of the four causal consistency replicated protocols – Full-Track, Opt-Track, Opt-Track-CRP, and *optP* (13) – were realized. 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.

3.4.5.1 Partial Replication Protocols: Meta-data size

As analyzed on Section 3.4.4.1.2 and 3.4.4.2.2, in the Full-Track and Opt-Track protocols, SM and RM messages contribute to the meta-data overheads, whereas FM messages are of small and constant byte size.

3.4.5.1.1 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 Figure 5. 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 ~ 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

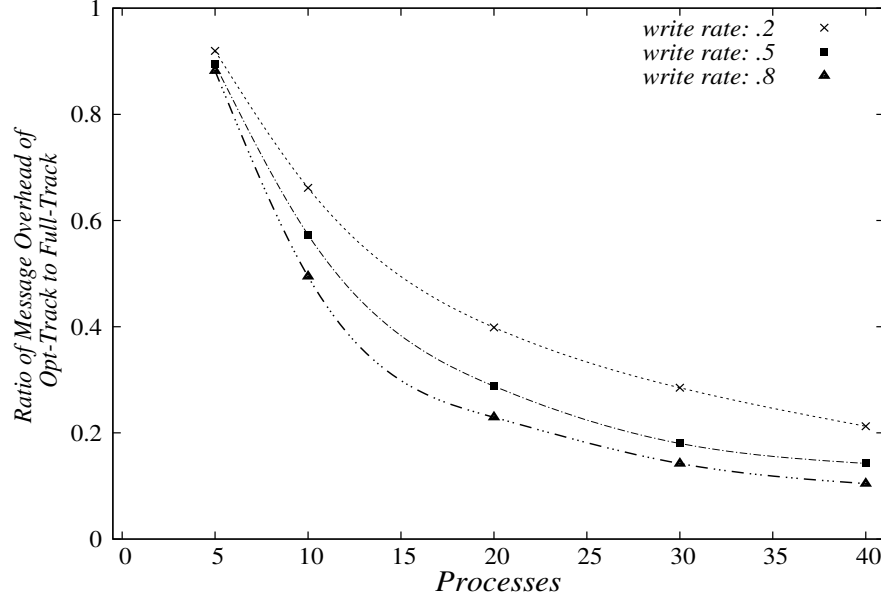


Figure 5. Total message meta-data space overhead as a function of n and w_{rate} in partial replication protocols.

Figure 5 that a higher write rate magnifies the difference of the message space overhead between Opt-Track and Full-Track.

3.4.5.1.2 Impact of Write Rate w_{rate}

The results for the average message space overhead are shown in Figure 6, Figure 7, and Figure 8 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

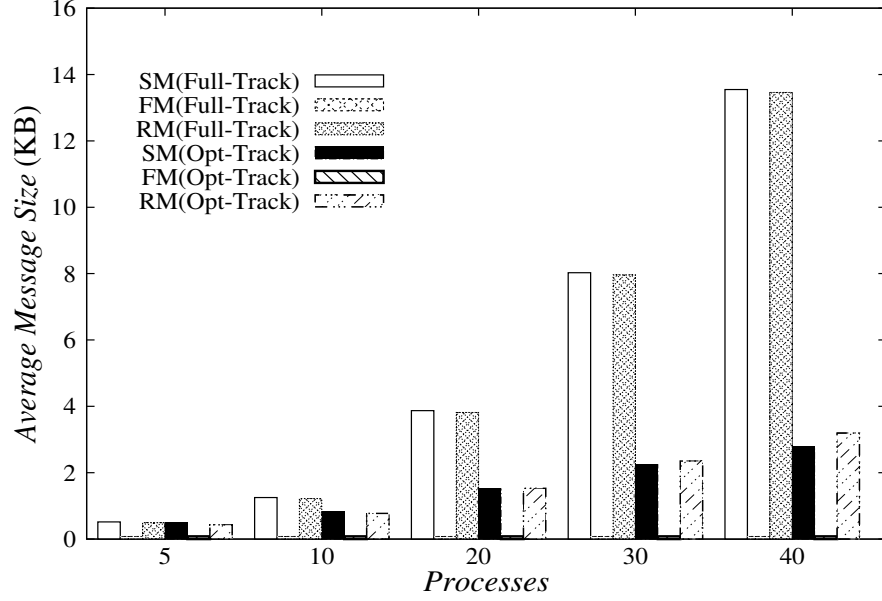


Figure 6. Average message meta-data space overhead as a function of n with lower w_{rate} (0.2) in partial replication protocols.

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, each log also involves more implicit information. 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 Figure 6 to Figure 8 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

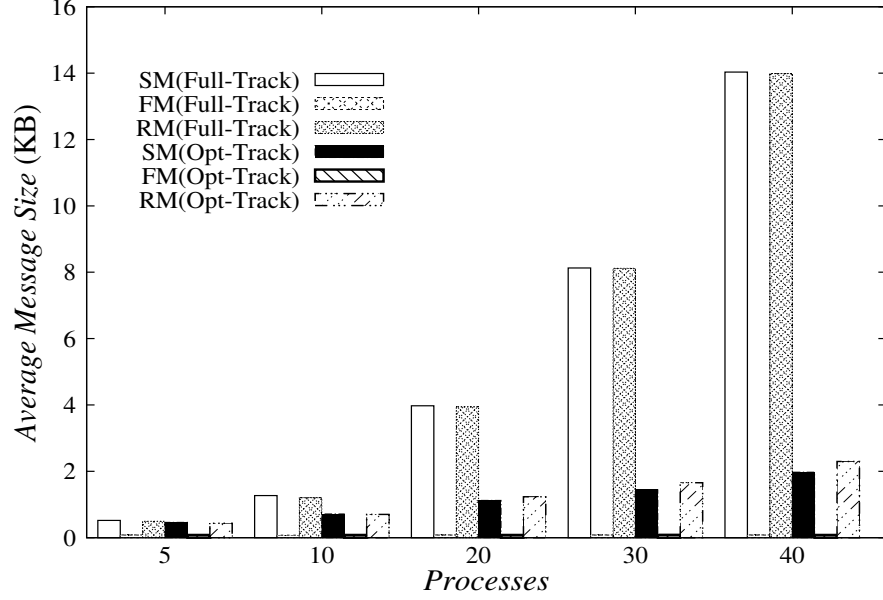


Figure 7. Average message meta-data space overhead as a function of n with medium w_{rate} (0.5) in partial replication protocols.

TABLE II. Average SM and RM space overhead for Full-Track and Opt-Track (KB)

		w_{rate}	the number of processes, n				
			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

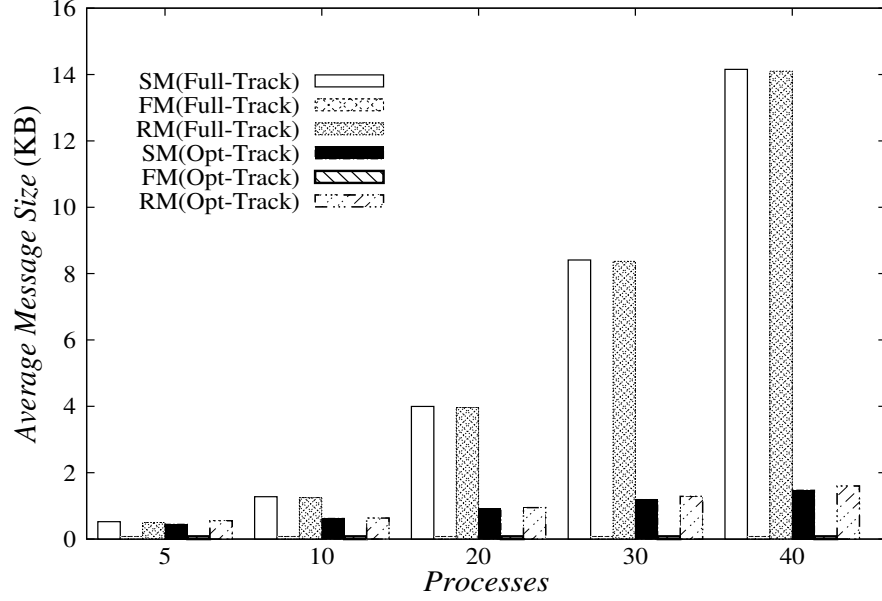


Figure 8. Average message meta-data space overhead as a function of n with higher w_{rate} (0.8) in partial replication protocols.

use the same data structure for FM message.) The analytic data is listed in Table II according to Figure 6 to Figure 8. The average SM and RM overheads decrease as the write rate increases in Opt-Track Protocol. 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 Opt-Track 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% ~ 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 3.4.2. 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 3.4.2, 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.

3.4.5.2 Full Replication Protocols: Meta-data size

The Opt-Track-CRP protocol and the *optP* protocol both use only SM messages, and no FM or RM messages.

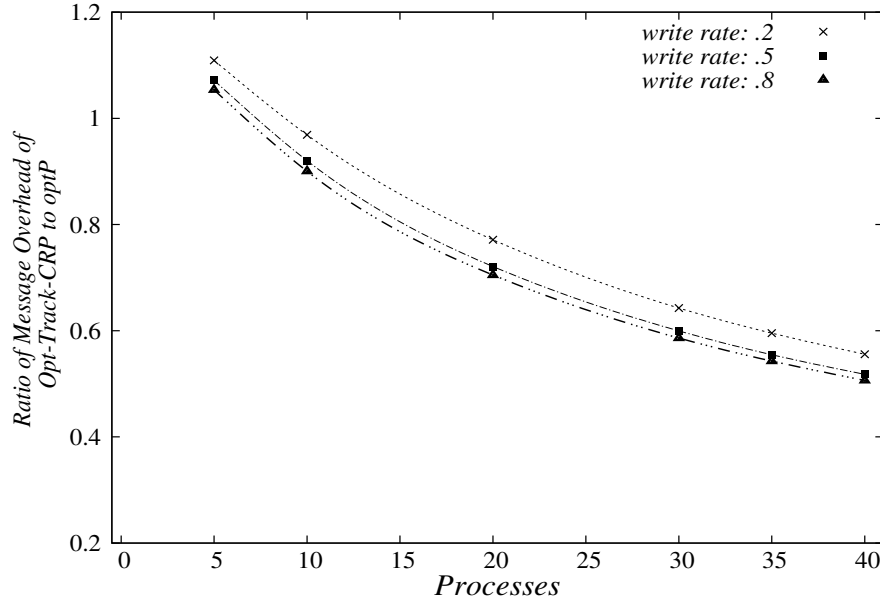


Figure 9. Total message meta-data space overhead as a function of n and w_{rate} in full replication protocols.

3.4.5.2.1 Scalability as a Function of n

The results for the ratio of message space overhead of Opt-Track-CRP to $optP$ are shown in Figure 9. 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.

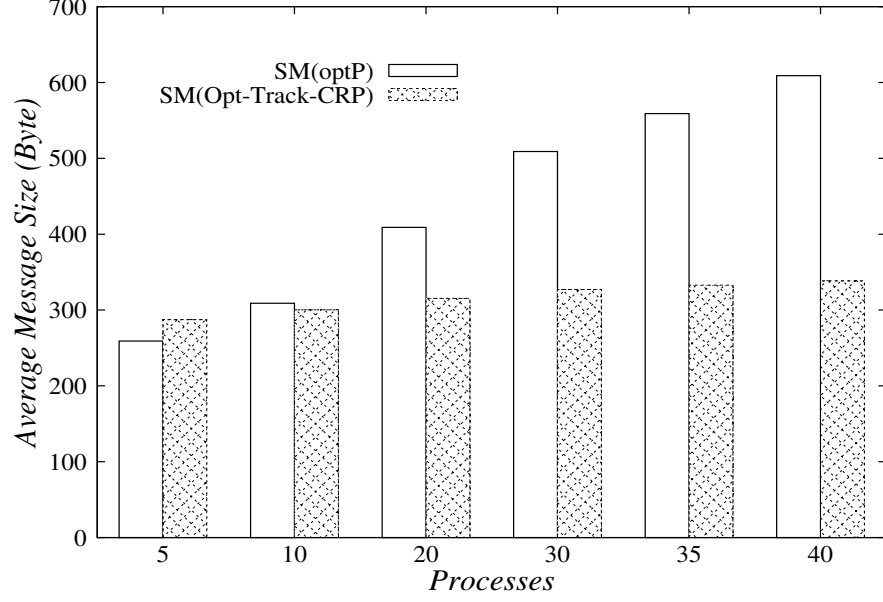


Figure 10. Average message meta-data space overhead as a function of n with lower w_{rate} (0.2) in full replication protocols.

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

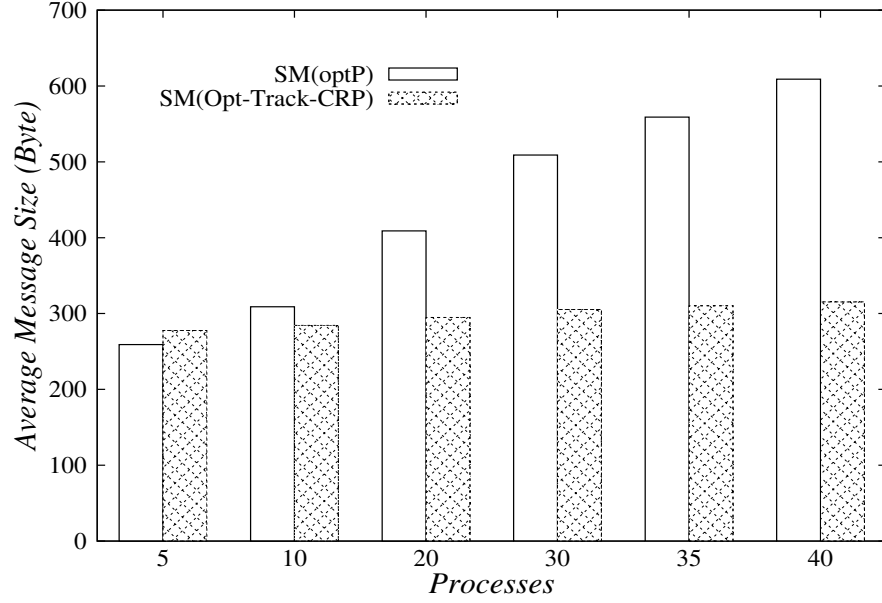


Figure 11. Average message meta-data space overhead as a function of n with medium w_{rate} (0.5) in full replication protocols.

3.4.5.2.2 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 Figure 10, Figure 11, and Figure 12 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 Figure 10 to Figure 12, 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

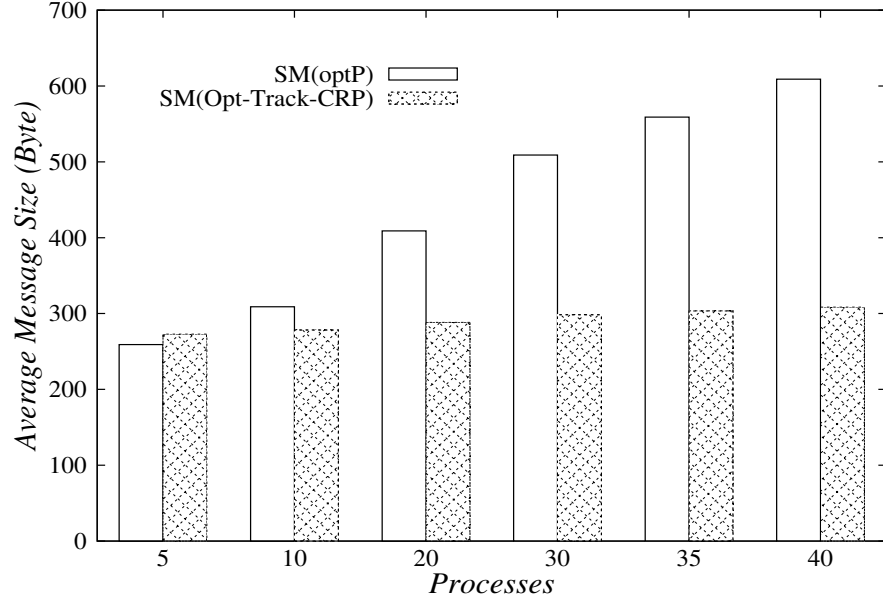


Figure 12. Average message meta-data space overhead as a function of n with higher w_{rate} (0.8) in full replication protocols.

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.

3.4.5.3 Partial Replication vs. Full Replication: Message Count

Compared with the existing causal distributed shared memory protocols, our suite of protocols (60) has the additional ability to implement causal consistency in partially replicated distributed shared memory systems. Further, the protocols in (17; 65; 16) do not provide scalability as they use a form of log serialization and exchange to implement causal consistency.

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.

3.4.5.4 Partial Replication vs. Full Replication: Message Space Overhead

Let f be the size of an image/data being written and let b be the number of bytes in an integer.

Under full replication, the net message payload size for the write multicast is $(n - 1) \times f$, and $n \times b \times (n - 1)$ for the message meta-data overheads in the worst case of Opt-Track-CRP protocol (61). In practice, the size of a write operation's record is only $O(1)$ and the size of the local log is determined only by the number of entries in the local log, denoted as d , as shown in the definition before; practically, d is only a small constant number. The read cost is zero.

Thus, the message size cost introduced by one write operation in full replication is $(n-1)f + nb(n-1)$.

Under partial replication using Opt-Track, the net message payload size is $((p-1) + \frac{(n-p)}{n})f$ for the write multicast. The read cost is $(\frac{r}{w})\frac{(n-p)}{n}f$ because there are $\frac{r}{w}$ reads per write, and $\frac{n-p}{n}$ of the reads fetch the file from a remote location. The corresponding message 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. However, it was shown in (66) through extensive simulations that the amortized log size and message overhead size is approximately $O(n)$, not $O(n^2)$. The amortized message meta-data overheads are $((p-1) + \frac{(n-p)}{n})nb + (\frac{r}{w})\frac{(n-p)}{n}(n+1)b$.

Thus, partial replication has lower message (payload + meta-data) size in the worst case if

$$\begin{aligned} (n-1)f + nb(n-1) &> \\ ((p-1) + \frac{(n-p)}{n})f + (\frac{r}{w})\frac{(n-p)}{n}f + & \quad (3.1) \\ ((p-1) + \frac{(n-p)}{n})n^2b + (\frac{r}{w})\frac{(n-p)}{n}(n^2+1)b & \end{aligned}$$

or, in practice, if

$$\begin{aligned} (n-1)f + db(n-1) &> \\ ((p-1) + \frac{(n-p)}{n})f + (\frac{r}{w})\frac{(n-p)}{n}f + & \quad (3.2) \\ ((p-1) + \frac{(n-p)}{n})nb + (\frac{r}{w})\frac{(n-p)}{n}(n+1)b & \end{aligned}$$

The above analysis was for r reads and w writes of the image/data. Consider such a system with $n=40$, $p=12$ and $w_{rate} = 0.5$. A large/medium/small data of 10KB/1KB/0.1KB,

respectively, has to be stored and transmitted in a write operation. Assume that one word holds 4 bytes ($b=4B$).

Table V summarizes the total message space overheads in the worst cases. Although full replication has lower meta-data overheads, partial replication has smaller total message space size for the larger image/data f .

Table VI shows the total message space overheads in the practical amortized sense. Note that $d < n$. As per the experiments in (66), the average meta-data overhead is 315B for Opt-Track-CRP; therefore $d = 2$. For Opt-Track, there are three types of control messages. Their structures are as shown in Table VII. SM corresponds to a multicast message arising from *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. Among the three types of control messages, SM and RM contribute mainly to the meta-data overhead: messages to write remotely, which take 1.976KB, and messages to remotely return a non-local value, which take 2.197KB. FM, to request a remote fetch of a non-local value, contributes very little to the meta-data overhead. The numerical values of the above quantities are shown in Table II. In sum, the average meta-data overhead for Opt-Track is around 2.08KB.

Thus, the experimental results closely corroborate the above theoretical analysis. Apparently, partial replication, in practice, has lower total message space overhead than full replication, no matter what size of f . This makes partial replication a viable alternative to full replication in DSM systems. Although the meta-data size of full replication is still less than

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

TABLE V. Total message overheads for full replication (Opt-Track-CRP) and partial replication (Opt-Track), when $n=40$, $p=12$, $w_{rate}=0.5$, in the worst case.

	$f=10\text{KB}$	$f=1\text{KB}$	$f=0.1\text{KB}$
Full Replication	390KB+6.09KB	39KB+6.09KB	3.9KB+6.09KB
Partial Replication	124KB+77.503KB	12.4KB+77.503KB	1.24KB+77.503KB

that of partial replication, the difference is far less than the message payload size variation between full replication and partial replication.

TABLE VI. Total message overheads for full replication (Opt-Track-CRP) and partial replication (Opt-Track), when $n=40$, $p=12$, $w_{rate}=0.5$, in practice.

	$f=10\text{KB}$	$f=1\text{KB}$	$f=0.1\text{KB}$
Full Replication	$390\text{KB}+0.15d \text{ KB}$	$39\text{KB}+0.15d \text{ KB}$	$3.9\text{KB}+0.15d \text{ KB}$
Partial Replication	$124\text{KB}+1.9402 \text{ KB}$	$12.4\text{KB}+1.9402 \text{ KB}$	$1.24\text{KB}+1.9402 \text{ KB}$

TABLE VII. Message meta-data structures in partial replication protocols

	<i>Full – Track</i>	<i>Opt – Track</i>
SM (Multicast)	x_h, v, Write	$x_h, v, \text{Site}_{id}, \text{clock}, L_w$
FM (Fetch)	x_h, v	x_h, v
RM (Remote Return)	$v, \text{LastWriteOn}\langle h \rangle$	$v, \text{LastWriteOn}\langle h \rangle$

CHAPTER 4

APPROXIMATE CAUSAL CONSISTENCY

This chapter is based on our previous publications (67; 68; 2). In this chapter, we focus on the relationship between adequate *credit* (no causal violation) and n (the number of processes) and discuss the problem of the trade-off between adequate *credit* and how much meta-data can be reduced further. The underlying system architecture is the same as the one provided in Chapter 2. The rest of this chapter is organized as follows. Section 4.1 presents the basic idea of approximate causal consistency. Section 4.2 presents the details of algorithm Approximate-Opt-Track. Section 4.3 describes some notes about the hop count instantiation of algorithm Approximate-Opt-Track. Section 4.4 shows the experimental system model. Section 4.5 demonstrates the experimental results in different settings. Section 4.6 illustrates the experimental evaluation.

4.1 Basic Idea of Approx-Opt-Track

Based on Opt-Track protocol, 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 Figure 13. 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 Figure 14. It is the same as in Figure 13, 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

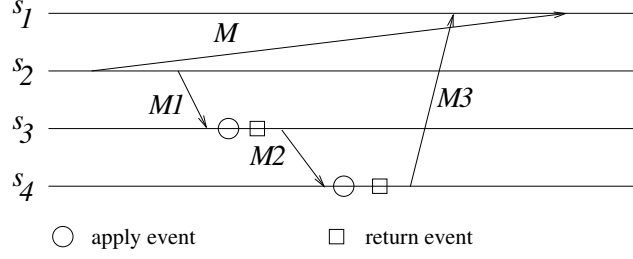


Figure 13. Illustration of causal consistency violation if credits are exhausted.

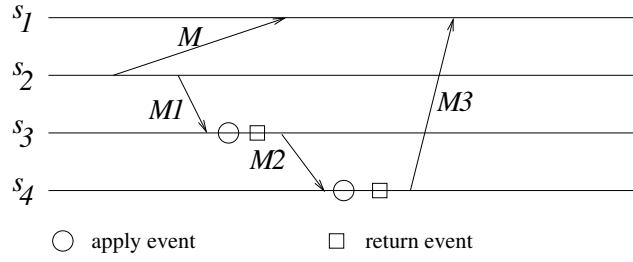


Figure 14. Illustration of meta-data reduction when credits are exhausted.

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 .

4.2 Approx-Opt-Track

Algorithm 5 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 LOG s. $LastWriteOn_i\langle h \rangle$ contains the piggybacked LOG for the most recent update applied at site s_i to locally replicated variable x_h .

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 5 implements the optimality mechanisms described in algorithm Opt-Track (61).

