Consistency Models in Distributed Systems with Physical Clocks

THÈSE N^{O} 6318 (2014)

PRÉSENTÉE LE 10 OCTOBRE 2014

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE DE SYSTÈMES D'EXPLOITATION

PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Jiaqing DU

acceptée sur proposition du jury:

Prof. S. Vaudenay, président du jury
Prof. W. Zwaenepoel, directeur de thèse
Dr S. M. Elnikety, rapporteur
Prof. R. Guerraoui, rapporteur
Prof. M. Shapiro, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2014

Abstract

Most existing distributed systems use logical clocks to order events in the implementation of various consistency models. Although logical clocks are straightforward to implement and maintain, they may affect the scalability, availability, and latency of the system when being used to totally order events in strong consistency models. They can also incur considerable overhead when being used to track and check the causal relationships among events in some weak consistency models.

In this thesis we explore how to efficiently implement different consistency models using loosely synchronized physical clocks. Compared with logical clocks, physical clocks move forward at approximately the same speed and can be loosely synchronized with well-known standard protocols. Hence a group of physical clocks located at different servers can be used to order events in a distributed system at very low cost.

We first describe Clock-SI, a fully distributed implementation of snapshot isolation for partitioned data stores. It uses the local physical clock at each partition to assign snapshot and commit timestamps to transactions. By avoiding a centralized service for timestamp management, Clock-SI improves the throughput, latency, and availability of the system.

We then introduce Clock-RSM, which is a low-latency state machine replication protocol that provides linearizability. It totally orders state machine commands by assigning them physical timestamps obtained from the local replica. By eliminating the message step for command ordering in existing solutions, Clock-RSM reduces the latency of consistent geo-replication across multiple data centers.

Finally, we present Orbe, which provides an efficient and scalable implementation of causal consistency for both partitioned and replicated data stores. Orbe builds an explicit total order, consistent with causality, among all operations using physical timestamps. It reduces the number of dependencies that have to be carried in update replication messages and checked on installation of replicated updates. As a result, Orbe improves the throughput of the system.

Keywords : distributed systems, snapshot isolation, linearizability, causal consistency

Résumé

La plupart des systèmes distribués existants utilisent des horloges logiques pour ordonner les événements dans l'implémentation de différents modèles de cohérence. Bien que les horloges logiques sont simples à implémenter et à entretenir, elles peuvent affecter l'adaptabilité, la disponibilité et le temps de latence du système lorsqu'il est utilisé pour ordonner les événements dans des modèles de cohérence solides. Elles peuvent aussi engager des dépenses considérables lorsqu'elles sont utilisées pour suivre et vérifier les relations de causalité entre les événements dans certains modèles de cohérence faibles.

Dans cette thèse, nous explorons la façon d'implémenter efficacement différents modèles de cohérence en utilisant des horloges physiques. Par rapport aux horloges logiques, les horloges physiques avancent automatiquement à la même vitesse et peuvent être ainsi synchronisées avec des protocoles standards largement utilisés dans la pratique. Ainsi, un groupe d'horloges physiques situées dans différents serveurs peut être utilisé pour ordonner les événements d'un système distribué à un coût très faible.

Nous décrivons d'abord Clock-SI, une implémentation entièrement distribuée de l'isolement de cliché pour les banques de données partitionnées. Elle utilise l'horloge physique locale à chaque partition afin d'affecter un cliché et d'attribuer des horodatages à des transactions. En évitant un service centralisé pour la gestion d'horodatage, Clock-SI améliore le débit, la latence et la disponibilité du système.

Nous présentons ensuite Clock-RSM qui est une machine d'état à protocole de réplication à faible latence et qui permet l'atomicité. Elle ordonne les commandes de la machine d'état en leurs attribuant des horodateurs physiques obtenus à partir de la réplique locale. En éliminant l'étape de message pour l'ordonnancement des commandes dans les solutions existantes, Clock-RSM réduit la latence de la géo-réplication à travers plusieurs centres de données.

Enfin, nous présentons Orbe qui fournit une implémentation efficace et évolutive de consistance causale pour les banques de données partitionnées mais aussi ré-

pliquées. Orbe construit un ordre total explicite entre toutes les opérations à l'aide de l'horodatage physiques, et l'ordre est consistant avec la causalité. Il réduit le nombre de dépendances prises en compte lors de la mise-à-jour des messages de réplications et vérifiés lors de l'installation de mise-à-jours répliquées. Par conséquent, Orbe permet d'améliorer le débit du système.