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 6 shows two functions used in Algorithm Approx-Opt-Track (Algorithm 5). 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 dependen-

Algorithm 5: Approx-Opt-Track Algorithm, which is a modification of Algorithm Opt-Track (61) (Code at site s_i)

```

WRITE( $x_h, v$ ):
1   $clock_i \leftarrow ++$ ;
2   $cr \leftarrow$  initial credits;
3  for all  $l \in LOG_i$  do
4     $l.cr \leftarrow l.cr -$  used credits;
5    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     $L_w \leftarrow LOG_i$ ;
8    for all  $o \in L_w$  do
9      if  $s_j \notin o.Dests$  then  $o.Dests \leftarrow o.Dests \setminus x_h.replicas$ ;
10     else  $o.Dests \leftarrow o.Dests \setminus x_h.replicas \cup \{s_j\}$ ;
11   for all  $o_{z, clock_z} \in L_w$  do
12     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   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    $l.Dests \leftarrow l.Dests \setminus x_h.replicas$ ;
16 PURGE;
17  $LOG_i \leftarrow LOG_i \cup \{\langle i, clock_i, x_h.replicas \setminus \{s_i\}, cr \rangle\}$ ;
18 if  $x_h$  is locally replicated then
19    $x_h \leftarrow v$ ;
20    $Apply_i[i] \leftarrow ++$ ;
21    $LastWriteOn_i\langle h \rangle \leftarrow LOG_i$ ;

READ( $x_h$ ):
22 if  $x_h$  is not locally replicated then
23   RemoteFetch[ $f(x_h)$ ] from randomly selected site  $s_j$  that replicates  $x_h$  to get  $x_h$  and
24    $LastWriteOn_j\langle h \rangle$ ;
25   MERGE( $LOG_i, LastWriteOn_j\langle h \rangle$ );
26 else MERGE( $LOG_i, LastWriteOn_i\langle h \rangle$ );
27 PURGE;
28 return  $x_h$ ;

On receiving  $m(x_h, v, j, clock_j, x_h.replicas, c, L_w)$  from site  $s_j$ :
29 for all  $o_{z, clock_z} \in L_w$  do
30   if  $s_i \in o_{z, clock_z}.Dests$  then wait until  $clock_z \leq Apply_i[z]$ ;
31 for all  $o \in L_w$  do
32    $o.cr \leftarrow o.cr -$  used credits;
33   if  $o.cr \leq 0 \wedge o.Dests \neq \emptyset$  then delete  $o$ ;
34  $x_h \leftarrow v$ ;
35  $Apply_i[j] \leftarrow clock_j$ ;
36  $L_w \leftarrow L_w \cup \{\langle j, clock_j, x_h.replicas, c - \text{used credits} \rangle\}$ ;
37 for all  $o_{z, clock_z} \in L_w$  do
38    $o_{z, clock_z}.Dests \leftarrow o_{z, clock_z}.Dests \setminus \{s_i\}$ ;
39  $LastWriteOn_i\langle h \rangle \leftarrow L_w$ ;

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

```

Algorithm 6: Procedures used in Algorithm 5, 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      $\lfloor$  remove  $l_{z,t_z}$  from  $LOG_i$ ;

```

MERGE(LOG_i, L_w):

```

4 for all  $l \in LOG_i$  do
5    $\lfloor$   $l.cr := l.cr - \text{used credits}$ ;
6 for all  $o \in L_w$  do
7    $\lfloor$   $o.cr := o.cr - \text{used credits}$ ;
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     $\lfloor$   $l_{s,t'}.Dests := l_{s,t'}.Dests \cap o_{z,t}.Dests$ ;
14     $\lfloor$   $l_{s,t'}.cr := \min(l_{s,t'}.cr, o_{z,t}.cr)$ ;
15     $\lfloor$  delete  $o_{z,t}$  from  $L_w$ ;
16  $LOG_i := LOG_i \cup L_w$ ;
17 for all  $l \in LOG_i$  do
18    $\lfloor$  if  $l.cr \leq 0 \wedge l.Dests \neq \emptyset$  then delete  $l$ ;

```

cies 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.

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; 61). 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.

4.3 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 Figure 13 and Figure 14, 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 Figure 13, but not in Figure 14. 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 Figure 14. 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*.

4.4 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.

4.4.1 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 *credit*. *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.

4.4.2 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.

4.4.3 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 $5ms$ to $2005ms$. The propagation time T_t is from $100ms$ to $3000ms$. Consider the real-world network communication situation. T_t is initially given as

$5ms \sim 300ms$ and T_e is set to be $10ms \sim 200ms$. 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 $5ms$ through $2005ms$ and that of T_t from $100ms$ through $3000ms$. 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 – 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'.

4.4.4 Causal Consistency Verification

In this model, 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 5, 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) (Algorithm 5) 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 5) and line 18 (MERGE, Algorithm 6). 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 .

4.5 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%. The mean of numerical results performed from three runs is represented for each combination of the four parameters. Each simulation execution

TABLE VIII. 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

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

4.5.1 Violation Error Rate (R_e)

In Section 4.4.4, 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 Figure 15 ~ Figure 17. 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 VIII highlights the results for two types of critical initial credits (cr_c) when R_e is around 0.5% (exactly, 0.4% ~ 0.6%) and $R_e = 0$ (no causal consistency violation). When n is larger, the critical initial cr is basically also larger.

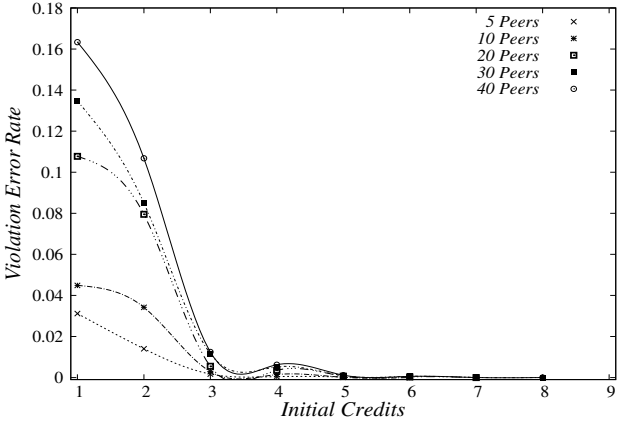


Figure 15. The Violation Error Rate for $w_{rate} = 0.2$.

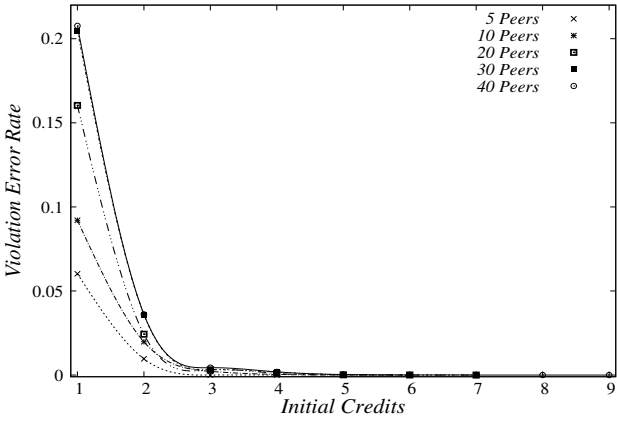


Figure 16. The Violation Error Rate for $w_{rate} = 0.5$.

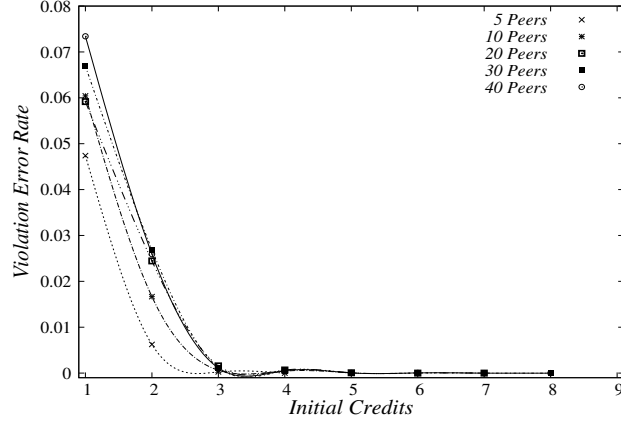


Figure 17. The Violation Error Rate for $w_{rate} = 0.8$.

TABLE IX. 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

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

Figure 18, Figure 19, and Figure 20 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 higher w_{rate} in more peers. Table IX lists the critical m_{ave} corresponding to the numerical data in Table VIII.

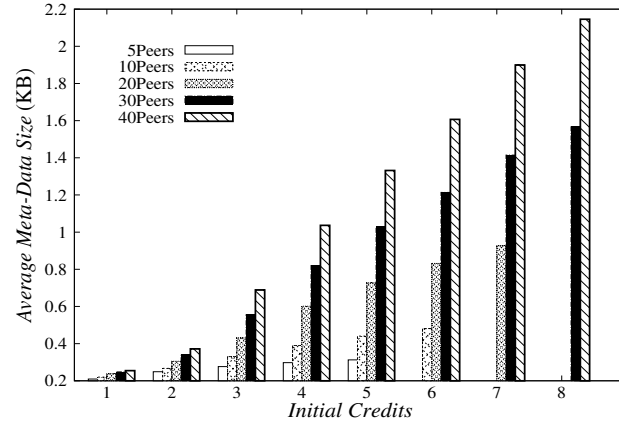


Figure 18. The Average Meta-Data Size (m_{ave}) for $w_{rate} = 0.2$.

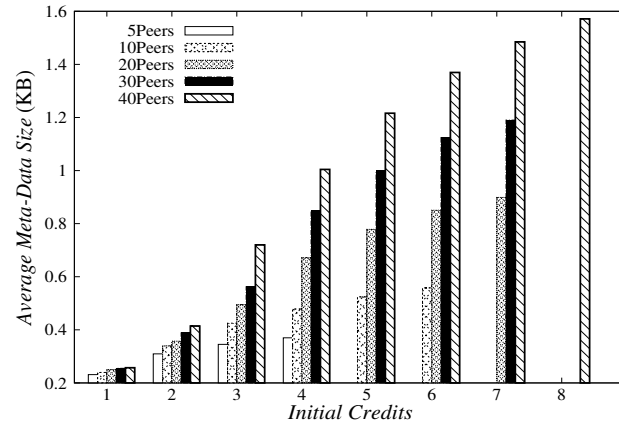


Figure 19. The Average Meta-Data Size (m_{ave}) for $w_{rate} = 0.5$.

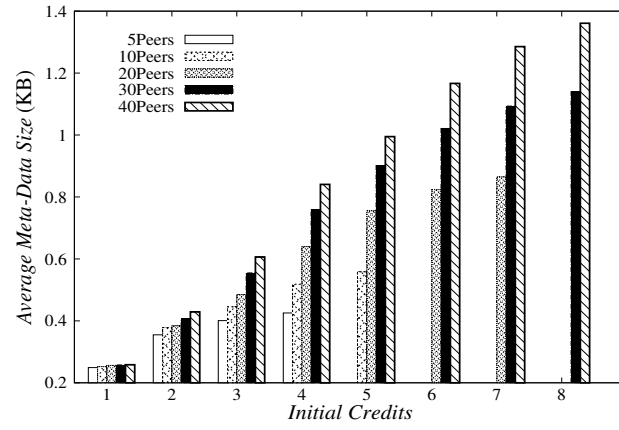


Figure 20. The Average Meta-Data Size (m_{ave}) for $w_{rate} = 0.8$.

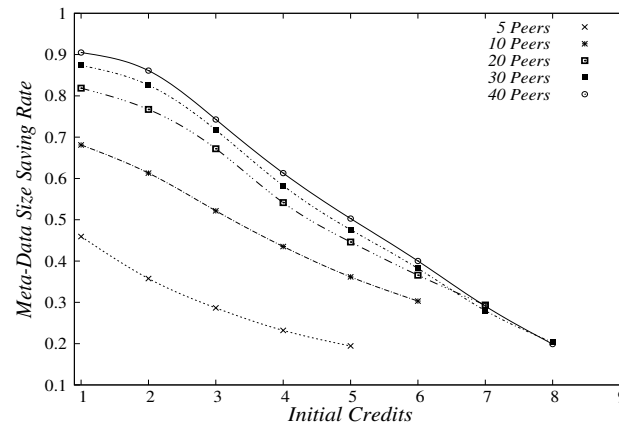


Figure 21. The Meta-Data Size Saving Rate (R_s) for $w_{rate} = 0.2$.

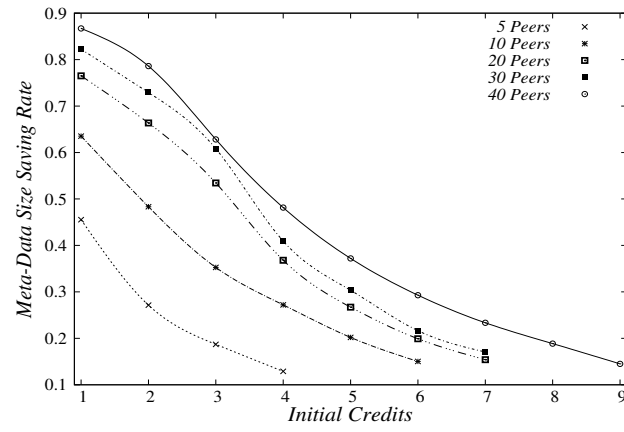


Figure 22. The Meta-Data Size Saving Rate (R_s) for $w_{rate} = 0.5$.

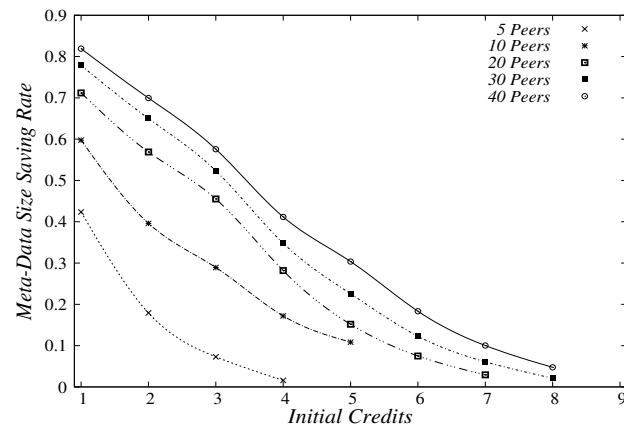


Figure 23. The Meta-Data Size Saving Rate (R_s) for $w_{rate} = 0.8$.

TABLE X. 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 XI. 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

4.5.3 Message Meta-Data Size Saving Rate (R_s)

Note that Approx-Opt-Track with $cr = \infty$ is equivalent to Opt-Track in (61). We define R_s as follows.

$$R_s = 1 - \frac{m_s(cr \neq \infty)}{m_s(\text{Opt} - \text{Track})} \quad (4.1)$$

Figure 21, Figure 22, and Figure 23 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 X lists the critical R_s corresponding to the numerical data in Table VIII. It can be seen that R_s is significantly negatively related to w_{rate} .

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

4.6.1 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 VIII 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.

4.6.2 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. Figure 15 ~ Figure 17 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 VIII summarizes the critical values of cr_0 and $cr_{\sim 0.5\%}$ from the results of R_e in Figure 15 ~ Figure 17.

4.6.3 Impact of initial cr on m_{ave}

Figure 18 ~ Figure 20, 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 Figure 18, 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 (61) 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 IX 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 0.48KB \sim 0.56KB for $R_e = 0$. For the case of 20 processes, $m_s(20)$ is around 0.86KB \sim 0.93KB for $R_e = 0$. For the case of 40 processes, $m_s(40)$ is around 1.36KB \sim 2.15KB 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.

4.6.4 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 4.1, R_s depends on m_s . R_s decreases as m_s increases, which positively depends on initial credits cr . Figure 21 \sim Figure 23 present the linear relationships between

R_s and initial cr for different w_{rate} . Consistent with the results in Figure 18 ~ Figure 20, 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 ~ 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 ~ 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 X 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% ~ 60% at a very slight cost of violation of causal consistency ($R_e \sim 0.5\%$). R_s reaches around 5% ~ 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 X reflect the effect of Approx-Opt-Track in the real world on causal consistency. Although it reduces 4.7% 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% in the lower and medium w_{rate} . From the above comprehensive analyses, it can be concluded that Approx-Opt-

TABLE XII. 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

Track provides a better network capacity utilization than Opt-Track without causing additional causal violations.

4.6.5 Impact of replica factor rate r_f on cr_c

This section illustrates the impact of different replica factor rates on cr_0 and $cr_{\sim 0.5\%}$. Table XI and Table XII 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 4.6.2. The lower the value of r_f , the fewer the write operations, which means that the read rate is higher. This makes

TABLE XIII. 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

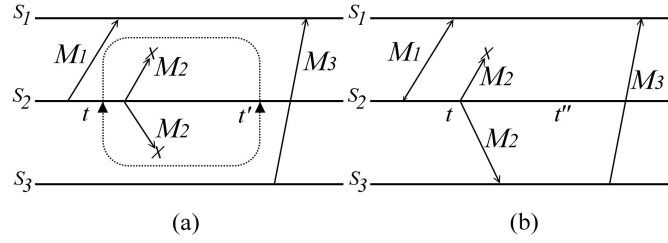


Figure 24. Examples of failures (a) partition (b) message loss.

R_e higher. It implies that it requires a slightly larger cr to maintain causal consistency with a lower r_f .

4.7 Discussions

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 Table XIII 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): Figure Figure 24 (a) 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). Figure Figure 24 (b) 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 Figure Figure 24 (a), 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 Figure Figure 24 (b), M_3 will be applied causally after receiving M_2 for the same reason as in Figure Figure 24 (a). 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 Figure Figure 24 (b), 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 Figure Figure 24 (b), 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. Figure 24 (b), 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.

CHAPTER 5

A PROACTIVE, COST-AWARE, OPTIMIZED DATA REPLICATION STRATEGY IN GEO-DISTRIBUTED CLOUD DATASTORES

This chapter is based on our previous publication (3). In this chapter, we focus on how to determine the suitable replica placement on-the-fly to increase the availability of data resources and maximize the system utilization in a time slot system.

The rest of this chapter is organized as follows. Section 5.1 presents the fundamental framework of our system cost model. Section 5.2 presents the details of CORP algorithm. Section 5.3 shows the experimental results in different settings and illustrates the experimental evaluation.

5.1 System Cost Model

5.1.1 Adaptive Cloud Data Provider Architecture

Our intention is to design a system architecture that can support modern social web storage services. The whole framework is a hierarchical geo-distributed cloud data store system composed of multiple geographically distributed store servers (see Figure 25). They are deployed in different zones dispersed across the world. (e.g., TX state is one zone). For simplicity, each zone employs only one store server (ZS). Multiple ZSs, which are connected with lower network access cost, constitute a Geo-Data Cluster (*GDC*). Multiple *GDC*s are fully connected by WANs with higher network access cost. Each cloud data object is replicated in a subset of

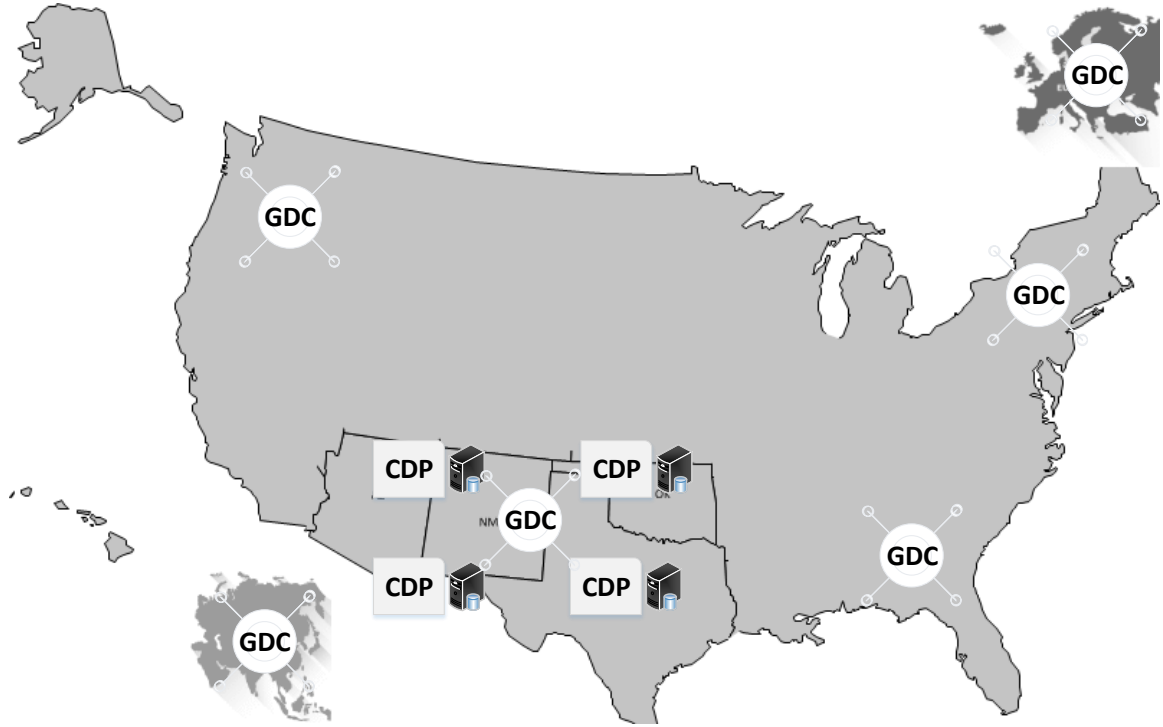


Figure 25. System architecture.

these ZSs. Since we focus on geo-replication, each zone hosts at most one replica (this may be straightforwardly extended as multiple replicas). Each ZS hosts one Cloud Data Provider (CDP) to manage cloud data replication for end-user access services. The top layer of CDP receives access requests, which are processed by the middleware layer. The system architecture model of CDP is schematized in Figure 26, where it presents the key components of our approach. This is patterned along (59; 69).

The target applications are social media applications, such as Instagram or Twitter. When a user creates a data object (e.g., a post, a tweet, or a photo) on his connecting ZS, this ZS and its binding *GDC* are referred to as the *master* ZS (MZS) and the *master GDC* (MGDC) of that data object, respectively. The MZS in the MGDC always stores this object until it is deleted by its creating user. A data object may be replicated across different zones in the same *GDC* or different *GDC*s. If a zone replicates an object to store it locally, this zone is called a *replica* zone of this object (the corresponding ZS is a *replica* ZS). Otherwise, it is a *nonreplica* zone. We denote a *replica* ZS h by $\delta[h] = 1$. Thus, if ZS h is a *nonreplica* ZS, $\delta[h] = 0$. When there exists at least one replica zone for an object in a *GDC*, this *GDC* is called a *replica GDC* of this object. For an object, the *master* zone and the *master GDC* are always a *replica* zone and a *replica GDC*, respectively. We also denote a *replica GDC* d by $\alpha[d] = 1$; for a *nonreplica GDC* d , $\alpha[d] = 0$.

User access requests are passed to the local ZS. If that ZS does not store a replica, it invokes a remote access request to retrieve the data object from other ZSs with replicas in the same *GDC* or different *GDC*s. Apparently, as the number of replicas of a data object increases, the data availability and utilization become higher; but the operational cost of maintaining and creating more replicas also increases. Furthermore, if the number and placement of replicas are inappropriate, the problem of poor QoS arises. Our approach is based on access request workload prediction. The major components of CDP, shown in Figure 26, are explained:

- *Workload-based Predictor (WP)* : Performs an estimation of future access demand to the local ZS for the target data object. The prediction approach utilizes the auto-

regressive integrated moving average model (ARIMA). For the MZS, the future demand estimate is directly passed to the local **CORP**, specified in the next step. For other ZSs, the demand estimate information will be transferred to the **CORP** of the MZS.

- *CORP Algorithm (CORP)* : We propose the **CORP** algorithm that performs cost optimization to determine the replica placement locations of the target data object based on the predicted future access demand from **WP**. **CORP** returns an optimal replica allocation pattern for the next time interval to minimize the total system cost (details in Section 5.2.1). For a target data object, **CORP** runs only in the corresponding MZS.
- *Cloud Data Provisioner* : 1) It receives access requests from *Access Control* and forwards them to the local ZS. 2) For the MZS of a target data object, it accepts an updated replica allocation pattern from **CORP** to figure out the migration process and to deploy suitable replicas on a periodic basis.

The function of **WP** designed in this paper realizes access workload prediction using the universal ARIMA time series forecasting approach (70) for the next time interval. ARIMA has been applied to the underlying workload fitting models in (71; 72), where the web-based workloads present strong autocorrelation.

At the end of each time interval, the data of historical workload traces during that time interval will be retrieved and transferred as the number of access requests by the Workload-based Predictor. The numbers of observed access requests for a sequence of time intervals compose a time series to fit the ARIMA model estimator. It is implemented as a cyclic shift operation, where the actual number of access requests at the last prediction cycle is added

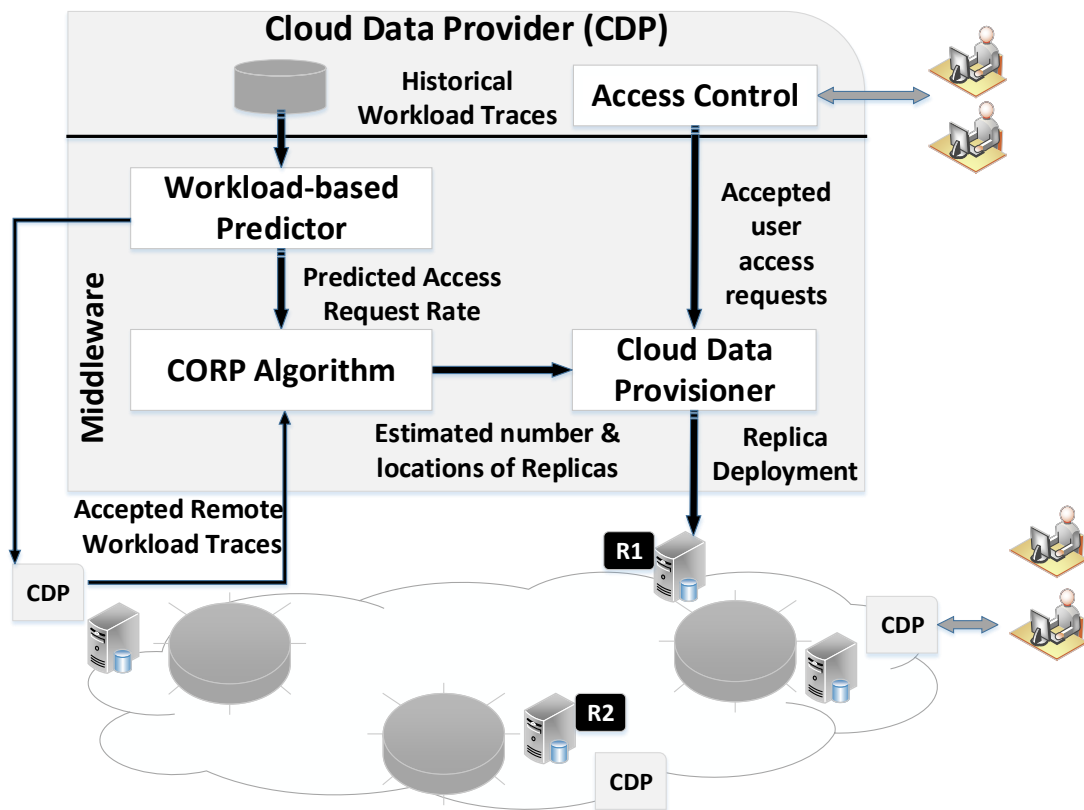


Figure 26. Architecture for adaptive cloud data provider.

to the time series while the oldest value is discarded. Once the access requested time series is completed, the ARIMA model-fitting process is performed with the Box-Jenkins approach (70). Based on this approach, the time series can be made to be stationary by differencing d times but remains nonstationary after differencing $d - 1$ times (called "integrated process").

Lags of the stationarized series in the forecasting model are called autoregressive terms. An autoregressive model (AR) is a linear regression of the current value of the series against one or more prior values of the series. The value of p lags in AR is called the order of the AR model. Lags of the forecasting errors are called moving-average terms. A moving average (MA) model is a linear regression of the current value of the series against the white noise or random shocks of one or more prior values of the series. The value of q lags in MA is called the order of the MA model. The Box-Jenkins ARIMA(p, d, q) model is a combination of the integrated AR and MA models. For selecting the suitable predictors in ARIMA, Akaike's Information Criterion (AIC) is used to determine the orders of p and q . AIC is defined as

$$AIC = -2\text{Log}(L) + 2(p + q + k + 1) \quad (5.1)$$

where L is the likelihood of the data, k may be one or zero ($k = 1$ if $c \neq 0$ and $k = 0$ if $c = 0$). The value of c is the average of the changes between consecutive observations. The corrected AIC with n parameters for ARIMA can be written as

$$AIC_c = AIC + \frac{2(p + q + k + 1)(p + q + k + 2)}{T - p - q - k - 2} \quad (5.2)$$

By minimizing AICc, p and q can be obtained. Using the above method to determine the orders of p , d , and q for the ARIMA model, the historical workload data is fit to the model to be used in WP, of which the output outcome is the predicted access requested rate for the next coming time interval. The length of the time interval can be flexibly adjusted for different applications. According to the evaluation in (73), an interval of 10 minutes could suit the selected cloud provider well.

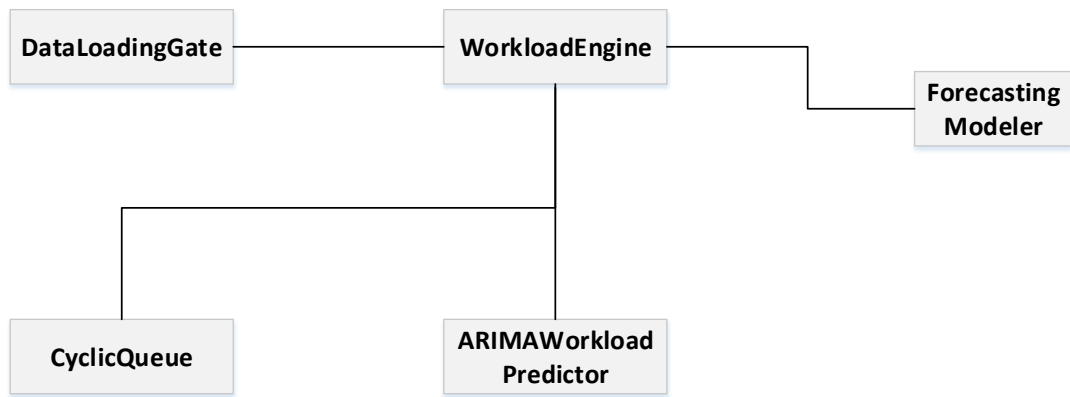


Figure 27. Diagram for ARIMA workload-based predictor.

5.1.2 ARIMA Configuration

Figure 27 presents the diagram of the WP function. The *WokloadEngine* is the key component in the WP. The *DataLoadingGate* is responsible for adding workload data into the *CyclicQueue* and to trigger the change of the most recent workload traces. The *CyclicQueue*,

which is composed of a queue object, is used to store the received workload data, which is modeled as a time series with a finite length. The length of the *CyclicQueue* is equal to the number of past time intervals influencing the prediction result. After *ARIMAWorkloadPredictor* creates a suitable ARIMA model, the workload prediction is accomplished by *ForecastingModeler*. *ARIMAWorkloadPredictor* and *ForecastingModeler* are realized by R packages (74), which is a statistical analysis program kit. They are not only capable of training a fitting model for a time series, but output a predicted value (the predicted number of access requests in the next time interval) fed back into the *WorkloadEngine*. The predicted value will be used in CORP algorithm we proposed to allocate replicas with the cost optimization. This procedure is discussed in the later subsections.

5.1.3 Prediction Complexity

Several approaches can be applied for ARIMA model fitting. The fitting process used in this paper is based on the Hyndman-Khandakar algorithm (74). It contains three major steps: (i) determine the order of differencing needed for stationarity; (ii) apply differentiation to the time series d times; (iii) decide the best fit model. The first step implements Kwiatkowski-Phillips-Schmidt-shin tests (75) used by R package. The time complexity of step (i) is $O(n^2)$ (where n is the number of training time series instances from the workload data). The second step requires $O(n)$ to perform d differentiations to the original time series values. The complete fitting process in the third step can be implemented with $O(kn^2)$ for k iterations. Because k is a finite value (independent of n), the total time complexity in (iii) can be counted as $O(n^2)$. Once the ARIMA model is ready, the prediction process for the next one time interval can

be accomplished with $O(\max(p, q))$, where p and q are the orders of AR and MA components, respectively.

5.2 Cost Optimization Replica Placement (CORP)

5.2.1 CORP Algorithm

We consider a time-slotted system, where the time is divided into slots with equal length time interval Δt and slot $k \in [1 \dots T]$ is referred to as the discrete time period $[k\Delta t, (k + 1)\Delta t]$. Let $GDC_i = \{h_1, h_2, \dots, h_{n_i}\}$ be a data storage cluster composed of n_i zones. Let $B = \{obj_1, obj_2, \dots, obj_l\}$ be the replicated data object set of a zone h . The costs incurred in time slot k include the following five components in each zone h . (i) storage cost S defines the cost of maintaining a data object replica per unit size per unit time. (ii) network cost $O_{gdc}(h)$ defines the data object transfer cost between different GDC s per unit size. (iii) network cost $O_{in}(h)$ defines the data object transfer cost within the same GDC per unit size. (iv) *Get* transaction cost t_g denotes the cost of issuing an object *Get* request. (v) *Put* transaction cost t_p denotes the cost of issuing an object *Put* request.

After creating an object z in time slot k , each zone maintains a detailed access record for this object. The access record is represented as 3-tuple $\langle GN_k[z][h], PN_k[z][h], V_k[z] \rangle$. These are the number of *Gets* and *Puts*, and the size for the requests for object z in zone h in time slot k in the system. Let $AN_k[z][h]$ be the sum of $GN_k[z][h]$ and $PN_k[z][h]$. $AN_k[z]$ means the

total access number of requesting object z in time slot k . $AN_k(z, d)$ denotes the total access request number in *GDC* d for an object in time slot k .

$$AN_k(z, d) = \sum_{\text{zone } h \text{ in } d} AN_k[z][h] \quad (5.3)$$

Let $R_k[z]$ be the replica factor for object z in time slot k (the replica factor represents how many replicas of object z will be made). It hinges on the object access request traces per time slot for each zone. In order to determine the optimal placement of deploying object replicas so as to minimize the overall system costs, we introduce the following cost functions for a replicated data object/ or a comment (z) created in zone h of *GDC* d with size of V in time slot k . For ease of notation and without loss of generality, we ignore the indices k and z in the following.

- *Replication Cost.* Since we consider modern social media services, there are two types of data – objects (e.g., a photo) and comment messages (e.g., a replying comment) – that should be replicated. Once a user creates an object or replies a comment, the system may need to replicate this object or comment to other zones in the same *GDC* d or in other *GDCs*. Thus, the replication cost contains two different network transfer costs. The first is the cost of replicating an object between different zones, which reside in different *GDCs*. We define $r[d]$ as the number of replica zones for an object z in *GDC* d .

$$r[d] = \sum_{\text{ZS } l \text{ in } d} \delta[l] \quad (5.4)$$

, where $\delta[l] = 1$ if a zone l is a *replica* zone; otherwise, $\delta[l] = 0$. Each ZS stores a Boolean vector $\delta[]$ for each object to track the registry of the object replicas distributed over the replica ZSs. By this bookkeeping implementation, the framework (each ZS) knows exactly which ZSs store which objects. The network transfer cost per unit data size for replicating object z from ZS h to these replica zones in *GDC* $d' \neq d$ is $\alpha[d'] (O_{gdc} + (r[d'] - 1) \times O_{in})$. Note that here if $r[d'] > 0$, $\alpha[d'] = 1$; otherwise, $\alpha[d'] = 0$. We define the **F**irst **N**etwork **T**ransfer **C**ost per unit data size as

$$FNTC = \sum_{d' \neq d} \alpha[d'] \{O_{gdc} + (r[d'] - 1) \times O_{in}\} \quad (5.5)$$

, where $\alpha[d'] = 1$ if *GDC* d' includes at least one *replica* zone; otherwise, $\alpha[d'] = 0$.

The second cost is the cost of replicating an object between zones within the same *GDC* d . Thus, the **S**econd **N**etwork **T**ransfer **C**ost per unit data size equals to

$$SNTC = (r[d] - 1) \times O_{in} \quad (5.6)$$

Since the data size of object z replicated is V , the total replication cost for creating this object (or replying to this comment) equals to

$$V \times (FNTC + SNTC) \quad (5.7)$$

- *Storage Cost.* The storage cost of an object z in zone h in time slot k with data size V equals to

$$SC(h) = S \times V \times \Delta t \times \delta[h] \quad (5.8)$$

- *Get Cost.* In our system, there are three different sub-cases for an object *Get* request, each of which corresponds to a *Get* cost function. Consider zone h issues $GN[h]$ *Get* requests for object z in time slot k . First, zone h stores a replica of object z . The *Get* cost equals to

$$GN[h] \times t_g \quad (5.9)$$

Second, zone h does not store a replica of object z , but some other ZSs store this object z in the same *GDC* d . The *Get* cost equals to

$$GN[h] \times (2t_g + V \times O_{in}) \quad (5.10)$$

Third, zone h does not store a replica of object z and no other zones store this object z in the same *GDC* d . The *Get* cost equals to

$$GN[h] \times (2t_g + V \times O_{gdc}) \quad (5.11)$$

- *Put Cost.* Similar to *Get* cost functions, there are also three *Put* cost functions, each of which corresponds to one of the above sub-cases. Consider zone h issues $PN[h]$ *Put* requests

for object z in time slot k , each of which needs to retrieve object z and replicate comment message M with size of m . First, ZS h stores a replica of object z . The *Put* cost equals to

$$PN[h] \times (t_p + m \times (FNTC + SNTC)) \quad (5.12)$$

Second, ZS h does not store a replica of object z , but some other ZSs store this object z in the same *GDC* d . The *Put* cost equals to

$$PN[h] \times (t_p + t_g + V \times O_{in} + m \times (FNTC + SNTC)) \quad (5.13)$$

Third, ZS h does not store a replica of object z and no other zones store this object z in the same *GDC* d . The *Put* cost equals to

$$PN[h] \times (t_p + t_g + V \times O_{gdc} + m \times FNTC) \quad (5.14)$$

Besides, when ZS h in *GDC* d is a *replica* ZS of object z , it leads to additional *Put* cost due to comment update messages from other remote ZS.

$$\begin{aligned} RUM = & \\ & (t_p + m \times (\beta[d'] \times O_{gdc} + \neg\beta[d'] \times O_{in})) \times \sum_{d' \neq d} \sum_{zs \ i \ in \ d'} PN[i] \\ & + (t_p + m \times O_{in}) \sum_{zs \ i \neq h \ in \ d} PN[i] \end{aligned} \quad (5.15)$$

, where $\beta[d'] = 0$, if $r[d'] \neq 1$. Otherwise, if $r[d'] = 1$, $\beta[d'] = 1$. Note that RUM is counted based on the point of view of incoming transmission. Therefore, RUM cannot be directly expressed by FNTC and SNTC.

- *Other Costs.* The cost of *Delete* to remove objects is free. *Post* transactions are considered as *Put* transactions. The *Post* cost is equal to the *Put* cost. *Copy* requests are implemented for replicating objects or object migration. A *Copy* request is composed of a *Get* transaction and a *Put* transaction. The cost of *Copy* equals to *Put* cost + *Get* cost.

Based on the above cost definitions, we can introduce three different cost functions as to an object z for a zone h in time slot k . First, zone h stores a replica of object z . The total zone cost for accessing the replica stored locally, defined as $SLC(h)$, is equal to

$$\begin{aligned} SLC(h) = & S \times V \times \Delta t + GN[h] \times t_g \\ & + PN[h] \times (t_p + m \times (FNTC + SNTC)) + RUM \end{aligned} \quad (5.16)$$

Second, zone h does not store a replica of object z , but some other ZSs store this object z in the same *GDC* d . The total zone cost for accessing a remote replica stored in a different ZS, defined as $RLC(h)$, is equal to

$$\begin{aligned} RLC(h) = & GN[h] \times (2t_g + V \times O_{in}) + PN[h] \times \\ & (t_p + t_g + V \times O_{in} + m \times (FNTC + SNTC)) \end{aligned} \quad (5.17)$$

Third, zone h does not store a replica of object z and no other zones store this object z in the same GDC d . The total zone cost for accessing a remote replica stored in a different GDC , defined as $RC(h)$, is equal to

$$RC(h) = GN[h] \times (2t_g + V \times O_{gdc}) + PN[h] \times (t_p + t_g + V \times O_{gdc} + m \times FNTC) \quad (5.18)$$

Therefore, the fundamental zone cost function $ZC_{z,k}$ of object z for ZS h in time slot k can be summed up with Storage Cost (SC), Transaction Cost (TC), and Network Transmission Cost (NTC). (For simplicity, we do not specify the subscripts z and k for SC, TC, NTC, and ZC.)

$$ZC(h) = SC(h) + TC(h) + NTC(h) \quad (5.19)$$

The transaction cost of ZS h can be represented as

$$TC(h) = GN[h] \times t_g + PN[h] \times t_p + \delta[h] \sum_d \sum_{zs} PN[i] \times t_p + \neg\delta[h] \times (GN[h] + PN[h]) \times t_g \quad (5.20)$$

The network transmission cost of ZS h can be represented as

$$\begin{aligned}
NTC(h) = & GN[h] \times (\neg\delta[h] \times V \times (\alpha[d] \times O_{in} + \neg\alpha[d] \times O_{gdc})) \\
& + PN[h] \times (\neg\delta[h] \times V \times (\alpha[d] \times O_{in} + \neg\alpha[d] \times O_{gdc})) \\
& + m \times (FNTC + \alpha[d] \times SNTC)) \\
& + (m \times (O_{in} + O_{gdc})) \times \sum_{d' \neq d} \sum_{zs \ i} PN[i] + (m \times O_{in}) \sum_{zs \ i \neq h \ in \ d} PN[i]
\end{aligned} \tag{5.21}$$

For object z in time slot k , if zone h stores a replica of the object (i.e., $\delta[h] = 1$), $SC(h)$ equals to $SLC(h)$ in (Equation 5.16). If h is not a *replica* zone but its binding *GDC* d is a *replica GDC* (i.e., $\delta[h] = 0$ and $\alpha[d] = 1$), $SC(h)$ equals to $RLC(h)$ in (Equation 5.17). If h 's binding *GDC* d is not a *replica GDC* (i.e., $\delta[h] = 0$ and $\alpha[d] = 0$), $SC(h)$ equals to $RC(h)$ in (Equation 5.18).

When the system assumes full replication, where all the data objects are replicated in all the zones, the total system cost for one object in a time slot equals to

$$\sum_d \sum_{zone \ h \ in \ d} SLC(h) \tag{5.22}$$

Although such full replication can reduce user access latency and maximize data availability, it is infeasible because of the immense size of the data stores and the large number of zones.

Consider the system under partial replication. Assume that there are r zones with replicas of an object z in time slot k . The total system cost TSC for object z in time slot k is equal to

$$TSC = \sum_d \sum_{\text{zone } h \text{ in } d} ZC(h) \quad (5.23)$$

, where $\sum_d \sum_{\text{zone } h \text{ in } d} \delta[h] = r$.

5.2.2 Cost Optimization Problem

Given the above system cost model, our goal is to determine the optimal distribution of replica placement for objects so as to minimize the overall TSC for each time slot. This problem is defined as follows.

$$\forall h, \delta[h]^{opt} \leftarrow_{\delta[h], \forall h} TSC \quad (5.24)$$

s.t. (repeated for $\forall k \in [1 \dots T]$)

(a) $\delta[h = master] \leftarrow 1$

(b) $\sum_d \sum_{\text{zone } h \text{ in } d} 1 = N$ (i.e., there are totally N zones in the system.)

In this cost optimization problem, GN , PN , V , and m for each object in a time slot are known and the argmin is over the set of $\delta[h]$ ($h=1, \dots, N$), for which TSC attains the minimum value. Since O_{gdc} is much higher than O_{in} , it is required to determine whether each GDC is a *replica GDC* in the system. If a GDC is not selected as a *replica GDC*, none of the zones of this GDC will store replicas of the object. On the other hand, if a GDC is assigned as a *replica GDC*, then, the system needs to determine whether each ZS h in this *replica GDC* is

required to store a *replica* of the object. Intuitively, this hinges on the comparison between $SLC(h)$ and $RLC(h)$, which is defined as

$$DIF(h) = SLC(h) - RLC(h) \quad (5.25)$$

Apparently, the value of $DIF(h)$ can specify whether it is optimal for a zone to store a *replica* of the object. However, when a zone is selected as a *replica* zone, the cost of replying comments to the associated object would increase. More precisely, $DIF(h)$ should be defined as

$$\begin{aligned} DIF(h) = & S \times V \times \Delta t - AN[h] \times (V \times O_{in} + t_g) \\ & + (m \times O_{in} + t_p) \times \sum_d \sum_{zone \ i \neq h} PN[i] \end{aligned} \quad (5.26)$$

Equation (Equation 5.26) reflects how a replica of the object in a zone affects the system cost. Without a replica, one zone will increase the network transfer cost $AN[h] \times V \times O_{in}$. With a replica, one zone will additionally bring the storage cost and the comment update cost. Formally, based on $DIF(h)$ in equation (Equation 5.26), one can determine whether zone h is required to store a replica for an object. In a *replica GDC*, if $DIF(h)$ is negative, zone h will be a *replica* zone. Otherwise, it is unnecessary to store a replica of the object in zone h . Similarly, we define a global $DIF_g(d)$ in the following equation to determine whether a *GDC* d is a *replica GDC* by comparing the cost with one replica to that without any replica. In addition to the cost of transferring update comments within a *GDC*, $DIF_g(d)$ considers that of updating comments between *GDCs*.

$$\begin{aligned}
DIF_g(d) = S \times V \times \Delta t - AN(d) \times (V \times O_{gdc} + t_g) + \\
(m \times O_{gdc} + t_p) \times \sum_{d' \neq d} \sum_{\text{zone } i \text{ in } d'} PN[i] + \\
(m \times O_{in} + t_p) \times \sum_{\text{zone } i \neq h \text{ in } d} PN[i]
\end{aligned} \tag{5.27}$$

In equation (Equation 5.27), the zone h having the maximum PN in GDC d is selected as a *replica* zone. Similar to $DIF(h)$, if $DIF_g(d)$ is negative, GDC d will be a *replica GDC*. Otherwise, it is not required to store a replica of the object in GDC d in order to lower the system cost.

On the above grounds, we propose Cost Optimization Replica Placement Algorithm (**CORP**), as Algorithm 7, which calculates the optimized cost of the object replicas in each time slot $k \in [1 \dots T]$. The replication strategy of **CORP** (run at the end of time slot $k - 1$) is summarized as follows. First, determine the replica placement in the *master GDC*. The *master* ZS must be a *replica* ZS. Then, one can determine whether each of the other zones in the *master GDC* is required to store a replica of object z . Second, the system needs to determine whether each of the other GDC s is a *replica GDC*. Then, the system will determine whether each zone in every *replica GDC* is a *replica* zone based on equation (Equation 5.26). We make some notes about this instantiation of **CORP**. Lines (2)-(3) specify the replica placement for the object in the *master GDC*. Lines (4)-(21) determine the replica placement for the object in the other GDC s. When GDC d is a *replica GDC* in time slot $k - 1$ and it is determined that d is still required to be a *replica GDC* in time slot k , lines (8)-(10) will specify the replica placement

Algorithm 7: CORP Algorithm based on the functions $DIF(h)$ and $DIF_g(d)$

Input : $\forall h, \delta_{k-1}[h], GN_k[h], PN_k[h]$, master ZS H_h , and master GDC H_d
Output: $\forall h, \delta_k[h]$