Mots-clés : systèmes distribués, isolement de cliché, atomicité, consistance causale

Acknowledgements

This thesis is the culmination of an interesting, albeit long and challenging, journey. It would not be possible without the help of many people. I am extremely grateful to all of them.

I would like to thank my advisor, Willy Zwaenepoel, for his patient support and guidance over the course of my PhD. Willy gave me the freedom to explore interesting problems in computer systems. He was always open to all of my research ideas and guided me through the challenging parts. I achieved much more than I expected because of his persistence on upholding high standards.

I would like to thank Sameh Elnikety for working with me on interesting research projects that eventually lead to this thesis. Sameh invested significant efforts in my work and stayed up late in a different time zone to work on paper deadlines with me. I learned a lot about distributed systems from him through our collaboration.

Over the years of my PhD, I have been fortunate to collaborate with some talented people. I would like to thank Amitabha Roy, Daniele Sciascia, Fernando Pedone, Nipun Sehrawat, and Calin Iorgulescu for working with me.

I would like to thank my jury members, Sameh Elnikety, Rachid Guerraoui, and Marc Shapiro, for reading this thesis and providing constructive feedback. I would like to thank Serge Vaudenay for serving as the president of my jury.

Thanks also go to the current and previous members of LABOS: Mihai Dobrescu, Katerina Argyraki, Laurent Bindschaedler, Pamela Delgado, Florin Dinu, Dan Dumitriu, Denisa Ghita, Calin Iorgulescu, Ming-Yee Iu, Jasmina Malicevic, Aravind Menon, Ivo Mihailovic, Amitabha Roy, and Simon Schubert. I really enjoyed the fun discussions with them on almost everything during the lunches and coffee breaks.

I would like to thank my parents for their unconditional love and support throughout my entire education. They sacrificed a lot to give me a good education. I am very proud to be their son. Last but foremost, I would like to thank my girlfriend Chunqing for her tireless support and persistent encouragement. I am extremely lucky to have her by my side over the years. My journey is more colorful because of her!

Contents

Abstract iii

Acknowledgements vii

List of figures xii

List of tables xv

1 Introduction 1
- 1.1 Background and Motivation . . . 1
- 1.2 Consistency Models . . . 3
  - 1.2.1 Consistency of Replicated Data Stores . . . 3
  - 1.2.2 Transaction Isolation Levels . . . 5
  - 1.2.3 Summary . . . 6
- 1.3 Problem Statement . . . 6
  - 1.3.1 Snapshot Isolation for Partitioned Data Stores . . . 6
  - 1.3.2 State Machine Replication across Data Centers . . . 7
  - 1.3.3 Scalable Causally Consistent Data Replication . . . 7
- 1.4 Solution Overview . . . 8
  - 1.4.1 Clock-SI . . . 8
  - 1.4.2 Clock-RSM . . . 9
  - 1.4.3 Orbe . . . 9
- 1.5 Thesis Organization . . . 10

2 Clock-SI: Snapshot Isolation for Partitioned Data Stores 11
- 2.1 Introduction . . . 11
  - 2.1.1 SI Overview . . . 11
  - 2.1.2 Problem Statement . . . 12
  - 2.1.3 Solution Overview . . . 12
- 2.2 Model and Definition . . . 14

2.2.1 System Model . . . 14
2.2.2 SI Definition . . . 14
2.3 Challenges . . . 15
2.4 Clock-SI . . . 16
2.4.1 Read Protocol . . . 16
2.4.2 Commit Protocol . . . 19
2.4.3 Choosing Older Snapshots . . . 20
2.4.4 Correctness . . . 21
2.4.5 Discussion . . . 22
2.5 Analytical Model . . . 23
2.5.1 Model Parameters . . . 23
2.5.2 Delay due to Pending Commit . . . 24
2.5.3 Delay due to Clock Skew . . . 25
2.5.4 Update Transaction Abort Probability . . . 26
2.5.5 Example . . . 26
2.6 Evaluation . . . 27
2.6.1 Implementation and Setup . . . 27
2.6.2 Micro-benchmarks . . . 28
2.6.3 Twitter Feed-Following Benchmark . . . 31
2.6.4 Effects of Taking Older Snapshots . . . 31
2.6.5 Model Verification . . . 32
2.6.6 NTP Precision . . . 34
2.7 Summary . . . 37