```

1 Initialize:  $\delta_k \leftarrow \delta_{k-1}$ ;
2 for each zone  $i \in H_d \wedge i \neq H_h$  do
3    $\sqsubset$   $SetRep(i)$ ;
4 for each GDC  $d \neq H_d$  do
5   Select  $h$  with the maximum  $AN_k[h]$  in  $d$ ;
6   Calculate  $DIF_g(d)$ ;
7   if  $DIF_g(d) < 0 \wedge IsReplicaGDC(k-1, d)$  then
8      $\delta_k[h] = 1$ ;
9     for each host  $i \neq h \in d$  do
10       $\sqsubset$   $SetRep(i)$ ;
11   else if  $DIF_g(d) < 0 \wedge \neg IsReplicaGDC(k-1, d)$  then
12     Migrate a new replica to site host  $h$  from another replica GDC;
13      $\delta_k[h] = 1$ ;
14     for each host  $i \neq h \in d$  do
15       $\sqsubset$   $SetRep(i)$ ;
16   else if  $DIF_g(d) > 0 \wedge IsReplicaGDC(k-1, d)$  then
17     for each host  $i \in d$  do
18        $\sqsubset$   $\delta_k[i] = 0$ 
19     if  $DIF_g(d) - O_{gdc} \times V < 0$  then
20       Select  $h$  with the maximum  $AN_k[h]$  in  $d$ ;
21        $\delta_k[h] = 1$ 
22    $\sqsubset$   $SetRep(i)$ 
23 Calculate  $DIF(i)$ ;
24 if  $DIF(i) > 0 \wedge \delta_{k-1}[i] = 1$  then  $\delta_k[i] = 0$ ;
25 if  $DIF(i) < 0 \wedge \delta_{k-1}[i] = 0$ 
     $\wedge DIF(i) + V \times O_{in} < 0$  then  $\delta_k[i] = 1$  ;
     $AN(k, d)$ :
26 return  $\sum_{host\ i\ in\ d} (GN_k[i] + PN_k[i])$ 
     $IsReplicaGDC(k, d)$ 
27 if  $\exists host\ i \in d: \delta_k[i]$  is true then
     $\sqsubset$  return true
28 else
     $\sqsubset$  return false

```

TABLE XIV. Definition of symbols and parameters used in the model.

Term	Meaning
D	A set of <i>GDCs</i>
S	The storage cost per unit size per unit time
$V(z)$	Size of data item z
Δt	Time slot interval
O_{gdc}	Out-network price between <i>GDCs</i>
O_{in}	Out-network price within a <i>GDC</i>
$GN_k[z][h]$	Number of <i>Gets</i> for object z from zone h in time slot k
$PN_k[z][h]$	Number of <i>Puts</i> for object z from zone h in time slot k
$V_k[z]$	Size of object z in time slot k
$AN_k[z][h]$	The sum of $GN_k[z][h]$ and $PN_k[z][h]$
$R_k[z]$	The number of replicas of object z in time slot k
t_g	<i>Get</i> transaction cost
t_p	<i>Put</i> transaction cost
m	Comment message size
$\delta[h]$	Binary replication factor for zone h
$\alpha[d]$	Binary replication factor for <i>GDC</i> d
$r[d]$	Number of replicas for an object in <i>GDC</i> d
$\beta[d]$	Binary factor for <i>GDC</i> d . If $r > 1$, $\beta=0$; if $r = 1$, $\beta=1$.

in *GDC* d . Lines (12)-(15) specify the case almost similar to the above one. However, *GDC* d is not a *replica GDC* in time slot $k - 1$. It needs to migrate a new replica from another *GDC*. The output of **CORP** is the replica placement distribution $\delta_k[h]$ for each ZS h . When there exists at least one ZS h such that $\delta_k[h] \neq \delta_{k-1}[h]$, *Cloud Data Provisioner* in the MZS would implement the migration process. Otherwise, no replica needs to be migrated or deleted. Table XX summarizes parameters and inputs to the model.

5.2.3 CORP + cache.

The regular CORP runs the ARIMA migration process by an equal time interval. Caching is commonly used to decrease network traffic and reduce network link utilization. In addition, for a target data object z , there is only home ZS that stores it in the first time interval, during which there are much more access requests from other ZSs over network. This leads to ineffective network utilization. We extend CORP to CORP + cache, which could save object z temporarily in ZS h after retrieving it from a *replica* ZS of object z . This replica ZS is called the source ZS of the client cache ZS h . As with the replica bookkeeping mechanism in CORP, a source ZS hosts a Boolean vector to record its own client cache ZSs in + cache. Thus, source ZSs for an object can know exactly where cache ZSs are. Accordingly, ZS h does not need to retrieve it each time an access request is issued, even if ZS h is not a *replica* ZS. ZS h is called a *caching* ZS for object z . When a *replica* zone issues an update to object z , the update request is applied to not only all *replica* ZSs, but also the ZSs with the cache data of object z . However, the caching data of object z in ZS h can only be accessed by users connecting to ZS h . It cannot be retrieved by other ZSs. CORP + cache runs the same algorithm as CORP. However, the migration process of CORP + cache differs from that of CORP. Assume that ZS h is a *caching* ZS for object z in *GDC* d in time slot $k - 1$. If ZS h is determined as a *replica* ZS, ZS h can straightforwardly become a *replica* zone. Then, if there exists other ZSs in *GDC* d that do not store a replica of object z in time slot $k - 1$ and these ZSs are determined as *replica* ZSs in time slot k , ZS h will replicate object z to these ZSs. On the other hand, assume that ZS h caching object z in time slot $k - 1$ does not need to be a *replica* ZS in time slot k .

If there is no other *replica* ZS in time slot $k - 1$ in the same GDC and some ZSs in the same GDC, which do not store object z , become *replica* ZS, then, ZS h needs to replicate object z to them. After that, the caching data of object z in ZS h is deleted.

5.3 Performance Evaluation

5.3.1 Experimental Setting

We evaluate the proposed **CORP** algorithms for replica placement of the data objects across *GDCs* with real traces of requests to the zone web servers from Twitter workload (76) and the CloudSim discrete event simulator (77). These realistic traces contain a mixture of temporal and spatial information for each http request. The number of http requests received for each of the target data objects (e.g., photo images) is aggregated in 1000-secs intervals. By implementing our approaches on the Amazon cloud provider, it allows us to evaluate the cost-effectiveness of request transaction, data store, and network transmission, and to explore the impact of workload characteristics. We also propose a clairvoyant Optimal Placement Solution, based on the time slot system and object access patterns known in advance to evaluate CORP and CORP+cache.

Data Object Workload: Our work focuses on the data store framework on image-based sharing in social media networks, where applications have geographically dispersed users who put and get data, and fit straightforwardly into a key-value model. We use actual Twitter traces as a representation of the real world. *Put* to a timeline occurs when users post a tweet, retweet, or reply messages. We crawl the real Twitter traces as the evaluation input data. Since the Twitter traces do not contain information of reading the tweets (i.e., the records of *Gets*),

we set five different ratios of *Put/Get* (P_{rate} : *Put* rate), where the patterns of *Gets* on the workloads follow Longtail distribution model (78). The simulation workload contains several Tweet objects. The volume of each target tweet in the workload is 2 MB. The simulation is performed for a period of 20 days. The results for each objects show that they have similar tendency.

The experiment has been performed via simulation using the CloudSim toolkit (77) to evaluate the proposed system. CloudSim is a JAVA-based toolkit that contains a discrete event simulator and classes that allow users to model distributed cloud environments, from providers and their system resources (e.g., physical machines and networking) to customers and access requests. CloudSim can be easily developed by extending the classes, with customized changes to the CloudSim core. We figure out our own classes for simulation of the proposed framework and model 9 *GDCs* in CloudSim simulator. Each *GDC* is composed of 4 zones. Each zone has only one *ZS* associated with 50GB storage space and corresponds to one (or a few) states in US or one country in Asia and in Europe. The price of the storage classes and network services are set for each *GDC* and each *ZS* based on Amazon Web Service (AWS) as of 2018.

5.3.2 Results and Discussion

The performance metrics we use are the performance in terms of cost and the cost improvement rates under varying P_{rate} of the proposed CORP and CORP+cache. In order to evaluate our proposed approach, we compare it to two different replication strategies. The first one is the standalone cache mode (+cache), where the home *ZS* is the only one *replica* *ZS*. The second one is the replication model with different numbers of *replica* *GDC*, where they are

randomly pre-selected and each GDC includes at most one *replica* ZS. Cost is represented by the total system cost, which is composed of TC (Equation 5.20), NTC (Equation 5.21), and SC (Equation 5.19).

First, we use the term ‘transaction’ to denote both a *Put* transaction or a *Get* query transaction. Active and aggressive replication has the potential to provide a reduction in the number of distributed *Get* transactions at the cost of *Put* transactions. Lowering the number of transactions to find a data placement increases throughput significantly in cloud environments, while an increased number of transactions would lead to an over-utilization of the underlying systems. Thus, the total transaction cost (TC) is totally subject to the number of transactions. In our evaluations, we set different *Put* rates to generate different evaluation workloads. The lower the *Put* rate, the more the number of *Get* transactions should be included (i.e., the total number of transactions increases). Figure 28 presents the TCs of various replication strategies in different *Put* rates. Since CORP brings additional migration process, TC for CORP+cache is slightly higher than the standalone cache model or 2 pre-selected replicas+cache but much lower than others’ TCs.

Second, Figure 29 presents the NTC of CORP+cache in comparison with various replication strategies in different put rates. According to Eq. (Equation 5.21), NTC highly depends on the amount of data transmitted over the network. Thus, the smaller the NTC, the lower the network bandwidth consumption. Although NTC of CORP+cache is slightly higher than that of the full GDC replication, it is much lower than others’ NTCs.

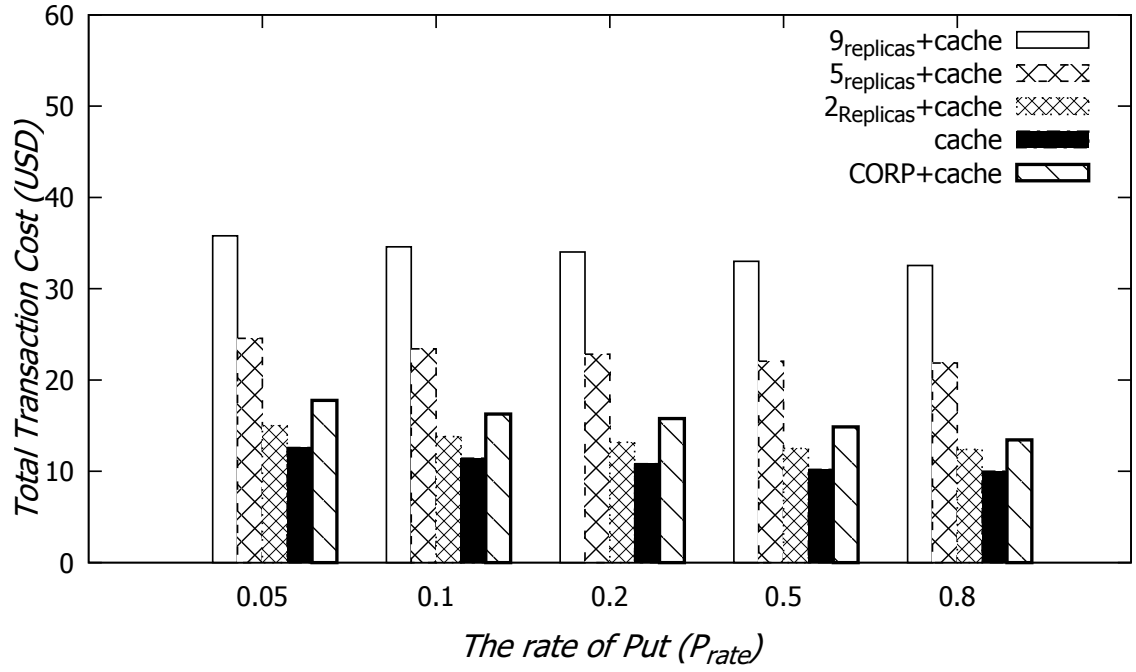


Figure 28. The Transaction Cost

Finally, the capacity of replica servers has a major impact on the performance of data replication strategy. Different data objects or applications require different storage services. Storage providers have to place large quantities of storage devices in order to offer uses good data storage services. Therefore, maximizing storage space utilization (SSU: the total available storage space in the entire system) becomes more and more important. The larger the available storage space, the more data objects the system can hold. Lower storage space occupation (SSO: the space size occupied by data objects) for a data object would increase SSU. The total

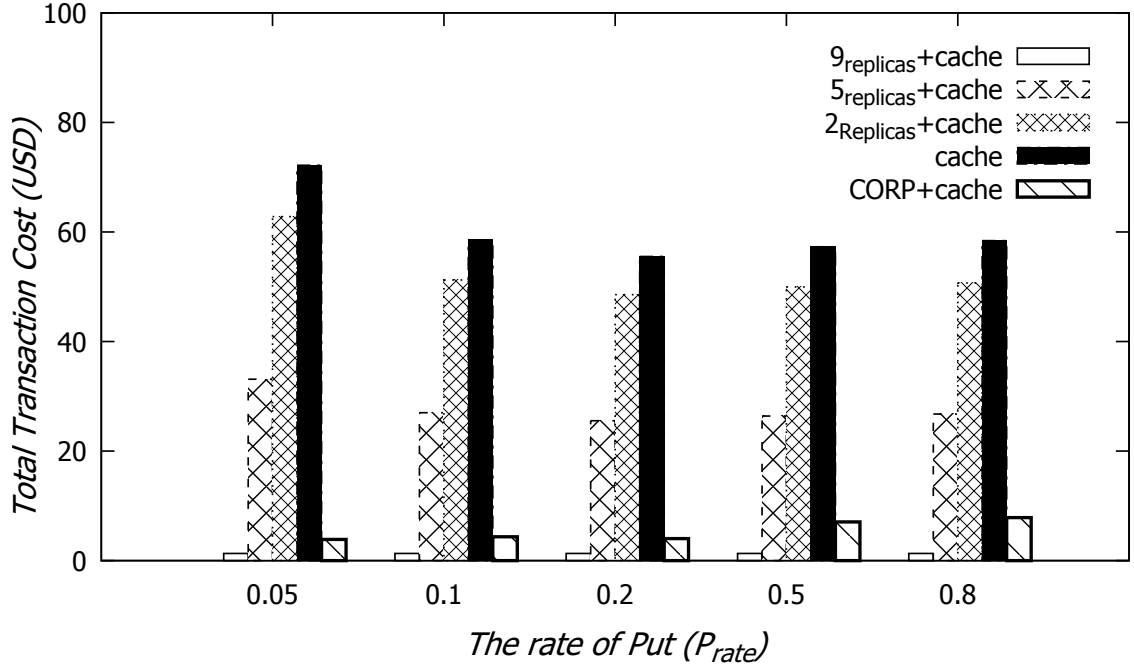


Figure 29. The Network Transmission Cost

storage space cost (SSC), which fully depends on Equation 5.8, is highly proportional to SSO. Figure 30 shows the results of storage cost (SC) of CORP+cache in comparison with other alternatives. We notice that the SC of CORP+cache falls in between the SC of full GDC replication and the SC of standalone cache mode. This implies that CORP+cache is able to determine the proper number of replicas to decrease TC and NTC.

Figure 31 presents the total system costs (TSC) for different replication approaches. With caching, the more the number of *replica* GDCs, the lower the TSC. CORP+cache can further

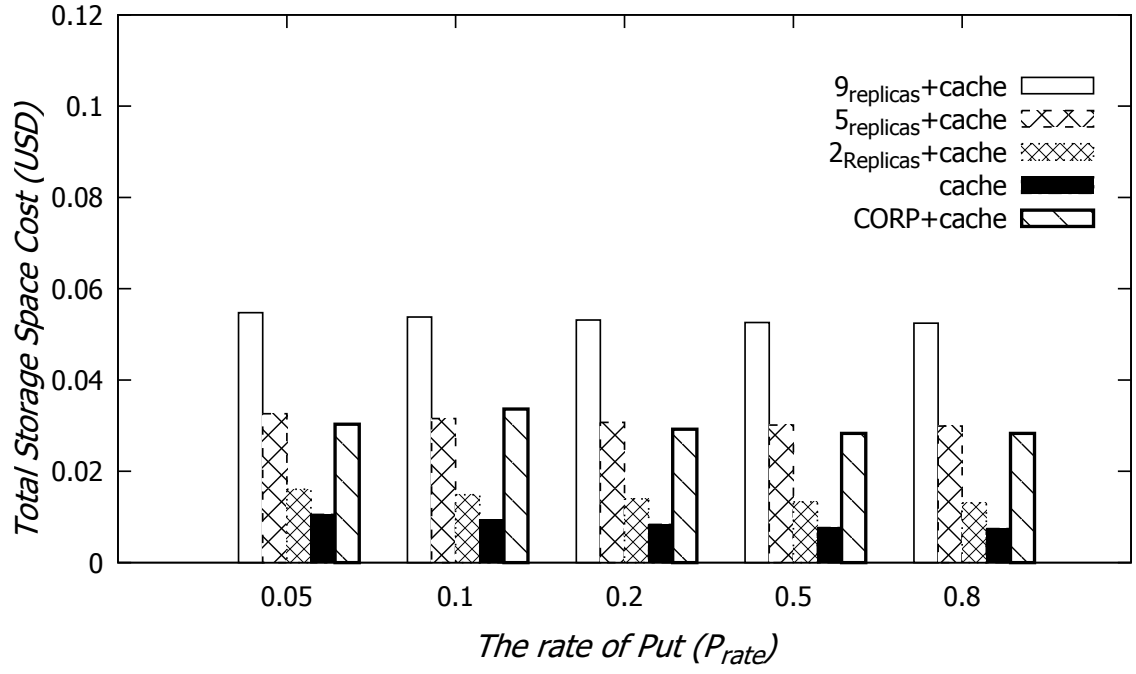


Figure 30. The Storage Space Cost

reduce TSC for all the workloads. Although CORP brings more migration costs for replica placement step, it can reduce the storage cost and network cost by proactively placing data objects at proper locations.

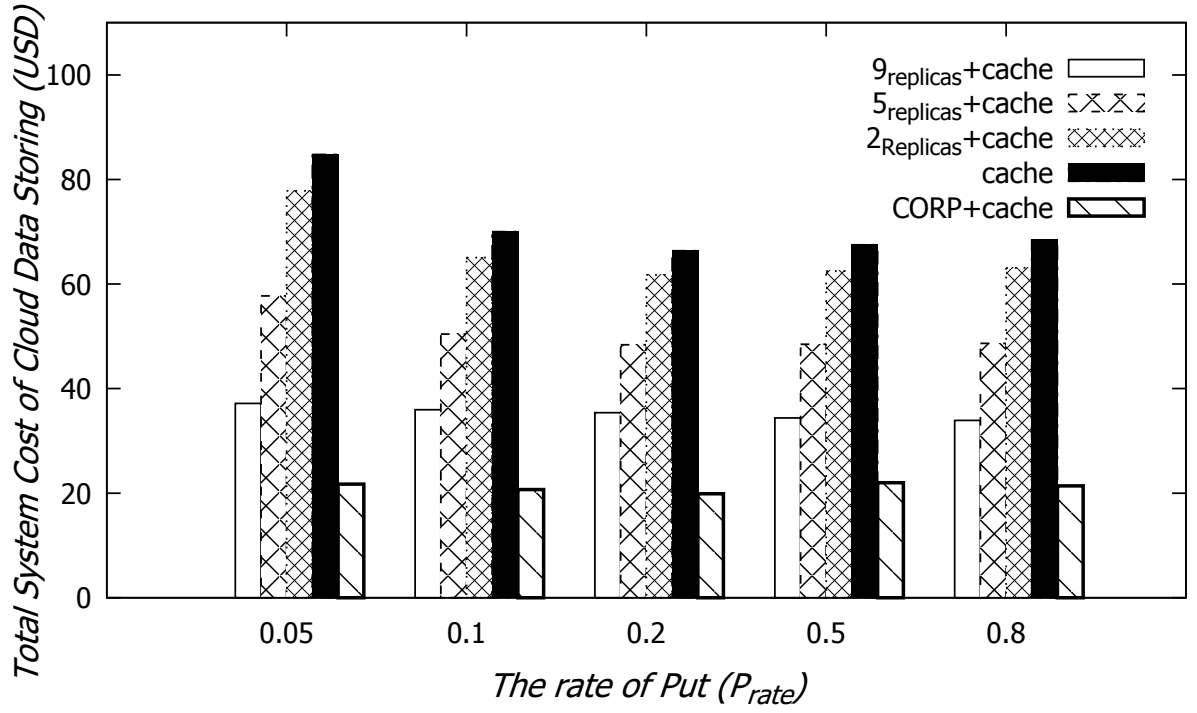


Figure 31. The total system cost.

5.3.2.0.1 Cost Improvement

We now investigate the effect of CORP+cache on cost improvement rate with respect to different replication modes, which is defined as.

$$\frac{cost(+cache) - CostofReplicationMode}{cost(+cache)} \quad (5.28)$$

TABLE XV. Cost improvement rates of different replication modes with respect to the standalone cache mode for different Put rates. (a) 2 static replicas with cache control (b) 5 static replicas with cache control (c) 9 static replicas with cache control (i.e., full GDC replication) (d) CORP+cache.

P_{rate}	0.05	0.1	0.2	0.5	0.8
(a)	8.24%	7.10%	6.96%	7.44%	7.81%
(b)	31.94%	27.96%	27.11%	28.17%	29.93%
(c)	56.16%	48.64%	46.70%	49.08%	50.44%
(d)	74.37%	70.46%	70.03%	67.39%	68.75%

, where $\text{cost}(+cache)$ means the total system cost of the standalone cache model. Table XV shows the cost improvement rates of different replication modes with respect to the +cache mode for different Put rates. The cost improvement rate against +cache increases as more *replica* GDCs are loaded. As Put rate decreases, the cost improvement of CORP+cache becomes higher. It shows that CORP + cache is cost-effective for Get-intensive (lower Put rates) workloads like most social networks or cloud computing environments, due to efficient utilization of network communication resources.

5.3.2.0.2 Accuracy Analysis

The proposed method, ordinary CORP, runs on a time slot system. At runtime, CORP is constantly updated. When new access requests arrive in the current time slot, they are getting involved into the time series and the data in the oldest time slot is removed from the time series. However, when a data object is created, there is not enough data in the time series initially (i.e., the number of time slots is zero initially for each data object). Based on the training dataset, the demand for each time slot per ZS is predicted. The output of the prediction model

TABLE XVI. Access number prediction accuracy by various error metrics. RMSD : root mean square deviation. NRMSD : normalized root mean square deviation. MAD : mean absolute deviation. MAPE : mean absolute percentage error

Accuracy metric	Predicted	Low 80%	High 80%	Low 95%	High 95%
RMSD	346.71	476.12	571.42	800.01	931.14
NRMSD	0.16	0.31	0.33	0.37	0.41
MAD	131.4	379.81	450.11	534.66	794.52
MAPE	0.13	0.15	0.19	0.18	0.24

is an integer value, which represents the access requests in the next time slot at a ZS. The predicted value is accompanied with two confidence ranges across the 80 and 95 percent bands. The accuracy of the prediction model is evaluated by different types of error metrics, namely, root mean square deviation (RMSD), normalized root mean square deviation (NRMSD), mean absolute deviation (MAD), and mean absolute percentage error (MAPE). The low 80% and high 80% indicate the limits for the 80% confidence interval for the prediction. The results for the 80% and 95% confidence intervals are given in Table XVI with $P_{rate} = 0.05$, where the average prediction accuracy (APA) is around 87 percent. With different Put rates, the corresponding APA values are approximately consistent with that of $P_{rate} = 0.05$.

5.3.2.0.3 CORP+cache VS. CORP

We propose CORP+cache to improve the system costs. Thus, in this section we present experiments aimed at evaluating how the total costs are improved by CORP+cache against

TABLE XVII. Δ_{saving} : The cost improvement results for different Put rates show that caching has taken an important step to improve the total system costs.

P_{rate}	0.05	0.1	0.2	0.5	0.8
Δ_{saving}	91.33%	84.67%	74.93%	57.18%	48.70%

CORP. Table XXII presents the results of comparing CORP+cache to CORP for different put rates. Δ_{saving} is defined as

$$\frac{cost(CORP) - cost(CORP + cache)}{cost(CORP)} \quad (5.29)$$

Since the evaluation data come from the social network, each individual data object brings a lot of requests in the initial time slots. It can be observed that the results indicate that the lower the P_{rate} (Get-intensive), the better the cost saving rate (Δ_{saving}) is.

5.3.2.0.4 CORP+cache evaluation

In order to evaluate the effectiveness of our proposed approach, we also implemented the Optimal Placement Solution (OPT) based on the same time slot system. OPT knows the exact temporal and spatial data object access patterns. OPT can figure out the optimal object placement for each time slot. The cost effectiveness of CORP(+cache) is highly related to how precise the predicted access number is. OPT(+cache) totally follows up CORP(+cache) model and uses the real object access (Put and Get) numbers as the inputs in different time slots.

Δ_{inc} is defined as

$$\frac{cost(CORP + cache) - cost(OPT + cache)}{cost(CORP + cache)} \quad (5.30)$$

TABLE XVIII. Δ_{inc} : The performance evaluation of CORP+cache compared to OPT+cache.

P_{rate}	0.05	0.1	0.2	0.5	0.8
Δ_{inc}	16.77%	13.84%	10.74%	9.66%	6.23%

TABLE XIX. $\Delta_{inc'}$: The performance evaluation of CORP compared to OPT in steady states.

P_{rate}	0.05	0.1	0.2	0.5	0.8
$\Delta_{inc'}$	1.89%	2.83%	1.53%	4.95%	3.26%

Table XVIII shows the comparisons between CORP+cache and OPT+cache for different Put rates. It is evident that CORP+cache only increases 6% \sim 16% of total system cost compared to OPT+cache.

In order to measure the cost effectiveness of CORP in steady states (including enough training time slots), we also compare the cost of CORP to that of OPT without caching in steady states. Table XIX presents the cost increase rates ($\Delta_{inc'}$) of CORP compared to OPT for different Put rates. We notice that $\Delta_{inc'}$ rates are around 2% \sim 5%. $\Delta_{inc'}$ is defined as

$$\frac{cost(CORP) - cost(OPT)}{cost(CORP)} \quad (5.31)$$

For simplicity, we ignore the cost of the bookkeeping process based on the following. The size of a Boolean vector δ is $O(N)$ bits, where N is the total number of zones in the system. The numbers of availability zones in most current modern cloud storage systems are from

several dozens to several hundreds (i.e., AWS is composed of 69 zones globally). Compared to a common data object's size or a text comment's size in most social media applications, the storage cost of keeping δ is much smaller than them. In addition, *Get* bucket is not supported in CORP (+cache) to retrieve multiple objects by one access request. Therefore, we do not consider the cost of *List* requests.

CHAPTER 6

CADROP: COST OPTIMIZED CONVERGENT CAUSAL CONSISTENCY IN SOCIAL NETWORK SYSTEMS

In this chapter, we formally develop a cost-effective protocol that ensures causal+ consistency in a partially geo-replicated data storage on social network platforms, where the causal+ order would be maintained for all replying comments corresponding to a post with a unique key or for different objects with explicit causal ordering. Consider the cost advantage of partial replication and the potential of dynamic replication in time-varying environments. We extend the ideas in Chapter 3 and Chapter 5 to a client-server architecture. The rest of this chapter is organized as follows. Section 6.1 presents the fundamental framework of our system cost model and the system definitions. Section 6.2 presents the details of the system design. Section 6.3 illustrates the overall algorithms. Section 6.4 shows the experimental results in different settings and illustrates the experimental evaluation.

6.1 Definitions and System Model

6.1.1 Causal Consistency

A causal consistency system requires that clients observe the results returned from the data repository servers, consistent with the causality order. Causality is the happen-before relationship between two events(15; 18). The two events must be visible to all clients in the same order, when they are causally related. In other words, when users in client A observe

that event M1 happens before M2, other users in client B can perceive that the effects of M1 occurring are visible to M2. Otherwise, a (potential) causality violation has occurred. When a series of access operations occur on a single thread, they are serialized as a local history h . The set of local histories from all threads form the global history H . For potential causality (15), if there are two operations o_1 and o_2 in O_H , we say that o_1 causally depends on o_2 , denoted as $o_1 \prec_{co} o_2$, if and only if one of the following conditions holds:

1. o_1 precedes another local operation o_2 in a single thread of execution (program order).
2. o_1 is a *write* operation and o_2 is a *read* operation that returns a value written by o_1 , even if o_1 and o_2 are performed at distinct threads (read-from order).
3. there is some other operation o_3 in O_H such that $o_1 \prec_{co} o_3$ and $o_3 \prec_{co} o_2$ (transitive closure).

Especially, the causality order defines a strict partial order on the set of operations O_H . For a causal consistency system, all the write operations that can be related by the potential causality have to be observed by each thread in the order defined by the causality order.

6.2 System Design

CaDRoP runs in a distributed key-[values] data store that manages a set of data objects in social networks. [values] is a list of values corresponding to an item key. In our system, one post, such as a picture on Instagram, is viewed as an object item and is assigned a global unique number as the item key. A post object is always saved in the head of [values] as v_0 . Afterwards, when a comment (e.g., a list of strings) is posted out under a post, this comment text, denoted

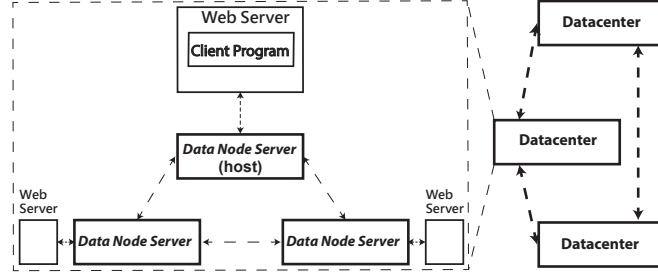


Figure 32. The system architecture.

as v_i ($i > 0$; i is the index of [values]), will be inserted to [values]. Thus, CaDRoP implements a multiversion data store system and treats the value of each update operation as an immutable version of the access object. When users request access to a data object, [values] (i.e., a list of version values) is the result returned. Each entry in [values] corresponds to one update operation. In order to track causality, each version value needs to be associated with some metadata. [values] is also a causal list. For example, consider two entries v_i and v_j in [values] and $i < j$. Assume that v_i and v_j are created by update operations o_a and o_b , respectively. CaDRoP can guarantee that $o_b \not\prec_{co} o_a$. Although the potential causality allows to prevent any causal anomalies, it increases higher costs to maintain many dependencies among different posts without any semantic coherency in social networks. For example, there is a cute dog photo posted in the morning and a blue sky image uploaded at noon. Tracking explicit causal order offers a more flexible solution. Under explicit causality, each application can have its own happens-before relationships between operations (26). Because it tracks only customized relevant dependencies, explicit causality decreases the number of dependencies per modification

and lowers metadata overhead. We have modeled a hybrid causality based on a column-based model. Our system maintains two types of columns:

- key columns: they are used to store data item keys.
- value columns: each value column contain a [values] corresponding to a data item key.

CaDRoP supports the explicit causality in key columns and implements the potential causality for each value column. Explicit causality can be captured through application user interface. For example, user Bob can click @ symbol on Facebook to post an image content to reply a post done by user Alice before. Thus the client program can capture the causal dependency between the two posts, even if they are realized by different users. Otherwise, the causal relationship between different object keys will be ignored in CaDRoP. The whole framework is a hierarchical geo-distributed cloud store system composed of multiple geographical *DCs* (see Fig. Figure 32). All the *DCs* are fully connected by WANs with higher network access cost. They are deployed and dispersed across the world. In each *DC*, there are multiple web servers, each of which serves the data access demands from one geographical region and connects to its own data node server, which is called the host server of that connected web server. Data can be replicated asynchronously between different data servers within the same *DC* or in different *DCs*. When a data server s_r stores an object with key k , s_r is called a *replica* server of object o_k . Otherwise, s_r is a *non-replica* server. When a DC_r includes at least one replica server of object o_k , DC_r is called a *replica DC* of object o_k . Otherwise, DC_r is a *non-replica DC*. CaDRoP supports partial replication of data. Each data object is replicated in a subset of *DCs*.

CaDRoP consists of the client layer and the data store layer. They communicate with each other through the client library. The client layer implemented in web servers is responsible for storing or retrieving information to or from data node servers and presenting information to the application users. Note that the client layer has to wait for the corresponding response to the current request before sending the next access request. The underlying store layer controls the physical storage in data store servers and the data propagation between them. CaDRoP provides the following operations to the clients:

- **POST(key, object):** A POST operation assigns an object item o_k (e.g., a picture or a clip) with an item key.
- **PUT(key, value):** A PUT operation assigns a text value (strings) to an item key. Then, a new version value will be created. Note that if an object is visible to clients, the corresponding key always exists, unless the data object of an item key is removed from the whole system.
- **[values] \leftarrow GET(key):** The GET operation returns [values] corresponding to an item key in causality order.

6.2.1 Convergent Conflict Handling

Causal consistency does not establish a global order for operations in O_H . Therefore, there exist some causally independent operations, which are characterized as concurrent. Formally, two operations o_1 and o_2 in O_H are concurrent if $o_1 \not\prec_{co} o_2$ and $o_2 \not\prec_{co} o_1$. Concurrent write operations applied to the same data object very likely lead to inconsistent data states. Those are

TABLE XX. Definition of symbols and parameters used in the model.

Term	Meaning
D	The set of <i>datacenters</i> (<i>DCs</i>)
L	The set of all the data server nodes
dm_c	Dependency meta-data <i>dep</i> m set at client c
o_k	An object with a unique key k
$cvl\langle k \rangle$	A causal version list of data object k (o_k)
$IOset$	The invisible object set
TS	the local Lamport timestamp for update operations
s_i	the data node server i
d	An item tuple $\langle k, v, dm \rangle$
VV_i	The version vector of data node s_i
$V(k)$	The size of data object k
Δt	Time slot interval
t_h	The h -th time slot
$GN_{t_h}[k][i]$	Number of <i>Gets</i> for o_k from s_i in time slot t_h
$PN_{t_h}[k][i]$	Number of <i>Puts</i> for o_k from s_i in time slot t_h
$AN_{t_h}[k][i]$	The sum of $GN_{t_h}[k][i]$ and $PN_{t_h}[k][i]$
m	Comment message size

said to be in “*conflict*”. Essentially, conflicts do not result in causal violation. However, when different concurrent versions of a data object are replicated to remote stores, this potentially leads to divergent undesired results to clients. Multiple concurrent versions of an object could be present in the system at the same time. In this work, CaDRoP uses the timestamp and the local data node identification to order the list of version values. This can achieve a global consistent state for different data replica nodes. Thus, CaDRoP can provide causal consistency with the convergence property.

6.3 Algorithm: CaDRoP

CaDRoP is adapted from Opt-Track protocol (1; 61), which aims at reducing the dependency metadata size and storage cost for causal ordering in a partially replicated shared system. We now give the formal CaDRoP algorithm in Algorithm 8 ~ 12.

6.3.1 The Client Layer

The client library maintains for its session a dependency metadata, denoted as dm_c . dm_c consists of a set of $\langle rid, TS, Dests \rangle$ tuples, each of which indicates an update operation (POST or PUT) initiated by data node server rid at clock time TS in the causal past. $Dests$ includes replica data node servers for that update operation. Only necessary replica node information is stored.

When PUT() or POST() is invoked, the client library retrieves the local dm_c and assigns POSTREQ or PUTREQ attribute to propagate a new object or a new value with dm_c to its host data node server. The host server is in charge of distributing requests to other replica node servers, handling responses from others, and returning feedbacks to the client. When GET() is invoked, the client library assigns GETREQ attribute to propagate an access request to its host data node. Function MERGE() in Algorithm 8 and 11 merges the piggybacked dependency metadata of the corresponding updates to an object key with the local client dm_c . Function PURGE() in Algorithm 8 and 11 removes old records with empty $Dests$, based on Implicit Tracking in Opt-Track protocol (1; 61). In this function, some new additional dependencies get added to dm_c and some old existing dependencies in dm_c are deleted. The merging process implements the optimality techniques in terms of Implicit Tracking in Opt-Track protocol and

makes the client aware of the necessary causal dependency information of update operations. When the client receives $f(k)$, it updates the object booking table to make users aware of what posts exist in a social network.

Algorithm 8: Client operations at client c_i

POST(object_key k , object o_k , dep dm_c):
 1 send $\langle \text{POSTREQ } k, o_k, dm_c \rangle$ to host data server s_i ;
 2 receive $\langle \text{POSTREPLY } dmr \rangle$;
 3 $dm_c \leftarrow dmr$;
 4 insert k into object name list;
PUT(object_key k , text v , dep dm_c):
 5 send $\langle \text{PUTREQ } k, v, dm_c \rangle$ to data server s_i ;
 6 receive $\langle \text{PUTREPLY } dmr \rangle$;
 7 $dm_c \leftarrow dmr$;
GET(object_name k):
 8 send $\langle \text{GETREQ } k \rangle$ to host data server s_i ;
 9 receive $\langle \text{GETREPLY } cvl\langle k \rangle \rangle$;
 10 **for** each $d \in cvl\langle k \rangle$ **do**
 11 MERGE($DM_c, d.dma$);
 12 $DM_c \leftarrow \text{PURGE}(DM_c)$;
 13 return $cvl\langle k \rangle.values$;
Upon receive $f(k)$:
 14 insert k into the object booking table;

6.3.2 The Storage Layer

The data storage layer is composed of multiple data node servers. Each data object can be replicated to one or more data node servers. As mentioned before, the CaDRoP data store layer exposes three main functions to the client library:

- $\langle \text{POSTREPLY } dmr \rangle \leftarrow \langle \text{POSTREQ } k, o_k, dm_c \rangle$.
- $\langle \text{PUTREPLY } dmr \rangle \leftarrow \langle \text{PUTREQ } k, v, dm_c \rangle$
- $\langle \text{GETREPLY } cvl\langle k \rangle \rangle \leftarrow \langle \text{GETREQ } k \rangle$

Note that dm denotes a dependency metadata set and dmr indicates a returned dm . In Algorithm 9, for a POSTREQ operation in a host data node server, it needs to update the local Lamport timestamp TS (line 1). Then, the metadata per data node server is tailored by REDUCE() in line 3 to minimize its space overhead. It is denoted as dm_s . If s_j is a replica node server, four elements (dm_s , the set of replicas, TS , o_k) are encapsulated into a package d . Line 4 propagates d to each other replica server s_j . If s_j is not a replica server, o_k is replaced with the key id k . The four elements are encapsulated into a package f . Line 5 propagates f to each non-replica server.

Lines 12-13 prune the $Dests$ information, based on the propagation condition in Opt-Track protocol. In the PURGE(), entries with empty $Dests$ are kept as long as they are the most recent update from the source node server. In CaDRoP, we assume that the host server for the client initiating a post o_k is always a replica of object o_k . Lines 14-16 store the source server (rid) and the timestamp(TS) of a POST operation as an entry (denoted as $dm(h)$) at the head of dm and create a data element d to save o_k and the associated metadata dm . Then, d is inserted to the head of $cvl\langle k \rangle$. Line 18 updates the version vector for the host server s_i . Line 19 updates the booking information for object o_k .

Compared with general key-value database systems, social network systems have access to data objects with much larger space overheads. Thus, CaDRoP implements a relay mechanism

to reduce the data communication cost across different DCs . If there are multiple replica servers in a remote datacenter DC_x , lines 7-9 will select a relay replica server s_r and propagate a package d to s_r . Once s_r receives d , it invokes REDUCE() to modify the dm_c from the source data node and, then, relays an updated d to other data nodes servers in the same DC_x . If there is no replica server in a remote datacenter DC_y , line 10-11 implements the similar process as line 5,8-9 to propagate a package f to a non-replica server s_{nr} .

Lines 37-48 handle the process, when d for a POST operation is received by a replica server. The determination ATP realizes an activation predicate of a safe protocol to stop the visibility of any update operation that arrives out of order with respect to \prec_{co} . Lines 49-55 deal with the process, when f for a POST operation is received by a non-replica server. After receiving d in a replica server, a copy of the object posted and the replica placement list (*replicas*) are stored. When receiving f in a non-replica server, it only needs to save *replicas*. Line 50 is required to maintain an explicit dependency between two POST operations.

The function for a PUTREQ operation is similar to that for a POST operation. Instead of replicating an object, PUTREQ propagates a text value in the package d . Note that when a data node server implementing function PUTREQ is a non-replica server, d would not be saved.

In Algorithm 10, lines 1-17 run in the case when a replica server in a remote DC_x receives a package d for a POST operation. Lines 18-28 handle the process when a non-replica server in a remote DC_y receives a package f for a POST operation. Lines 29-37 or 38-50 deal with the case when a replica server within the same DC or in a remote DC_x receives a package d for a PUT operation. When a data server s_i receives a d (for an object o_k or a text value v) or

a f (for an object notification), $\text{ATP}()$ is used to check if the d or f is visible to clients. If the received item is not visible, it will be temporarily stored in $IOset$ until the $\text{ATP}()$ test becomes true.

For a GETREQ operation to a key k in s_i , if s_i is a replica server of object o_k , $cvl\langle k \rangle$ returns to the client. If s_i is a non-replica server of object o_k , s_i needs to fetch $cvl\langle k \rangle$ from a replica server. However, if $cvl\langle k \rangle$ is fetched from a different data server, CaDRoP uses $\text{ATP}()$ to check if each value is causally visible.

In CaDRoP, $\text{LINK}()$ is used to insert a d with an updated value into $cvl\langle k \rangle$ in causality order, when an update value is visible. Since $cvl\langle k \rangle$ is a causal list of values, the following condition must be satisfied:

$$\begin{aligned} & \forall d' \in cvl\langle k \rangle : \\ & (d'.dm(h).rid, d'.dm(h).TS) \neq (d.dm(h).rid, d.dm(h).TS) \end{aligned} \tag{6.1}$$

However, some entries in $cvl\langle k \rangle$ are concurrent with d . CaDRoP can sort those concurrent entries by their TS and rid , in ascending order. Thus, the text values of $cvl\langle k \rangle$ saved in different data servers can be present in the same convergent order. As shown in Figure 33, when two users retrieve the $cvl\langle k \rangle$ from s_1 and s_2 , respectively, they can obtain a consistent result in causality order.

6.3.3 Dynamic Replication Model

Most of the existing causal consistency protocols are based on static replication models in geo-replicated data stores. In other words, the numbers of replicas for a variety of data ob-

Algorithm 9: Operations at data node s_i in DC_i (part1)

```

Upon receive(POSTREQ  $o_k, dm_c$ )
1   $TS \leftarrow \text{LamportTimestamp.increaseAndGet}();$ 
2  for each data node  $s_j$  in the local  $DC_i$  do
3     $dm_s \leftarrow \text{REDUCE}(dm_c.clone, L, s_j);$ 
4    if  $s_j \in k.replicas$  then
5      send  $d(o_k, k.replicas, TS, dm_s)$  to  $s_j$ ;
6    else send  $f(k, k.replicas, TS, dm_s)$  to  $s_j$ ;
7  for each  $DC_j \neq DC_i$  do
8    if  $DC_j$  is a replica DC of  $o_k$  then
9      select a replica server  $s_r$  in  $DC_j$ ;
10     send  $d(o_k, k.replicas, TS, dm_c)$  to  $s_r$ ;
11  else
12    select a data node  $s_{nr}$  in  $DC_j$ ;
13    send  $f(k, k.replicas, TS, dm_c)$  to  $s_{nr}$ ;
14 for each  $o \in dm_c$  do
15    $o.Dests := \setminus L;$ 
16  $dm_c := \cup\{ \langle rid = s_i, TS, L \setminus \{s_i\} \rangle \};$ 
17  $dm \leftarrow \text{PURGE}(dm_c);$ 
18 create  $d(o_k, dm)$  and  $cvl\langle k \rangle$ ;
19  $\text{LINK}(cvl\langle k \rangle, d) : \text{insert } d \text{ to } cvl\langle k \rangle;$ 
20  $VV_i[i].\text{increment};$ 
21  $\text{OBJTABLEUPDATE}(k, replicas);$ 
22 return  $\langle \text{POSTREPLY } dmr = dm \rangle$  to the request client;

Upon receive(PUTREQ  $k, v, dm_c$ )
23  $TS \leftarrow \text{LamportTimestamp.increaseAndGet}();$ 
24 for each  $s_j \in k.replicas$ , in the local  $DC_i$  do
25    $dm_s \leftarrow \text{REDUCE}(dm_c.clone, o_k.replicas, s_j);$ 
26   send  $d(k = v, rid = s_i, TS, dm_s)$  to  $s_j$ ;
27 for each replica  $DC_j \neq DC_i$  do
28   select a replica server  $s_r$  in  $DC_j$ ;
29   send  $d(k = v, rid = s_i, TS, dm_c)$  to  $s_r$ ;
30 for each  $o \in dm_c$  do
31    $o.Dests := \setminus k.replicas;$ 
32  $dm_c := \cup\{ \langle rid = s_i, TS, k.replicas \setminus \{s_i\} \rangle \};$ 
33  $dm \leftarrow \text{PURGE}(dm_c);$ 
34 if  $s_i \in o_k.replicas$  then
35   create  $d(k = v, dm)$ ;
36    $\text{LINK}(cvl\langle k \rangle, d) : \text{insert } d \text{ to } cvl\langle k \rangle;$ 
37    $VV_i[i].\text{increment};$ 
38 return  $\langle \text{PUTREPLY } dmr = dm \rangle$  to the request client;

Upon receive  $d(o_k, replicas, TS, dm_s)$  from  $DC_i$ 
39  $rid \leftarrow replicas.getFirst();$ 
40 if  $\text{ATP}(dm_s, VV_i, s_i) = \text{true}$  then
41    $dm_s := \cup\{ \langle rid, TS, replicas \rangle \};$ 
42   for each  $o \in dm_s$  do
43      $o.Dests := \setminus s_i;$ 
44     create  $d'(o_k, dm_s)$  and  $cvl\langle k \rangle$ ;
45     insert  $d'$  to  $cvl\langle k \rangle$ ;
46      $VV_i[rid] \leftarrow TS;$ 
47     update  $IOset$ ;
48     send  $f(k)$  to the local client  $c_i$ ;
49 else insert  $d$  into  $IOset$ ;
50  $\text{OBJTABLEUPDATE}(k, replicas);$ 

Upon receive  $f(k, replicas, TS, dm_s)$  from  $DC_i$ 
51  $rid \leftarrow replicas.getFirst();$ 
52 if  $\text{ATP}(dm_s, VV_i, s_i) = \text{true}$  then
53    $VV_i[rid] \leftarrow TS;$ 
54   update  $IOset$ ;
55   send  $f(k)$  to the local client  $c_i$ ;
56 else insert  $d$  into  $IOset$ ;
57  $\text{OBJTABLEUPDATE}(k, replicas);$ 

```

Algorithm 10: Operations at data node s_i in DC_i (part2)