3 Clock-RSM: Low-Latency Inter-Datacenter State Machine Replication 39
3.1 Introduction . . . 39
3.1.1 Problem Statement . . . 39
3.1.2 Solution Overview . . . 40
3.2 Model and Definition . . . 41
3.2.1 System Model . . . 41
3.2.2 State Machine Replication . . . 41
3.2.3 Geo-Replication . . . 42
3.3 Clock-RSM . . . 42
3.3.1 Protocol States . . . 42
3.3.2 Protocol Execution . . . 43
3.3.3 Extension . . . 45

3.4 Latency Analysis . . . 46
3.4.1 Clock-RSM . . . 46
3.4.2 Paxos . . . 48
3.4.3 Mencius . . . 48
3.4.4 Intuition and Comparison . . . 49
3.5 Failure Handling . . . 51
3.5.1 Reconfiguration . . . 51
3.5.2 Recovery and reintegration . . . 53
3.5.3 Discussion . . . 53
3.6 Evaluation . . . 54
3.6.1 Implementation . . . 54
3.6.2 Latency in Wide Area Replication . . . 55
3.6.3 Numerical Comparison of Latency . . . 61
3.6.4 Throughput on A Local Cluster . . . 62
3.7 Correctness . . . 64
3.8 Summary . . . 66

4 Orbe: Scalable Causal Consistency 67
4.1 Introduction . . . 67
4.1.1 Problem Statement . . . 67
4.1.2 Solution Overview . . . 68
4.2 Model and Definition . . . 69
4.2.1 Architecture . . . 69
4.2.2 Causal Consistency . . . 70
4.3 DM Protocol . . . 71
4.3.1 Definitions . . . 71
4.3.2 Protocol . . . 73
4.3.3 Cost over Eventual Consistency . . . 75
4.3.4 Conflict Detection and Resolution . . . 75
4.4 DM-Clock Protocol . . . 76
4.4.1 Read-Only Transaction . . . 76
4.4.2 Definitions . . . 77
4.4.3 Protocol . . . 77
4.4.4 Garbage Collection . . . 79
4.5 Failure Handling . . . 79
4.5.1 DM Protocol . . . 79

4.5.2 DM-Clock Protocol 80
4.6 Dependency Cleaning 81
4.6.1 Intuition 81
4.6.2 Protocol Extension 82
4.6.3 Message Overhead 82
4.7 Evaluation 83
4.7.1 Implementation and Setup 83
4.7.2 Microbenchmarks 84
4.7.3 Scalability 85
4.7.4 Comparison with Eventual Consistency 86
4.7.5 Comparison with COPS 88
4.7.6 Dependency Cleaning 90
4.8 Summary 93

5 Related Work **95**
5.1 Distributed Transactions 95
5.2 State Machine Replication 97
5.3 Causal Consistency 98
5.4 Physical Clocks in Distributed Systems 99

6 Conclusion **101**

Bibliography **108**

Curriculum Vitae **109**

List of Figures

2.1 Snapshot unavailability due to clock skew. 15
2.2 Snapshot unavailability due to the pending commit of an update transaction. 16
2.3 Latency distribution of read-only transactions in a LAN. 29
2.4 Latency distribution of read-only transactions in a WAN. Partitions are in data centers in Europe, US West and US East. Clients are in our local cluster in Europe. 29
2.5 Throughput comparison for single-partition read-only and update-only transactions. Each transaction reads/updates eight items. 30
2.6 Throughput of Twitter Feed-Following application. 90% read-tweets and 10% post-tweets. 32
2.7 Read transaction throughput with write hot spots. 33
2.8 Read transaction latency with write hot spots. 33
2.9 Transaction delay rate when $\Delta = 0$ while varying the total number of data items. 34
2.10 Transaction delay rate in a small data set while varying Δ, the snapshot age. 35
2.11 Transaction delay time in a small data set while varying Δ, the snapshot age. 35
2.12 Transaction abort rate in a small data set while varying Δ, the snapshot age. 36
2.13 Distribution of the clock skew between two clocks plus one-way network latency in LAN. 36

3.1 Average (bars) and 95%ile (lines atop bars) commit latency at each of five replicas. Workload is balanced. Leader is placed at CA. 56
3.2 Average (bars) and 95%ile (lines atop bars) commit latency at each of five replicas. Workload is balanced. Leader is placed at VA. 56