```

Upon receive  $d(o_k, replicas, TS, dm_c)$  from  $DCs \neq DC_i$ 
1 for each data node  $s_j (\neq s_i)$  in  $DC_i$  do
2    $dm_s \leftarrow \text{REDUCE}(dm_c.clone, replicas, s_j)$ ;
3   if  $s_j$  is a replica node of  $o_k$  then send  $d(o_k, k.replicas, TS, dm_s)$  to  $s_j$ ;
4   else send  $f(k, k.replicas, TS, dm_s)$  to  $s_j$ ;
5  $\text{REDUCE}(dm_c, replicas, s_i)$ ;
6  $rid \leftarrow replicas.getFirst()$ ;
7 if  $ATP(dm_c, VV_i, s_i) = \text{true}$  then
8    $dm_c := \cup\{ \langle rid, TS, replicas \rangle \}$ ;
9   for each  $o \in dm_c$  do
10     $o.Dests := \setminus s_i$ ;
11   create  $d'(o_k, dm_c)$  and  $cvl\langle k \rangle$ ;
12   insert  $d'$  to  $cvl\langle k \rangle$ ;
13    $VV_i[rid] \leftarrow TS$ ;
14   update  $IOset$ ;
15   send  $f(k)$  to the local client  $c_i$ ;
16 else insert  $d$  into  $IOset$ ;
17  $\text{OBJTABLEUPDATE}(k, replicas)$ ;
Upon receive  $f(k, replicas, TS, dm_c)$  from  $DCs \neq DC_i$ 
18 for each data node  $s_j (\neq s_i)$  in  $DC_i$  do
19    $dm_s \leftarrow \text{REDUCE}(dm_c.clone, replicas, s_j)$ ;
20   send  $f(k = v, replicas, TS, dm_s)$  to  $s_j$ ;
21  $\text{REDUCE}(dm_c, L, s_i)$ ;
22  $rid \leftarrow replicas.getFirst()$ ;
23 if  $ATP(dm_c, VV_i, s_i) = \text{true}$  then
24    $VV_i[rid] \leftarrow TS$ ;
25   update  $IOset$ ;
26   send  $f(k)$  to the local client  $c_i$ ;
27 else insert  $d$  into invisible object list;
28  $\text{OBJTABLEUPDATE}(k, replicas)$ ;
Upon receive  $d(k = v, rid = s_j, TS, dm_s)$  from  $DC_i$ 
29 if  $ATP(dm_s, VV_i, s_i) = \text{true}$  then
30    $dm_s := \cup\{ \langle rid, TS, replicas \rangle \}$ ;
31   for each  $o \in dm_s$  do
32     $o.Dests := \setminus s_i$ ;
33   create  $d'(k = v, dm_s)$ ;
34   insert  $d'$  to  $cvl\langle k \rangle$ ;
35    $VV_i[rid] \leftarrow TS$ ;
36   update  $IOset$ ;
37 else insert  $d$  into  $IOset$ ;
Upon receive  $d(k = v, rid, TS, dm_c)$  from  $DCs \neq DC_i$ 
38 for each replica data node  $s_j (\neq s_i)$  in  $DC_i$  do
39    $dm_s \leftarrow \text{REDUCE}(dm_c.clone, replicas, s_j)$ ;
40   send  $d(o_k, k.replicas, TS, dm_s)$  to  $s_j$ ;
41  $\text{REDUCE}(dm_c, replicas, s_i)$ ;
42 if  $ATP(dm_c, s_i) = \text{true}$  then
43    $dm_c := \cup\{ \langle rid, TS, replicas \rangle \}$ ;
44   for each  $o \in dm_c$  do
45     $o.Dests := \setminus s_i$ ;
46   create  $d'(k = v, dm_c)$ ;
47   insert  $d'$  to  $cvl\langle k \rangle$ ;
48    $VV_i[rid] \leftarrow TS$ ;
49   update  $IOset$ ;
50 else insert  $d$  into  $IOset$ ;
Upon receive  $\langle \text{GETREQ } k \rangle$ 
51 if  $s_i \notin k.replicas$  then
52   send  $\langle \text{REQUEST } k \rangle$  to a replica node in  $DC_i$  or a remote  $DC$ ;
53   receive  $\langle \text{RREQ } cvl\langle k \rangle \rangle$ ;
54   for each  $d \in cvl\langle k \rangle$  do
55     if  $(ATP(dm_d, VV_i, s_i) = \text{false})$  then
56       remove  $d$  from  $cvl\langle k \rangle$ ;
57 else fetch  $cvl\langle k \rangle$ ;
58 send  $\langle \text{GETREPLY } cvl\langle k \rangle \rangle$ ;
Upon receive  $\langle \text{REQUEST } k \rangle$ 
59 fetch  $cvl\langle k \rangle$  and return  $\langle \text{PREQ } cvl\langle k \rangle \rangle$ ;

```

Algorithm 11: Functions used in Algorithm 8, 9, and 10

```

boolean ATP(depm dm, int[] VVsi, node si):
1  for each  $o \in dm$  do
2    if  $s_i \in o_{z,ts}.Dests$  then
3      if  $ts > VV_i[z]$  then return false;
4  return true;

REDUCE(depm dm, node_list replicas, node sn):
5  for each  $o \in dm$  do
6    if  $s_n \in o.Dests$  then  $o.Dests = \backslash replicas$ ;
7    else  $o.Dests := \backslash replicas \cup s_n$ ;
8    if  $o_z.Dests = \emptyset \wedge (\exists o'_z \in dm | o_z.ts < o'_z.ts)$  then  $dm \setminus o_z$ ;
9  return  $dm$ ;

PURGE( $dm$ ):
10 for each  $o \in dm$  do
11   if  $o_z.Dests = \emptyset \wedge (\exists o'_z \in dm | o_z.ts < o'_z.ts)$  then  $dm \setminus o_z$ ;
12 return  $dm_c$ ;

MERGE( $dm_c, dm_d$ ):
13 for all  $o_{z,tz} \in dm_d$  and  $o_{s,ts} \in dm_c$  and  $s = z$  do
14   if  $tz < ts \wedge o_{s,ts} \notin dm_c$  then mark  $o_{z,tz}$  ;
15   if  $ts < tz \wedge o_{z,ts} \notin dm_d$  then mark  $o_{s,ts}$  ;
16   delete marked entries;
17   if  $tz = ts$  then
18      $o_{s,ts}.Dests := \cap o_{z,ts}.Dests$ ;
19     delete  $o_{z,t}$  from  $dm_d$ ;
20  $dm_c := dm_c \cup dm_d$ ;

OBJTABLEUPDATE(object_id k, node_list replicas):
21 ObjectTable $\langle k \rangle = replicas$ ;

```

Algorithm 12: Cache operations at data server s_i

Upon receive(PUTREQ k, v, dm_c) in a replica master node

- 1 **for** each slave caching node s_a of object o_k **do**
- 2 fetch seq by object key k and node id s_a ;
- 3 $seq.increase()$;
- 4 send (CACHE $d(k = v, seq, dm_r)$) to s_a ;

Upon receive(CACHE $d(k = v, seq, dm_r)$) in a cache node

- 5 **wait until** ($d.seq = k.seq + 1$);
- 6 $k.seq.increase()$;
- 7 insert $d(k = v, dm_r)$ to $cvl\langle k \rangle$;

Upon receive(REQUEST k) from a non-replica node s_j

- 8 insert $\langle s_j, seq=0 \rangle$ to a cache seq map for object key k ;
- 9 fetch $cvl\langle k \rangle$ and return (PREQ $cvl\langle k \rangle$);

Upon receive(GETREQ k) in a non-replica node

- 10 send(REQUEST k) to a replica node in DC_i or a remote DC ;
- 11 receive (RREQ $cvl\langle k \rangle$);
- 12 **for** each $d \in cvl\langle k \rangle$ **do**
- 13 **if** $ATP(dm_d, VV_i, s_i) = \text{false}$ **then**
- 14 move d from $cvl\langle k \rangle$ to invisible list of object k ;
- 15 save $cvl\langle k \rangle$ in s_i and set $k.seq$ to 'O';
- 16 send(GETREPLY $cvl\langle k \rangle$);

jects are predetermined. All replication decisions are made before the system is operational and replica configuration is not changed during operation. However, static replication of data resources in dynamic environments hosting time-varying workloads is obviously ineffective for optimizing system utilization, especially in social network systems. Dynamic replication strategies have been widely used as means of increasing the data availability of large-scale cloud store systems. **CORP** model, a proactive dynamic data replication strategy, has been proposed in (3) to effectively improve the total system cost in a social network system. According to the current data resource allocation and historical changes in workload patterns, **CORP** employs the autoregressive integrated moving average (ARIMA) model to predict data object access

frequency in the near future. In order to optimize system cost, we incorporate **CORP** model as the underlying replication mechanism into CaDRoP protocol. Based on the requirement of **CORP**, a time slot system is required to realize the data migration process in CaDRoP. Each data server is equipped with a physical clock, which generates monotonically increasing timestamps. Physical clocks are synchronized by a time synchronization protocol, such as NTP. The correctness of the CaDRoP is independent of the synchronization precision.

CORP strategy runs at the end of each time slot and outputs a set of replicas for each data object. Then, the home server for that object triggers the migration process, based on the replica placement at the current time slot and that at the next time slot. It is noted that the regular CORP runs the ARIMA prediction model by an equal time interval. At runtime, the prediction is constantly updated. When new access requests arrive in the current time slot, they are getting involved into the time series and the information in the oldest time slot is removed from the time series. However, when a data object is created, there is not sufficient data in the time series initially (i.e., the training data set is not enough). Thus, we extend CaDRoP as CaDRoP (+cache), which is based on a PUSH model, to reduce the network transmission cost, especially in the initial time slot(s). Algorithm 12 presents the cache functions used in CaDRoP (+cache). When a non-replica s_i receives a requesting data package with key k by fetching cvl from another replica server s_r , $cvl\langle k \rangle$ may be cached in s_i (line 16) with a sequence number seq assigned by s_r . For object o_k , s_i becomes a slave server of s_r . Afterwards, whenever s_r receives an update value (lines 1-4), then, s_r relays the update value to s_i with a seq (increasing by one per PUT). Based on the seq , s_i can maintain a visible $cvl\langle k \rangle$ in causality order. Algorithm 13

presents the migration processes in CaDRoP (+cache). When the migration process initiates, **CORP** outputs a new set of replicas of a key k (denoted as $k.replicas'$) for the next time slot t_h to the home server s_i . Based on different replica distributions, s_i will send the $replicas'$ (lines 2-7) or replicate $cvl\langle k \rangle + replicas'$ (line 8) to the other servers within the same DC . Similar to POST or PUT operations, the migration process utilizes the relay mechanism to reduce the network transmission cost across different DC s. The home s_i may just send $k.replicas'$ to DC_j in the following three cases: 1) DC_j is not a replica DC in t_h (lines 10-12). 2) DC_j was a replica DC or included a cache server in t_{h-1} , and is a replica DC in t_h (lines 13-18). 3) DC_j was not a replica DC in t_{h-1} , but will be a replica DC in t_h (line 19-21). After receiving $k.replicas'$ or $k.replicas + cvl\langle k \rangle$ from other DC s, it is required not only to update the replica placement and store $cvl\langle k \rangle$ (if received), but also to relay them to other servers within the same DC (lines 25-32 and 37-43).

6.4 Performance Evaluation

We evaluate the proposed **CaDRoP** protocols by real traces of requests to the web servers from Twitter workload (76) and the CloudSim discrete event simulator (77). These realistic traces contain a mixture of temporal and spatial information for each http request. The number of http requests received for each of the target data objects (e.g., photo images) is aggregated in 1000-secs intervals. By implementing our approaches on the Amazon cloud provider, it allows us to evaluate the cost-effectiveness of request transaction, data store, and network transmission, and to explore the impact of workload characteristics. We also evaluate CaDRoP

and CaDRoP(+cache) by a clairvoyant Optimal Placement (OPT) Solution, proposed in (3), based on the time slot system and object access patterns known in advance.

Data Object Workload: Our work focuses on the data store framework on image-based sharing in social media networks, where applications have geographically dispersed users who PUT and GET data, and fit straightforwardly into a key-[values] model. We use actual Twitter traces as a representation of the real world. PUT or POST, denoted as *Put*, to a timeline occurs when users post a tweet, retweet, or reply messages. We crawl the real Twitter traces as the evaluation input data. Since the Twitter traces do not contain information of reading the tweets (i.e., the records of *Gets*), we set five different ratios of *Put/Get* (P_{rate} : *Put* rate), where the patterns of *Gets* on the workloads follow Longtail distribution model (78). The simulation workload contains several Tweet objects. The volume V of each target tweet in the workload is 2 MB. The simulation is performed for a period of 20 days. The results for each object show that they have similar tendency.

The experiment has been performed via simulation using the CloudSim toolkit (77) to evaluate the proposed system. CloudSim is a JAVA-based toolkit that contains a discrete event simulator and classes that allow users to model distributed cloud environments, from providers and their system resources (e.g., physical machines and networking) to customers and access requests. CloudSim can be easily developed by extending the classes, with customized changes to the CloudSim core. We figure out our own classes for simulation of the proposed framework and model 9 *DCs* in CloudSim simulator. Each *DC* is composed of 4 pairs of web servers and data servers. Each data server incorporates a 50GB storage space and each web server is in

charge of user's query processing from one (or a few) states in US or one country in Asia and in Europe. The price of the storage classes and network services are set in terms of Amazon Web Service (AWS) as of 2019.

6.4.1 Results and Discussion

The performance metrics we use are the performance in terms of cost and the cost improvement rates under varying P_{rate} of the proposed CaDRoP(+cache). In order to evaluate our proposed algorithm, we compare it to different replication factors (RF). RF is the number of replica DC , where it is randomly pre-selected and each replica DC includes one replica data server. More specifically, when RF is constant and the replica placement for each key is predetermined, the simulation proceeds only by Algorithm8 \sim 11 (we denote this strategy as 'CaS'). Cost is represented by the total system cost, which is composed of transaction cost (TC), network transmission cost (NTC), and storage cost (SC). We use the term 'transaction' to denote data query operations, such as *Put* or *Get*. NTC depends on the size of the packet (e.g., a d packet) transmitted. SC includes the costs of storing data items (including the dm data) and the bookkeeping management of data replication information.

6.4.1.1 CaS VS. CaS+cache

To evaluate the effectiveness of the cache component, we examine the system performance with the comparisons between CaS and CaS+cache on cost improvement rate with respect to different RF , which is defined as:

$$\frac{cost(CaS) - cost(CaS + cache)}{cost(CaS)} \quad (6.2)$$

Table XXI shows the cache effectiveness of different RF modes for different *Put* rates increases as RF decreases. As *Put* rate decreases, the cost improvement of CaS+cache becomes higher except for full DC replication (RF=9).

6.4.1.2 CaS+cache VS. CaDRoP+cache

We now evaluate the cost effectiveness of CaDRoP by comparing CaDRoP+cache with CaS+cache. By running the same workloads as before, Figure 34 presents the TCs of various RF models in different *Put* rates. Lowering the number of transactions to fetch objects from remote data servers increases throughput in cloud environments, while an increased number of transactions would lead to an over-utilization of the underlying systems. Thus, the total TC is completely subject to the number of transactions. The results show that CaDRoP can achieve the best performance for TC under the same cache capacity, although it needs to bring additional transactions for the migration process. Figure 35 presents the NTC of CaDRoP+cache in comparison with various RF models in different *Put* rates. The smaller the NTC, the lower the network bandwidth consumption. Although NTC of CaDRoP+cache is slightly higher than that of the full *DC* replication, it is much lower than others' NTCs. Figure 36 shows the results of SC of CaDRoP+cache in comparison with other alternatives. It is noteworthy that the SC of CaDRoP+cache falls in between the SCs of the replication models with RF=9 and RF=2. This implies that the proper number of replicas for CaDRoP+cache is able to decrease TC and NTC. Figure 37 presents the total system costs (TSC) for CaDRoP+cache and CaS+cache in different RF values. It illustrates that CaDRoP+cache can reduce TC and NTC at the slight cost of SC.

6.4.1.2.1 CaDRoP+cache VS. CaDRoP

We propose CaDRoP+cache to improve the system costs. Thus, in this section we present experiments aimed at evaluating how the total costs are improved by CaDRoP+cache against CaDRoP. Table XXII presents the results of the cost saving ratio (Δ_{saving}) for different *Put* rates. Δ_{saving} is defined as

$$\frac{cost(CaDRoP) - cost(CaDRoP + cache)}{cost(CaDRoP)} \quad (6.3)$$

Since the evaluation data come from the social network, each individual data object brings a lot of requests in the initial time slots. It can be observed that the results indicate that the lower the P_{rate} (Get-intensive), the better the Δ_{saving} is.

6.4.1.2.2 CaDRoP+cache evaluation

In order to evaluate the effectiveness of our proposed approach, we also implemented the Optimal Placement Solution (OPT) proposed in (3) as the clairvoyant replication strategy. OPT knows the exact temporal and spatial data object access patterns. OPT can figure out the optimal object placement for each time slot. The cost effectiveness of CaDRoP+cache is highly related to how precise the predicted access number is. OPT+cache totally follows up CaDRoP+cache model and uses the real object access (*Put* and *Get*) numbers as the inputs in different time slots. Δ_{inc} is defined as

$$\frac{cost(CaDRoP + cache) - cost(OPT + cache)}{cost(CaDRoP + cache)} \quad (6.4)$$

Δ_{inc} in Table XXII presents the comparisons between CaDRoP+cache and OPT+cache for different *Put* rates. It is evident that CaDRoP+cache only increases 6% \sim 16% of total system cost compared to OPT+cache.

In order to measure the cost effectiveness of CaDRoP in steady states (including enough training time slots), we also compare the cost of CaDRoP to that of OPT without caching in steady states. $\Delta_{inc'}$ in Table XXII gives the cost increase ratios ($\Delta_{inc'}$) of CORP compared to OPT for different *Put* rates. We notice that $\Delta_{inc'}$ rates are around 2% \sim 4.5%. $\Delta_{inc'}$ is defined as

$$\frac{cost(CaDRoP) - cost(OPT)}{cost(CaDRoP)} \quad (6.5)$$

6.4.1.3 CoCaCo VS. CaDRoP+cache

In order to empirically evaluate the effectiveness of our approach, we compare it to another causal+ consistency protocol, CoCaCo proposed in (14), for the following reasons. First, CoCaCo implements causal consistency both within and across datacenters. Second, it can be applied to partially replicated systems across datacenters. Third, it also realizes multi-version storage systems to preserve all the updated values for convergent conflict handling. Fourth, the architecture of CoCaCo is highly similar to that of CaDRoP. In order to achieve the potential causal consistency for all the text values with the same key, each update operation (PUT or POST) needs to specify a unique id and to assign it to the corresponding text value. 6.4.1.3 demonstrates the simulation results for CoCaCo and CaDRoP+cache by running the workloads used in the above experiments in various *Put* rates. As the RF value decreases, the overheads

of storing dm decrease in terms of the SC results, but the volume of transmitting dm over networks increases in terms of the NTC results. For CoCaCo, the TC costs are apparently higher than those of CaDRoP+cache, since CoCaCo invokes more acknowledgement messages and implements access requests. The SC costs of CoCaCo are lower than those of CaDRoP+cache in the lower RF values, while CoCaCo's SC is higher in the higher RF value.

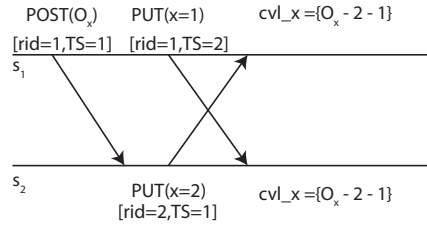


Figure 33. An example with the convergence property.

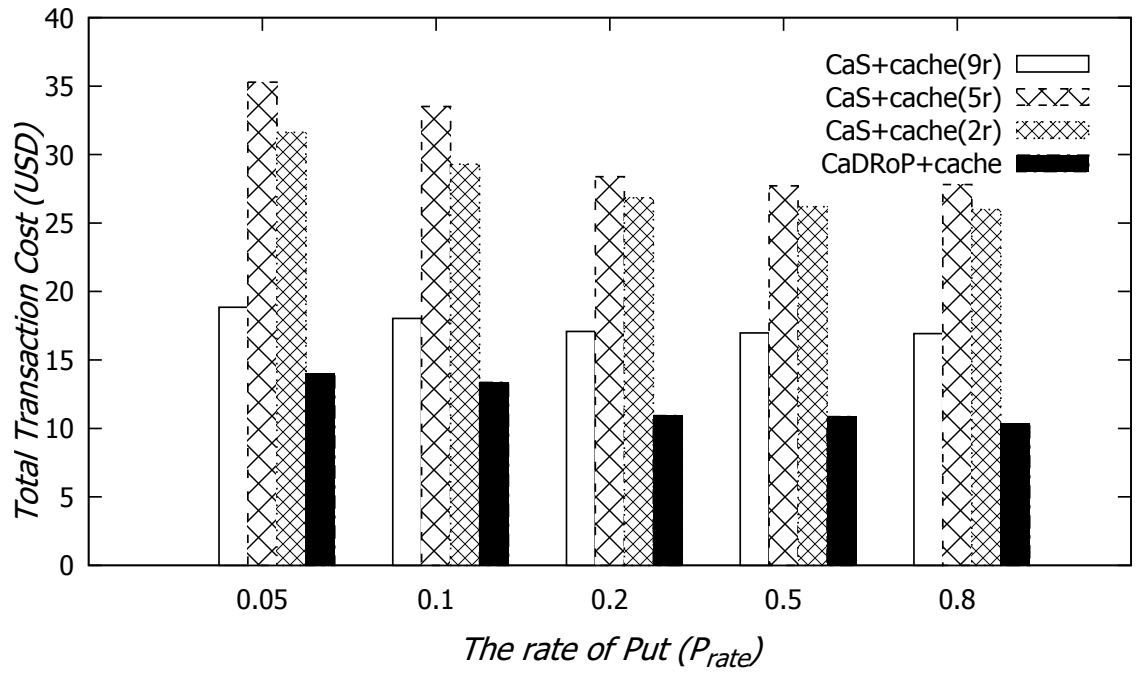


Figure 34. The Transaction Cost

Algorithm 13: Migration operations at s_i for o_k in DC_i at t_{h-1}

```

Upon receive  $\langle \text{MIG } k, k.\text{replicas}' \rangle$  in a home  $s_i$ 
1 for each  $s_j (\neq s_i)$  in  $DC_i$  do
2   if  $s_j \notin k.\text{replicas}'$  then
3     send  $f(k, k.\text{replicas}')$  to  $s_j$ ;
4   else if  $s_j \in k.\text{replicas}'$  and  $s_j \in k.\text{replicas}$  then
5     send  $f(k, k.\text{replicas}')$  to  $s_j$ ;
6   else if  $s_j \in k.\text{replicas}'$  and  $s_j$  is a caching server then
7     send  $f(k, k.\text{replicas}')$  to  $s_j$ ;
8   else send  $\langle \text{MIGR } k, k.\text{replicas}', cvl\langle k \rangle \rangle$  to  $s_j$ ;
9 for each  $DC_j \neq DC_i$  do
10   if  $R(DC_j) = \text{false}$  in  $t_h$  then
11     select  $s_j$  with the largest  $AN_{t_h}$  from  $DC_j$ ;
12     send  $f(k, k.\text{replicas}')$  to  $s_j$ ;
13   else if  $R(DC_j) = \text{true}$  in  $t_{h-1}$  then
14     select a replica  $s_j$  from  $DC_j$ ;
15     send  $f(k, k.\text{replicas}')$  to  $s_j$ ;
16   else if  $DC_j$  includes one caching server in  $t_{h-1}$  then
17     select a caching server  $s_j$  from  $DC_j$ ;
18     send  $f(k, k.\text{replicas}')$  to  $s_j$ ;
19   else
20     select  $s_j$  with the largest  $AN$  from  $DC_j$ ;
21     send  $\langle \text{MIGRB } k, k.\text{replicas}', cvl\langle k \rangle \rangle$  to  $s_j$ ;

Upon receive  $f(k, k.\text{replicas}')$  from  $DC_i$ 
22 if  $s_i \notin k.\text{replicas}'$  &  $s_i \in k.\text{replicas}$  then
23   remove  $cvl\langle k \rangle$ ;
24 OBJTABLEUPDATE( $k, \text{replicas}'$ );

Upon receive  $f(k, k.\text{replicas}')$  from  $DC_j$  ( $j \neq i$ )
25 for each  $s_j (\neq s_i)$  in  $DC_i$  do
26   if  $s_j \notin k.\text{replicas}'$  then
27     send  $f(k, k.\text{replicas}')$  to  $s_j$ ;
28   else
29     fetch  $cvl\langle k \rangle$ ;
30     send  $\langle \text{MIGR } k, k.\text{replicas}', cvl\langle k \rangle \rangle$  to  $s_j$ ;
31 if  $s_i \notin k.\text{replicas}'$  &  $s_i \in k.\text{replicas}$  then
32   remove  $cvl\langle k \rangle$ ;
33 OBJTABLEUPDATE( $k, \text{replicas}'$ );

Upon receive  $\langle \text{MIGR } k, k.\text{replicas}', cvl\langle k \rangle \rangle$ 
34 for each  $d \in cvl\langle k \rangle$  do
35   if  $(ATP(dm_d, VV_i, s_i) = \text{false})$  then
36     move  $d$  from  $cvl\langle k \rangle$  to the invisible list of  $o_k$ ;
37 OBJTABLEUPDATE( $k, \text{replicas}'$ );