3.3 Average (bars) and 95%ile (lines atop bars) commit latency at each of three replicas. Workload is balanced. Leader is placed at CA. 57
3.4 Average (bars) and 95%ile (lines atop bars) commit latency at each of three replicas. Workload is balanced. Leader is placed at VA. 58
3.5 Latency distribution at JP with five replicas. The leader is at CA. Workload is balanced. 59
3.6 Latency distribution at CA with three replicas. The leader is at VA. Workload is balanced. 59
3.7 Average (bars) and 95%ile (lines atop bars) commit latency at each of five replicas. The leader of Paxos and Paxos-bcast is at CA. Workload is imbalanced. 60
3.8 Latency distribution at SG with five replicas. The leader of Paxos and Paxos-bcast is at CA. Workload is imbalanced. 60
3.9 Average commit latency. *all* includes latencies at all replicas of a group while *highest* only includes the latency at one replica that provides the highest latency. 61
3.10 Throughput for small (10B), medium (100B), and large (1000B) commands with five replicas on a local cluster. 63

4.1 System architecture. The data set is replicated by multiple data centers. Clients are collocated with the data store in the data center and are used by the application tier to access the data store. 70
4.2 An example of the DM protocol with two partitions replicated at two data centers. A client updates item X and then item Y at two partitions. X and Y are propagated concurrently, but their installation at the remote data center is constrained by causality. 74
4.3 Latency distribution of client operations. 85
4.4 Maximum throughput of varied workloads with two to eight partitions. The legend gives the put:get ratio. 86
4.5 Maximum throughput of workloads with varied put:get ratios for both Orbe (causal consistency) and eventual consistency. 87
4.6 Average number of dependency checking messages per replicated update and percentage of dependency metadata in the update replication traffic with varied put:get ratios in Orbe. 87
4.7 Maximum throughput of operations with varied inter-operation delays for both Orbe and COPS. The legend gives the put:get ratio. 89

4.8 Average number of dependencies for each PUT operation with varied inter-operation delays for both Orbe and COPS. The legend gives the put:get ratio. 89
4.9 Aggregated replication transmission traffic across data centers with varied inter-operation delays for both Orbe and COPS. The legend gives the put:get ratio. 90
4.10 CPU utilization of a server that runs a group of clients with varied inter-operation delays for both Orbe and COPS. The legend gives the put:get ratio. 91
4.11 Maximum throughput of operations with and without dependency cleaning enabled. 91
4.12 Aggregated replication transmission traffic with and without dependency cleaning enabled. 92
4.13 Average percentage of dependency metadata in update replication traffic with and without dependency cleaning enabled. 92

List of Tables

3.1 Definition of symbols used in Algorithm 3. 43
3.2 Number of message steps, message complexity, and command commit latency of Paxos, Paxos-bcast, Mencius-bcast, and Clock-RSM. 49
3.3 Average round-trip latencies (ms) between EC2 data centers. CA, VA, EU, JP, SG, and BR correspond to California, Virginia, Ireland, Japan (Tokyo), Singapore, Australia, and Brazil (São Paulo), respectively. 55
3.4 Latency reduction of Clock-RSM over Paxos-bcast. Negative latency reduction means Clock-RSM provides higher latency. 62

4.1 Definition of symbols. 72
4.2 Maximum throughput of client operations on a single partition server without replication. 85

1 Introduction

This thesis is concerned with building distributed systems with both good performance and desired consistency properties. We introduce protocols that efficiently implement three widely used consistency models in a distributed system by using loosely synchronized physical clocks.

1.1 Background and Motivation

Distributed systems overcome the limitations of a single server, such as computation and storage, by interconnecting a group of them to solve challenging problems [74]. They lay the foundation of recent trends in cloud computing and big data. In a distributed environment, however, some properties provided by a centralized system may not be satisfied easily. Hence various consistency models, from strong to weak, are proposed to trade off consistency with performance for different types of applications [22, 26, 58].

We focus on distributed data storage systems, which is an important category of distributed systems targeting large-scale data management. To manage data in huge volumes, a typical distributed data store normally partitions the managed data set across a large number of commodity servers. For better performance and data safety, it also replicates the data set by storing multiple copies of each data partition at different servers. The data set may be placed at multiple data centers to improve access latency and tolerate catastrophic failures.