Upon receive  $\langle \text{MIGRB } k, k.\text{replicas}', cvl\langle k \rangle \rangle$  from  $DC_j$  ( $j \neq i$ )
38 for each  $s_j (\neq s_i)$  in  $DC_i$  do
39   if  $s_j \in k.\text{replicas}'$  then
40     send  $\langle \text{MIGR } k, k.\text{replicas}', cvl\langle k \rangle \rangle$  to  $s_j$ ;
41 for each  $d \in cvl\langle k \rangle$  do
42   if  $(ATP(dm_d, VV_i, s_i) = \text{false})$  then
43     move  $d$  from  $cvl\langle k \rangle$  to the invisible list of  $o_k$ ;
44 OBJTABLEUPDATE( $k, \text{replicas}'$ );

```

TABLE XXI. Cost improvement rates in different Put rates and RF values.

P_{rate}	0.05	0.1	0.2	0.5	0.8
RF=9	3.46%	2.87%	5.24%	4.41%	4.16%
RF=5	72.62%	58.88%	55.05%	21.35%	6.74%
RF=2	79.46%	69.49%	56.33%	29.08%	11.95%

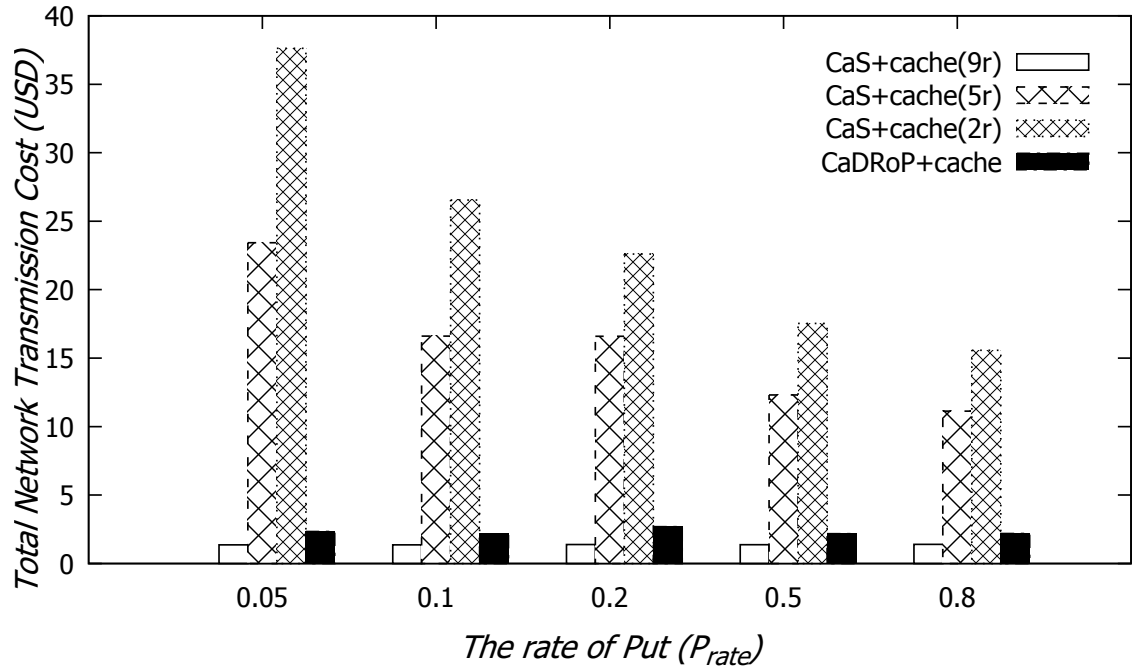


Figure 35. The Network Transmission Cost

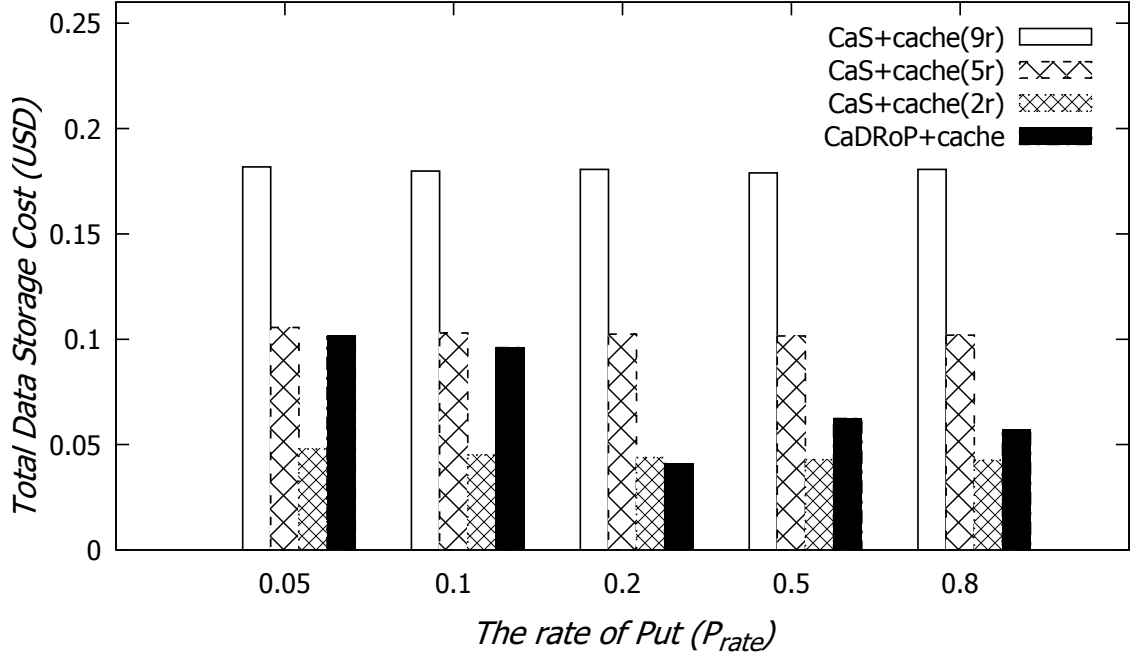


Figure 36. The Storage Space Cost

TABLE XXII. Δ_{saving} : The cost improvement results for different Put rates show that caching has taken an important step to improve the total system costs. Δ_{inc} : The performance evaluation of CaDRoP+cache compared to OPT+cache. $\Delta_{inc'}$: The performance evaluation of CaDRoP compared to OPT in steady states.

P_{rate}	0.05	0.1	0.2	0.5	0.8
Δ_{saving}	91.95%	85.01%	75.14%	57.84%	49.02%
Δ_{inc}	16.08%	13.17%	10.21%	9.02%	6.17%
$\Delta_{inc'}$	1.72%	2.62%	1.6%	4.51%	3.31%

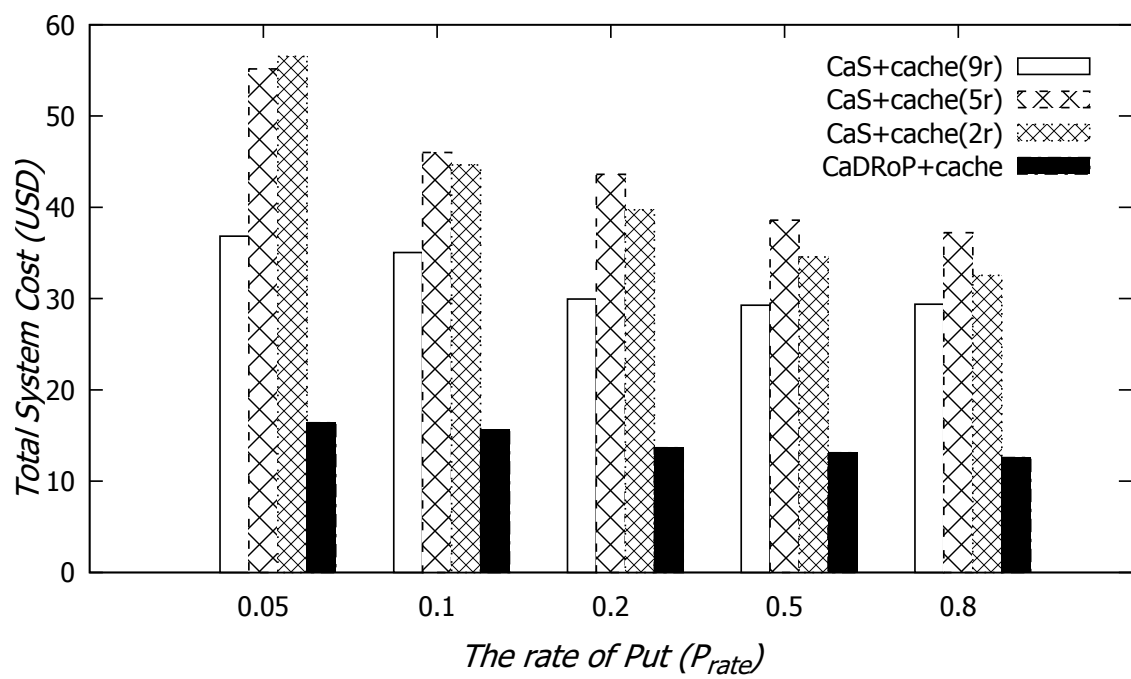


Figure 37. The Total System Cost

TABLE XXIII. The price cost comparisons between CoCaCo and CaDRoP+cache in different P_{ut} rates and RF models. ‘SC’ includes two costs: (i) storing data objects + (ii) storing dm data. Similarly, ‘NTC’ includes two costs: (i) transmitting data objects + (ii) transmitting dm data.

	CoCaCo				CaDRoP+cache			
	SC	TC	NTC	TSC	SC	TC	NTC	TSC
$P_{rate}=0.05$	RF=9	0.155+0.029	19.57	1.3+0.256	21.3			
	RF=5	0.086+0.018	27.525	186.1+14.23	228	0.099+0.002	13.97	2.273+0.06
	RF=2	0.035+0.007	25.07	329.7+24.77	380			16.4
$P_{rate}=0.1$	RF=9	0.155+0.028	18.594	1.3+0.250	20.3			
	RF=5	0.086+0.017	25.503	97.33+6.805	130	0.094+0.002	13.36	2.112+0.051
	RF=2	0.035+0.007	24.072	160.2+12.47	196			15.62
$P_{rate}=0.2$	RF=9	0.155+0.026	18.279	1.3+0.246	20			
	RF=5	0.086+0.017	25.091	78.002+4.362	107	0.039+0.002	10.93	2.607+0.085
	RF=2	0.035+0.006	22.683	90.62+6.136	119			13.67
$P_{rate}=0.5$	RF=9	0.155+0.026	17.797	1.3+0.245	19.5			
	RF=5	0.086+0.017	24.704	26.837+2.197	54	0.061+0.001	10.86	2.113+0.07
	RF=2	0.035+0.006	22.290	42.10+3.024	67			13.11
$P_{rate}=0.8$	RF=9	0.155+0.025	17.731	1.3+0.235	19.4			
	RF=5	0.086+0.016	22.628	19.516+1.890	44	0.055+0.002	10.33	2.112+0.078
	RF=2	0.035+0.006	20.216	29.296+2.602	52			12.58

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In Chapter 3, we considered the problem of providing causal consistency in large-scale DSM systems under partial replication and proposed two algorithms. The first algorithm, Full-Track, is optimal in the sense that each update is applied at the earliest instant while removing false causality in the system. The second algorithm, Opt-Track, is additionally optimal in the sense that it minimizes the size of meta-information carried on messages and stored in local logs. In addition, as a derivative of the second algorithm, we proposed an optimized algorithm, Opt-Track-CRP, that reduces the message overhead, the processing time, and the local storage cost at each site in the fully replicated scenario. We also conducted a performance analysis of the message space and message count complexity of the algorithms under a wide range of system conditions. Experimental results indicated that Opt-Track was seen to show significant gains over Full-Track in partial replication and highly successful in trimming the size of meta-data. In full replication, the results also supported that Opt-Track-CRP performed better than *optP* in scalability and network capacity utilization.

In Chapter 4, we introduced the concepts of *credits* (cr) and proposed algorithm Approx-Opt-Track to further improve the meta-data size of Opt-Track. By controlling a parameter called cr , 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 . The experimental results indicated 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. In other words, 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.

In Chapter 5, we proposed the CORP(+cache) algorithms to realize a proactive provisioning replication model by using ARIMA for cloud datastores. Our approach employed a mixture of ARIMA and analytic schemes to analyze data access workload patterns in a predefined window length to predict workload distribution for the next time interval. The CDP can dynamically deploy required data replicas in the distributed storage system in responding to the predicted data access requests for the next interval. Simulations indicated that, with caching, as the number of *replica* GDCs increases, the TSC decreases. CORP+cache is capable of further reducing the TSC against the static replication mode with caching. Compared to the optimal placement solution (OPT+cache), CORP+cache increases only 6% \sim 16% of TSC of OPT+cache. Without caching, the TSC of CORP is highly close to that of OPT for the time slot system in a steady state.

In Chapter 6, we considered the problem of ensuring causal+ consistency between posts and for the comments under each post in social network systems. By extending Opt-Track into a client-server topology, we proposed algorithm CaDRoP+cache that is adapted to the proposed CORP strategy in in geo-replicated datastores. Simulations showed that, with caching, as RF increases, the TSC decreases. CaDRoP+cache is around 55% \sim %70 lower than CaS+cache

in different predetermined RF models. The simulation results also showed that the TSC of CaDRoP+cache is usually improved better in lower P_{rate} . It implies that CaDRoP+cache is cost-effective for most social applications with Get-intensive workloads.

These algorithms we designed in this work provide a suite of protocols for enforcing causal consistency in large-scale geo-replicated data store systems. As for the future work, this work can be further explored in the following directions.

- Extend the cost optimized replication protocol so that it is adapted to dynamic ring-binding within each datacenter and compare its cost-effectiveness with other alternative replication strategies.
- Extend Approx-Opt-Track algorithm to the client-server topology architecture with fault-tolerance support and perform realistic workloads to test its performances in more depth.
- Extend the CaDRoP algorithm to provide fault-tolerance and support data sharding within each datacenter. We would like to explore partition fault-tolerance mechanisms to design highly reliable high performance systems in the case a partition goes down in a datacenter.

APPENDICES

Appendix A

IEEE LICENSE DOCUMENTS



Home

Help

Email Support

Sign in

Create Account



Performance of Causal Consistency Algorithms for Partially Replicated Systems

Conference Proceedings:

2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)

Author: Ta-Yuan Hsu; Ajay D. Kshemkalyani

Publisher: IEEE

Date: 23-27 May 2016

Copyright © 2016, IEEE

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

- 1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
- 2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
- 3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

- 1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
- 2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
- 3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

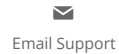
BACK

CLOSE WINDOW

Appendix A (Continued)



RightsLink®



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

Author: Ta-Yuan Hsu

Publication: Parallel and Distributed Systems, IEEE Transactions on

Publisher: IEEE

Date: 1 Jan. 2018

Copyright © 2018, IEEE

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

- 1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
- 2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
- 3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

- 1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
- 2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
- 3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.


If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.





BACK


CLOSE WINDOW

Appendix B

ELSEVIER PERMISSIONS

 Home
  Help
  Email Support
  TAYUAN HSU ▾



Causal consistency algorithms for partially replicated and fully replicated systems

Author: Ta-Yuan Hsu, Ajay D. Kshemkalyani, Min Shen

Publication: Future Generation Computer Systems

Publisher: Elsevier

Date: September 2018

© 2017 Elsevier B.V. All rights reserved.

Please note that, as the author of this Elsevier article, you retain the right to include it in a thesis or dissertation, provided it is not published commercially. Permission is not required, but please ensure that you reference the journal as the original source. For more information on this and on your other retained rights, please visit: <https://www.elsevier.com/about/our-business/policies/copyright#Author-rights>

BACK CLOSE WINDOW

© 2020 Copyright - All Rights Reserved | [Copyright Clearance Center, Inc.](#) | [Privacy statement](#) | [Terms and Conditions](#)
Comments? We would like to hear from you. E-mail us at customercare@copyright.com

The following document is obtained from the ELSEVIER official author web page available at <https://www.elsevier.com/about/policies/copyrightAuthor-rights>. It clearly states that, as far as full acknowledgement is given, the author can reuse all or part of a previously published article through Elsevier to be included in a thesis or dissertation.

Appendix B (Continued)



ELSEVIER

[About Elsevier](#)
[Products & Solutions](#)
[Services](#)
[Shop & Discover](#)


[Home](#) > [About](#) > [Policies](#) > [Copyright](#)

Copyright

Describes the rights related to the publication and distribution of research. It governs how authors (as well as their employers or funders), publishers and the wider general public can use, publish and distribute articles or books.

[Journal author rights](#)
[Government employees](#)
[Elsevier's rights](#)
[Protecting author rights](#)
[Open access](#)

Journal author rights

In order for Elsevier to publish and disseminate research articles, we need publishing rights. This is determined by a publishing agreement between the author and Elsevier. This agreement deals with the transfer or license of the copyright to Elsevier and authors retain significant rights to use and share their own published articles. Elsevier supports the need for authors to share, disseminate and maximize the impact of their research and these rights. In Elsevier proprietary journals* are defined below:

For subscription articles	For open access articles
<p>Authors transfer copyright to the publisher as part of a journal publishing agreement, but have the right to:</p> <ul style="list-style-type: none"> Share their article for Personal Use, Internal Institutional Use and Scholarly Sharing purposes, with a DOI link to the version of record on ScienceDirect (and with the Creative Commons CC-BY-NC-ND license for author manuscript versions) Retain patent, trademark and other Intellectual property rights (including research data). Proper attribution and credit for the published work. 	<p>Authors sign an exclusive license agreement, where authors have copyright but license exclusive rights in their article to the publisher**. In this case authors have the right to:</p> <ul style="list-style-type: none"> Share their article in the same ways permitted to third parties under the relevant user license (together with Personal Use rights) so long as it contains a CrossMark logo, the end user license, and a DOI link to the version of record on ScienceDirect. Retain patent, trademark and other Intellectual property rights (including research data). Proper attribution and credit for the published work.

*Please note that society or third party owned journals may have different publishing agreements. Please see the [Journal's guide for authors for journal specific copyright information](#).

**This includes the right for the publisher to make and authorize [commercial use](#), please see ["Rights granted to Elsevier"](#) for more details.

Help and Support

- Download a sample publishing agreement for subscription articles in [English](#) and [French](#).
- Download a sample publishing agreement for open access articles for authors choosing a [commercial user license](#) and [non-commercial user license](#).
- For authors who wish to self-archive see our [sharing guidelines](#)
- See our [author pages](#) for further details about how to promote your article.
- For use of Elsevier material not defined below please see our [permissions page](#) or visit the [Permissions Support Center](#) ».

Appendix C

ACM LICENSE DOCUMENTS



ACM (Association for Computing Machinery) - License Terms and Conditions

This is a License Agreement between Ta-Yuan Hsu ("You") and ACM (Association for Computing Machinery) ("Publisher") provided by Copyright Clearance Center ("CCC"). The license consists of your order details, the terms and conditions provided by ACM (Association for Computing Machinery), and the CCC terms and conditions.

All payments must be made in full to CCC.

Order Date	21-Oct-2020	Type of Use	Republish in a thesis/dissertation
Order license ID	1071939-1	Portion	Chapter/article
ISBN-13	978-1-4503-6894-0		

LICENSED CONTENT

Publication Title	Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing	Rightsholder	ACM (Association for Computing Machinery)
Article Title	A Proactive, Cost-aware, Optimized Data Replication Strategy in Geo-distributed Cloud Datastores	Publication Type	Conference Proceeding
Date	12/31/2018	Start Page	143
Language	English	End Page	153

REQUEST DETAILS

Portion Type	Chapter/article	Rights Requested	Main product
Page range(s)	143-153	Distribution	Worldwide
Total number of pages	10	Translation	Original language of publication
Format (select all that apply)	Print, Electronic	Copies for the disabled?	No
Who will republish the content?	Academic institution	Minor editing privileges?	No
Duration of Use	Life of current and all future editions	Incidental promotional use?	No
Lifetime Unit Quantity	Up to 9,999	Currency	USD

NEW WORK DETAILS

Title	Cost-Effective Protocols for Enforcing Causal Consistency in Geo-Replicated Data Store Systems	Institution name	University of Illinois at Chicago
Instructor name	Ajay D. Kshemkalyani	Expected presentation date	2020-12-31

ADDITIONAL DETAILS

Order reference number	N/A	The requesting person / organization to appear on the license	Ta-Yuan Hsu
------------------------	-----	---	-------------

REUSE CONTENT DETAILS

Title, description or numeric reference of the portion(s)	Chapter 4, pages 93-127	Title of the article/chapter the portion is from	A Proactive, Cost-aware, Optimized Data Replication Strategy in Geo-distributed Cloud Datastores
Editor of portion(s)	Hsu, Ta-Yuan; Kshemkalyani, Ajay D.	Author of portion(s)	Hsu, Ta-Yuan; Kshemkalyani, Ajay D.
Volume of serial or monograph	N/A	Issue, if republishing an article from a serial	N/A
Page or page range of portion	143-153	Publication date of portion	2019-12-01

Appendix C (Continued)



ACM (Association for Computing Machinery) - License Terms and Conditions

This is a License Agreement between Ta-Yuan Hsu ("You") and ACM (Association for Computing Machinery) ("Publisher") provided by Copyright Clearance Center ("CCC"). The license consists of your order details, the terms and conditions provided by ACM (Association for Computing Machinery), and the CCC terms and conditions.

All payments must be made in full to CCC.

Order Date	21-Oct-2020	Type of Use	Republish in a thesis/dissertation
Order license ID	1071941-1	Publisher	ACM
ISBN-13	978-1-4503-4227-8	Portion	Chapter/article

LICENSED CONTENT

Publication Title	Proceedings of the Third International Workshop on Adaptive Resource Management and Scheduling for Cloud Computing	Rights holder	ACM (Association for Computing Machinery)
Article Title	Performance of Approximate Causal Consistency for Partially Replicated Systems	Publication Type	Conference Proceeding
Date	12/31/2015	Start Page	7
Language	English	End Page	13
Country	United States of America		

REQUEST DETAILS

Portion Type	Chapter/article	Rights Requested	Main product
Page range(s)	7-13	Distribution	Worldwide
Total number of pages	6	Translation	Original language of publication
Format (select all that apply)	Print, Electronic	Copies for the disabled?	No
Who will republish the content?	Academic institution	Minor editing privileges?	No
Duration of Use	Life of current and all future editions	Incidental promotional use?	No
Lifetime Unit Quantity	Up to 9,999	Currency	USD

NEW WORK DETAILS

Title	Cost-Effective Protocols for Enforcing Causal Consistency in Geo-Replicated Data Store Systems	Institution name	University of Illinois at Chicago
Instructor name	Ajay D. Kshemkalyani	Expected presentation date	2020-12-31

ADDITIONAL DETAILS

Order reference number	N/A	The requesting person / organization to appear on the license	Ta-Yuan Hsu
------------------------	-----	---	-------------

REUSE CONTENT DETAILS

Title, description or numeric reference of the portion(s)	Chapter 3, pages 66-92	Title of the article/chapter the portion is from	Performance of Approximate Causal Consistency for Partially Replicated Systems
Editor of portion(s)	Kshemkalyani, Ajay D.; Hsu, Ta-yuan	Author of portion(s)	Kshemkalyani, Ajay D.; Hsu, Ta-yuan
Volume of serial or monograph	N/A	Issue, if republishing an article from a serial	N/A
Page or page range of portion	7-13	Publication date of portion	2015-12-31

Appendix C (Continued)



ACM (Association for Computing Machinery) - License Terms and Conditions

This is a License Agreement between Ta-Yuan Hsu ("You") and ACM (Association for Computing Machinery) ("Publisher") provided by Copyright Clearance Center ("CCC"). The license consists of your order details, the terms and conditions provided by ACM (Association for Computing Machinery), and the CCC terms and conditions.

All payments must be made in full to CCC.

Order Date	22-Oct-2020	Type of Use	Republish in a thesis/dissertation
Order license ID	1071947-1	Publisher	ACM
ISBN-13	978-1-4503-2928-6	Portion	Chapter/article

LICENSED CONTENT

Publication Title	Proceedings of the 2015 International Conference on Distributed Computing and Networking	Rightholder	ACM (Association for Computing Machinery)
Article Title	OPCAM : Optimal Algorithms Implementing Causal Memories in Shared Memory Systems	Publication Type	Conference Proceeding
Date	12/31/2014	Start Page	1
Language	English	End Page	4
Country	United States of America		

REQUEST DETAILS

Portion Type	Chapter/article	Rights Requested	Main product
Page range(s)	1-4	Distribution	Worldwide
Total number of pages	4	Translation	Original language of publication
Format (select all that apply)	Print, Electronic	Copies for the disabled?	No
Who will republish the content?	Academic institution	Minor editing privileges?	No
Duration of Use	Life of current and all future editions	Incidental promotional use?	No
Lifetime Unit Quantity	Up to 9,999	Currency	USD

NEW WORK DETAILS

Title	Cost-Effective Protocols for Enforcing Causal Consistency in Geo-Replicated Data Store Systems	Institution name	University of Illinois at Chicago
Instructor name	Ajay D. Kshemkalyani	Expected presentation date	2020-12-31

ADDITIONAL DETAILS

Order reference number	N/A	The requesting person / organization to appear on the license	Ta-Yuan Hsu
------------------------	-----	---	-------------

REUSE CONTENT DETAILS

Title, description or numeric reference of the portion(s)	Chapter 2, pages 15-65	Title of the article/chapter the portion is from	OPCAM : Optimal Algorithms Implementing Causal Memories in Shared Memory Systems
Editor of portion(s)	Hsu, Ta-yuan; Kshemkalyani, Ajay D.; Shen, Min	Author of portion(s)	Hsu, Ta-yuan; Kshemkalyani, Ajay D.; Shen, Min
Volume of serial or monograph	N/A	Issue, if republishing an article from a serial	N/A
Page or page range of portion	1-4	Publication date of portion	2014-12-31

Appendix C (Continued)



ACM (Association for Computing Machinery) - License Terms and Conditions

This is a License Agreement between Ta-Yuan Hsu ("You") and ACM (Association for Computing Machinery) ("Publisher") provided by Copyright Clearance Center ("CCC"). The license consists of your order details, the terms and conditions provided by ACM (Association for Computing Machinery), and the CCC terms and conditions.

All payments must be made in full to CCC.

Order Date	22-Oct-2020	Type of Use	Republish in a thesis/dissertation
Order license ID	1071959-1	Publisher	ACM
ISBN-13	978-1-4503-4037-3	Portion	Chapter/article

LICENSED CONTENT

Publication Title	Proceedings of the Fifth International Workshop on Network-Aware Data Management	Rightholder	ACM (Association for Computing Machinery)
Article Title	Approximate causal consistency for partially replicated geo-replicated cloud storage	Publication Type	Conference Proceeding
Date	12/31/2014	Start Page	1
Language	English	End Page	8
Country	United States of America		

REQUEST DETAILS

Portion Type	Chapter/article	Rights Requested	Main product
Page range(s)	1-8	Distribution	Worldwide
Total number of pages	8	Translation	Original language of publication
Format (select all that apply)	Print, Electronic	Copies for the disabled?	No
Who will republish the content?	Academic institution	Minor editing privileges?	No
Duration of Use	Life of current and all future editions	Incidental promotional use?	No
Lifetime Unit Quantity	Up to 9,999	Currency	USD

NEW WORK DETAILS

Title	Cost-Effective Protocols for Enforcing Causal Consistency in Geo-Replicated Data Store Systems	Institution name	University of Illinois at Chicago
Instructor name	Ajay D. Kshemkalyani	Expected presentation date	2020-12-31

ADDITIONAL DETAILS

Order reference number	N/A	The requesting person / organization to appear on the license	Ta-Yuan Hsu
------------------------	-----	---	-------------

REUSE CONTENT DETAILS

Title, description or numeric reference of the portion(s)	Chapter 3, pages 66-92	Title of the article/chapter the portion is from	Approximate causal consistency for partially replicated geo-replicated cloud storage
Editor of portion(s)	Hsu, Ta-yuan; Kshemkalyani, Ajay D.	Author of portion(s)	Hsu, Ta-yuan; Kshemkalyani, Ajay D.
Volume of serial or monograph	N/A	Issue, if republishing an article from a serial	N/A
Page or page range of portion	1-8	Publication date of portion	2014-12-31

CITED LITERATURE

1. Hsu, T.-Y., Kshemkalyani, A., and Shen, M.: Causal consistency algorithms for partially replicated and fully replicated systems. *Future Generation Computer Systems* , 86:1118 – 1133, 2018.
2. Hsu, T. and Kshemkalyani, A. D.: Value the recent past: Approximate causal consistency for partially replicated systems. *IEEE Transactions on Parallel and Distributed Systems* , 29(1):212–225, 2018.
3. Hsu, T.-Y. and Kshemkalyani, A. D.: A proactive, cost-aware, optimized data replication strategy in geo-distributed cloud datastores. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing* , UCC’19, page 143–153, New York, NY, USA, 2019. Association for Computing Machinery.
4. Iliadis, I., Sotnikov, D., Ta-Shma, P., and Venkatesan, V.: Reliability of geo-replicated cloud storage systems. In *Dependable Computing (PRDC), 2014 IEEE 20th Pacific Rim International Symposium on* , pages 169–179, Nov 2014.
5. Kshemkalyani, A. D. and Singhal, M.: *Distributed Computing: Principles, Algorithms, and Systems* . New York, NY, USA, Cambridge University Press, 1 edition, 2011.
6. Lloyd, W., Freedman, M. J., Kaminsky, M., and Andersen, D. G.: Don’t settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* , SOSP ’11, pages 401–416, New York, NY, USA, 2011. ACM.
7. Gilbert, S. and Lynch, N.: Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* , 33(2):51–59, June 2002.
8. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Voshall, P., and Vogels, W.: Dynamo: Amazon’s highly available key-value store. *SIGOPS Oper. Syst. Rev.* , 41(6):205–220, October 2007.
9. Mahajan, P., Alvisi, L., and Dahlin, M.: Consistency, Availability, and Convergence. University of Texas at Austin TR-11-22, 2011.

10. Raynal, M., Schiper, A., and Toueg, S.: The causal ordering abstraction and a simple way to implement it. *Inf. Process. Lett.* , 39(6):343–350, October 1991.
11. Kshemkalyani, A. D. and Singhal, M.: An optimal algorithm for generalized causal message ordering. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing* , PODC '96, pages 87–, New York, NY, USA, 1996. ACM.
12. Kshemkalyani, A. D. and Singhal, M.: Necessary and sufficient conditions on information for causal message ordering and their optimal implementation. *Distributed Computing* , 11(2):91–111, 1998.
13. Baldoni, R., Milani, A., and Piergiovanni, S. T.: Optimal propagation-based protocols implementing causal memories. *Distributed Computing* , 18(6):461–474, 2006.
14. Tang, Y., Sun, H., Wang, X., and Liu, X.: Achieving convergent causal consistency and high availability for cloud storage. *Future Generation Computer Systems* , 74:20 – 31, 2017.
15. Ahamad, M., Neiger, G., Burns, J., Kohli, P., and Hutto, P.: Causal memory: Definitions, implementation and programming. *Distributed Computing* , 9(1):37–49, 1995.
16. Petersen, K., Spreitzer, M., Terry, D. B., Theimer, M., and Demers, A. J.: Flexible update propagation for weakly consistent replication. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles, SOSP 1997, St. Malo, France, October 5-8, 1997* , pages 288–301, 1997.
17. Belaramani, N., Dahlin, M., Gao, L., Nayate, A., Venkataramani, A., Yalagandula, P., and Zheng, J.: Practi replication. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3* , NSDI'06, pages 5–5, Berkeley, CA, USA, 2006. USENIX Association.
18. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* , 21(7):558–565, July 1978.
19. Mahajan, P., Alvisi, L., and Dahlin, M.: Consistency, availability, and convergence. Technical Report UTCS TR-11-22, Dept. of Comp. Sc., The U. of Texas at Austin, Austin, TX, USA, 2011.
20. Ladin, R., Liskov, B., Shrira, L., and Ghemawat, S.: Providing high availability using lazy replication. *ACM. Trans. Comput. Syst.* , 10(4):360–391, November 1992.

21. Lloyd, W., Freedman, M. J., Kaminsky, M., and Andersen, D. G.: Stronger semantics for low-latency geo-replicated storage. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* , pages 313–328, Lombard, IL, 2013. USENIX.
22. Du, J., Elnikety, S., Roy, A., and Zwaenepoel, W.: Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing* , SOCC '13, pages 11:1–11:14, New York, NY, USA, 2013. ACM.
23. Du, J., Iorgulescu, C., Roy, A., and Zwaenepoel, W.: Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, November 03 - 05, 2014* , pages 4:1–4:13, 2014.
24. Akkoorath, D. D., Tomsic, A. Z., Bravo, M., Li, Z., Crain, T., Bieniusa, A., Pregoça, N., and Shapiro, M.: Cure: Strong semantics meets high availability and low latency. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)* , pages 405–414, 2016.
25. Almeida, S., Leitão, J. a., and Rodrigues, L.: Chainreaction: A causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems* , EuroSys '13, pages 85–98, New York, NY, USA, 2013. ACM.
26. Bailis, P., Ghodsi, A., Hellerstein, J. M., and Stoica, I.: Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* , SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM.
27. Mehdi, S. A., Little, C., Crooks, N., Alvisi, L., Bronson, N., and Lloyd, W.: I can't believe it's not causal! scalable causal consistency with no slowdown cascades. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* , pages 453–468,
28. Spirovska, K., Didona, D., and Zwaenepoel, W.: Optimistic causal consistency for geo-replicated key-value stores. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)* , pages 2626–2629, 2017.
29. Didona, D., Guerraoui, R., Wang, J., and Zwaenepoel, W.: Causal consistency and latency optimality: Friend or foe? *Proc. VLDB Endow.* , 11(11):1618–1632, July 2018.

30. Roohitavaf, M., Demirbas, M., and Kulkarni, S.: Causalspartan: Causal consistency for distributed data stores using hybrid logical clocks. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)* , pages 184–193, 2017.
31. Lu, H., Hodsdon, C., Ngo, K., Mu, S., and Lloyd, W.: The snow theorem and latency-optimal read-only transactions. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* , OSDI’16, page 135–150, USA, 2016. USENIX Association.
32. Gunawardhana, C., Bravo, M., and Rodrigues, L.: Unobtrusive deferred update stabilization for efficient geo-replication. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* , pages 83–95, Santa Clara, CA, July 2017. USENIX Association.
33. Bravo, M., Rodrigues, L., and Van Roy, P.: Saturn: A distributed metadata service for causal consistency. In *Proceedings of the Twelfth European Conference on Computer Systems* , EuroSys ’17, page 111–126, New York, NY, USA, 2017. Association for Computing Machinery.
34. Fouto, P., Leitão, J., and Pregoça, N.: Practical and fast causal consistent partial geo-replication. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)* , pages 1–10, 2018.
35. Mahmood, T., PN, S., Rao, S., Vijaykumar, T., and Thottethodi, M.: Karma: Cost-effective geo-replicated cloud storage with dynamic enforcement of causal consistency. *IEEE Transactions on Cloud Computing* , pages 1–1, 06 2018.
36. Xiang, Z. and Vaidya, N. H.: Global stabilization for causally consistent partial replication. *CoRR* , abs/1803.05575, 2018.
37. Zawirski, M., Pregoça, N., Duarte, S., Bieniusa, A., Balegas, V., and Shapiro, M.: Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference* , Middleware ’15, page 75–87, New York, NY, USA, 2015. Association for Computing Machinery.
38. Spirovska, K., Didona, D., and Zwaenepoel, W.: Paris: Causally consistent transactions with non-blocking reads and partial replication. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)* , pages 304–316, July 2019.

39. Crain, T. and Shapiro, M.: Designing a causally consistent protocol for geo-distributed partial replication. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data* , PaPoC '15, pages 6:1–6:4, New York, NY, USA, 2015. ACM.
40. Ranganathan, K. and Foster, I.: Identifying dynamic replication strategies for a high-performance data grid. In *Grid Computing — GRID 2001* , ed. C. A. Lee, pages 75–86, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
41. Rahman, R. M., Barker, K., and Alhajj, R.: Replica placement in data grid: considering utility and risk. *International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II* , 1:354–359 Vol. 1, 2005.
42. Chang, R.-S., Chang, H.-P., and Wang, Y.-T.: A dynamic weighted data replication strategy in data grids. In *2008 IEEE/ACS International Conference on Computer Systems and Applications* , pages 414–421, March 2008.
43. Wei, Q., Veeravalli, B., Gong, B., Zeng, L., and Feng, D.: Cdrm: A cost-effective dynamic replication management scheme for cloud storage cluster. In *2010 IEEE International Conference on Cluster Computing* , pages 188–196, Sep. 2010.
44. Sun, D.-W., Chang, G.-R., Gao, S., Jin, L.-Z., and Wang, X.-W.: Modeling a dynamic data replication strategy to increase system availability in cloud computing environments. *Journal of Computer Science and Technology* , 27, 03 2012.
45. Sun, D., Chang, G., Miao, C., and Wang, X.: Analyzing, modeling and evaluating dynamic adaptive fault tolerance strategies in cloud computing environments. *The Journal of Supercomputing* , 66(1):193–228, Oct 2013.
46. Gill, N. K. and Singh, S.: A dynamic, cost-aware, optimized data replication strategy for heterogeneous cloud data centers. *Future Gener. Comput. Syst.* , 65(C):10–32, December 2016.
47. Mansouri, Y. and Buyya, R.: Dynamic replication and migration of data objects with hot-spot and cold-spot statuses across storage data centers. *Journal of Parallel and Distributed Computing* , 12 2018.
48. Shankaranarayanan, P. N., Sivakumar, A., Rao, S., and Tawarmalani, M.: Performance sensitive replication in geo-distributed cloud datastores. In *2014 44th Annual*

IEEE/IFIP International Conference on Dependable Systems and Networks , pages 240–251, June 2014.

49. Mansouri, N. and Javidi, M. M.: A new prefetching-aware data replication to decrease access latency in cloud environment. *Journal of Systems and Software* , 144:197–215, 2018.
50. Rasool, Q., Li, J., and Zhang, S.: Replica placement in multi-tier data grid. In *2009 Eighth IEEE International Conference on Dependable, Autonomic and Secure Computing* , pages 103–108, Dec 2009.
51. Qu, Y. and Xiong, N.: Rfh: A resilient, fault-tolerant and high-efficient replication algorithm for distributed cloud storage. In *2012 41st International Conference on Parallel Processing* , pages 520–529, Sep. 2012.
52. Lin, Y., Wu, J., and Liu, P.: A list-based strategy for optimal replica placement in data grid systems. In *2008 37th International Conference on Parallel Processing* , pages 198–205, Sep. 2008.
53. Ling, Y., Ouyang, Y., and Luo, Z.: A novel fault-tolerant scheduling algorithm with high reliability in cloud computing systems. *Journal of Convergence Information Technology* , 7:107–115, 08 2012.
54. Bonvin, N., Papaioannou, T. G., and Aberer, K.: A self-organized, fault-tolerant and scalable replication scheme for cloud storage. In *Proceedings of the 1st ACM Symposium on Cloud Computing* , SoCC '10, pages 205–216, New York, NY, USA, 2010. ACM.
55. Nguyen, T., Cutway, A., and Shi, W.: Differentiated replication strategy in data centers. In *Network and Parallel Computing* , eds. C. Ding, Z. Shao, and R. Zheng, pages 277–288, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
56. Caron, E., Desprez, F., and Muresan, A.: Forecasting for grid and cloud computing on-demand resources based on pattern matching. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science* , pages 456–463, Nov 2010.
57. Sladescu, M., Fekete, A., Lee, K., and Liu, A.: Event aware workload prediction: A study using auction events. In *Web Information Systems Engineering - WISE 2012* , pages 368–381, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

58. Islam, S., Keung, J., Lee, K., and Liu, A.: Empirical prediction models for adaptive resource provisioning in the cloud. *Future Gener. Comput. Syst.* , 28(1):155–162, January 2012.
59. Calheiros, R. N., Masoumi, E., Ranjan, R., and Buyya, R.: Workload prediction using arima model and its impact on cloud applications’ qos. *IEEE Transactions on Cloud Computing* , 3(4):449–458, Oct 2015.
60. Shen, M., Kshemkalyani, A. D., and Hsu, T. Y.: OPCAM: optimal algorithms implementing causal memories in shared memory systems. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking, ICDCN 2015, Goa, India, January 4-7, 2015* , pages 16:1–16:4, 2015.
61. Shen, M., Kshemkalyani, A. D., and Hsu, T. Y.: Causal consistency for geo-replicated cloud storage under partial replication. In *IPDPS Workshops* , pages 509–518. IEEE, 2015.
62. H  lary, J. and Milani, A.: About the efficiency of partial replication to implement distributed shared memory. In *2006 International Conference on Parallel Processing (ICPP 2006), 14-18 August 2006, Columbus, Ohio, USA* , pages 263–270, 2006.
63. Chandra, P., Gambhire, P., and Kshemkalyani, A. D.: Performance of the optimal causal multicast algorithm: A statistical analysis. *IEEE Trans. Parallel Distrib. Syst.* , 15(1):40–52, 2004.
64. Chandra, P. and Kshemkalyani, A. D.: Causal multicast in mobile networks. In *12th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2004), 4-8 October 2004, Vollandam, The Netherlands* , pages 213–220, 2004.
65. Padhye, V., Rajappan, G., and Tripathi, A. R.: Transaction management using causal snapshot isolation in partially replicated databases. In *33rd IEEE International Symposium on Reliable Distributed Systems, SRDS 2014, Nara, Japan, October 6-9, 2014* , pages 105–114, 2014.
66. Hsu, T. Y. and Kshemkalyani, A. D.: Performance of causal consistency algorithms for partially replicated systems. In *IPDPS Workshops* , pages 525–534. IEEE, 2016.
67. Kshemkalyani, A. D. and Hsu, T.-Y.: Approximate causal consistency for partially replicated geo-replicated cloud storage. In *Proceedings of the Fifth International Work-*

shop on Network-Aware Data Management , NDM '15, pages 3:1–3:8, New York, NY, USA, 2015. ACM.

68. Hsu, T. Y. and Kshemkalyani, A. D.: Performance of approximate causal consistency for partially replicated systems. In *Workshop on Adaptive Resource Management and Scheduling for Cloud Computing (ARMS-CC)* , pages 7–13. ACM, 2016.
69. Calheiros, R. N., Ranjan, R., and Buyya, R.: Virtual machine provisioning based on analytical performance and qos in cloud computing environments. In *2011 International Conference on Parallel Processing* , pages 295–304, Sep. 2011.
70. E. P. Box, G., M. Jenkins, G., C. Reinsel, G., and Ljung, G.: *Time Series Analysis: Forecasting and Control* . Hoboken NJ, Wiley, 5 edition, 2016.
71. Urdaneta, G., Pierre, G., and van Steen, M.: Wikipedia workload analysis for decentralized hosting. *Comput. Netw.* , 53(11):1830–1845, July 2009.
72. Arlitt, M. and Jin, T.: A workload characterization study of the 1998 world cup web site. *Netwrk. Mag. of Global Internetwkg.* , 14(3):30–37, May 2000.
73. Mao, M., Li, J., and Humphrey, M.: Cloud auto-scaling with deadline and budget constraints. In *2010 11th IEEE/ACM International Conference on Grid Computing* , pages 41–48, Oct 2010.
74. Hyndman, R. and Khandakar, Y.: Automatic time series forecasting: The forecast package for r. *Journal of Statistical Software, Articles* , 27(3):1–22, 2008.
75. Kwiatkowski, D., Phillips, P., Schmidt, P., and Shin, Y.: Testing the null hypothesis of stationarity against the alternative of a unit root. how sure are we that economic time series have unit root? *Journal of Econometrics* , 54:159–178, 10 1992.
76. Li, R., Wang, S., Deng, H., Wang, R., and Chang, K. C.-C.: Towards social user profiling: unified and discriminative influence model for inferring home locations. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '12* , pages 1023–1031, 2012.
77. Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. A. F., and Buyya, R.: Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Softw. Pract. Exper.* , 41(1):23–50, January 2011.

78. Beaver, D., Kumar, S., Li, H. C., Sobel, J., and Vajgel, P.: Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* , OSDI'10, pages 47–60, 2010.
79. Bailis, P., Fekete, A., Ghodsi, A., Hellerstein, J. M., and Stoica, I.: The potential dangers of causal consistency and an explicit solution. In *Proceedings of the Third ACM Symposium on Cloud Computing* , SoCC '12, pages 22:1–22:7, New York, NY, USA, 2012. ACM.
80. Birman, K. P.: A response to cheriton and skeen's criticism of causal and totally ordered communication. *Operating Systems Review* , 28(1):11–21, 1994.
81. Birman, K., Schiper, A., and Stephenson, P.: Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.* , 9(3):272–314, August 1991.
82. Cheriton, D. R. and Skeen, D.: Understanding the limitations of causally and totally ordered communication. *SIGOPS Oper. Syst. Rev.* , 27(5):44–57, December 1993.
83. Gambhire, P. and Kshemkalyani, A. D.: Reducing false causality in causal message ordering. In *HiPC* , eds. M. Valero, V. K. Prasanna, and S. Vajapeyam, volume 1970 of *Lecture Notes in Computer Science* , pages 61–72. Springer, 2000.
84. Lady, K., Kim, M., and Noble, B. D.: 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* , pages 2–7, 2014.
85. Lloyd, W., Freedman, M. J., Kaminsky, M., and Andersen, D. G.: Don't settle for eventual consistency. *Commun. ACM* , 57(5):61–68, 2014.
86. Lloyd, W., Freedman, M. J., Kaminsky, M., and Andersen, D. G.: Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* , nsdi'13, pages 313–328, Berkeley, CA, USA, 2013. USENIX Association.

VITA

NAME	Ta-Yuan Hsu
EDUCATION	Ph.D., Electrical and Computer Engineering, University of Illinois at Chicago, Chicago, Illinois, United States, 2020 M.Sc., Computer Science, Illinois Institute of Technology, Chicago, Illinois, United States, 2010 B.Sc., Physics, National Central University, Taoyuan, R.O.C., 2003
EXPERIENCE	IT Intern, USAIRC�, Taipei, R.O.C. IT analyst, CMP Complex Security Center, Taoyuan, R.O.C. Graduate Research Assistant, University of Illinois at Chicago Teaching Assistant, University of Illinois at Chicago
PUBLICATIONS	Ta-Yuan Hsu and Ajay D. Kshemkalyani: A proactive, cost-aware, optimized data replication strategy in geo-distributed cloud datastores. In <i>Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing, UCC’19</i> , page 143–153, New York, NY, USA, 2019. Association for Computing Machinery. Ta-Yuan Hsu and Ajay D. Kshemkalyani: Value the recent past: Approximate causal consistency for partially replicated systems. <i>IEEE Transactions on Parallel and Distributed Systems</i> , 29(1):212–225, 2018. Ta-Yuan Hsu , Ajay D. Kshemkalyani, and Min Shen: Causal consistency algorithms for partially replicated and fully replicated systems. <i>Future Generation Computer Systems</i> , 86:1118 – 1133, 2018. Ta-Yuan Hsu and Ajay D. Kshemkalyani: Performance of approximate causal consistency for partially replicated systems. In <i>Workshop on Adaptive Resource Management and Scheduling for Cloud Computing (ARMS-CC)</i> , pages 7–13. ACM, 2016.

Ta-Yuan Hsu and Ajay D. Kshemkalyani: Performance of causal consistency algorithms for partially replicated systems. In *IPDPS Workshops*, pages 525–534. IEEE, 2016.

Ajay D. Kshemkalyani and **Ta-Yuan Hsu**: Approximate causal consistency for partially replicated geo-replicated cloud storage. In *Proceedings of the Fifth International Workshop on Network-Aware Data Management*, NDM '15, pages 3:1–3:8, New York, NY, USA, 2015. ACM.

Min Shen, Ajay D. Kshemkalyani, and **Ta-Yuan Hsu**: Causal consistency for geo-replicated cloud storage under partial replication. In *IPDPS Workshops*, pages 509–518. IEEE, 2015.

Min Shen, Ajay D. Kshemkalyani, and **Ta-Yuan Hsu**: OPCAM: optimal algorithms implementing causal memories in shared memory systems. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking, ICDCN 2015, Goa, India, January 4-7, 2015*, pages 16:1–16:4, 2015.

Ta-Yuan Hsu, Ajay D. Kshemkalyani, Modeling Social Network Topology with Variable Social Vector Clocks, In *2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM) 2015*: 584-589

Ta-Yuan Hsu, A. D. Kshemkalyani, Variable social vector clocks for exploring user interactions in social communication networks. In *International Journal of Space-Based and Situated Computing (IJSSC)* 5(1): 39-52 (2015)

Ta-Yuan Hsu, A. D. Kshemkalyani, M. Shen, Modeling User Interactions in Social Communication Networks with Variable Social Vector Clocks. In *2014 IEEE 28th International Conference on Advanced Information Networking and Applications Workshops*, Victoria, BC, 2014, pp. 96-101.

HONORS

The Fifty for the Future Award from the Illinois Technology Foundation. (2018)