One of the core challenges in building a distributed system is providing the required consistency properties efficiently. Event ordering, a fundamental concept in distributed systems, plays an important role in defining and implementing consistency models [50]. The performance of a distributed system is highly related to how it orders events. Strongly consistent systems normally totally order events using a single logical clock, which is typically implemented by a counting variable and main-

tained by a single centralized service. Weakly consistency systems may partially order events using a group of logical clocks to provide properties such as causality and convergence [62].

Using logical clocks to order events has the following two potential problems, although they are easy to implement and manage. First, a centralized logical clock serves all ordering requests via messages over the network. It can easily become a performance bottleneck and a single point of failure, which affects the scalability and availability of a system. Second, when a group of logical clocks are used to track and check the causal relationships among events, the dependency metadata may increase linearly with the number of servers a system employs. Storing and transmitting metadata of large size and checking dependencies across a large number of servers affects the performance of the system.

This thesis presents efficient solutions for building distributed data stores that requires different consistency properties. All the solutions are based upon one essential idea: *using a group of loosely synchronized physical clocks to order events.*

Compared with logical clocks, physical clocks move forward automatically at roughly the same speed. They do not need be advanced after the occurrence of each event. Commodity servers are all equipped with a reliable physical clock. The physical clocks of a group of servers can be loosely synchronized by a clock synchronization protocol, such as the Network Time Protocol (NTP) [2]. With the assistance of the operating system, each physical clock provides monotonically increasing timestamps very close to real time. With these properties, a group of physical clocks can help order events in a distributed environment at very low cost.

Our solutions use loosely synchronized physical clocks to order events in a distributed system and overcome the limitations of logical clocks. We assign to each event a timestamp obtained from the local physical clock and use this timestamp to totally order all events across the whole system. As we show in this thesis, this simple idea is very powerful and can be used to efficiently solve fundamental problems in distributed computing.

In the rest of this chapter, we first briefly review a number of widely used consistency models and show the important role of event ordering in their definitions and implementations. We then describe the challenges of implementing three consistency models, which are all related to event ordering. We also present an overview of our solutions that efficiently implement these three consistency models with the assistance of loosely synchronized physical clocks.

1.2 Consistency Models

A *consistency model* is a contract between a data store and its clients. It specifies the consistency properties that a data store promises to provide to its clients. In this section we present a brief overview of various consistency models and show that event ordering is key in characterizing these models.

For a centralized system that consists of a single process, events are naturally ordered by the sequential execution of the process. For a distributed system that orchestrates a large number of processes, determining the order of any two events at different processes is not trivial. The stronger a consistency model is, the closer it orders events to a single process. There exists a broad range of consistency models that allow programmers to trade off consistency for a potential gain in performance.

1.2.1 Consistency of Replicated Data Stores

We first introduce four consistency models widely used in the context of data replication. We assume that a data store provides two basic operations: read and update the value of a data item atomically. The system consists of multiple replicas that each store a full copy of the managed data set. Clients access the data store concurrently from different replicas.

Linearizability [39] is a very strong consistency model. A linearizable replicated data store provides to its clients the same behavior as an unreplicated data store. A replicated data store is linearizable if it satisfies the following three properties: 1) The result of any client operation is the same as if the operations by all clients were executed in some sequential order. 2) The operations of each client appear in this sequence in the order of its execution. 3) If any operation x finishes before operation y starts in real time, then x precedes y in this sequence. Linearizability is a local property and hence is composable. If every data item of a data store is linearizable, the data store is also linearizable.

Sequential consistency [52] is also a strong consistency model but weaker than linearizability. It provides the first two properties of linearizability but not the last one. It does not require that two independent operations from different clients follow the real time order in the global sequence. Sequential consistency is a global property and is not composable. If two data items are replicated separately and each of them is sequentially consistent, the combined data set is not guaranteed to provide sequential consistency.

Global coordination among replicas is indispensable to implement strong consistency models. To provide linearizability or sequential consistency, a data store

normally executes client operations in the same total order at all replicas. Concurrent operations are often ordered through a single sequencer node. Coordination services such as Chubby [18] and Zookeeper [40] and distributed data stores such as MegaStore [12] and Spanner [22] are examples of strongly consistent systems. They all use Paxos [52, 53] or a similar protocol [43] to totally order operations from each replica through a distributed leader replica.

Eventual consistency [76] is a weak consistency model. Its widely adopted informal definition specifies a liveness property: If no new updates are applied to a data item, eventually all reads to the item return the same value. A replicated data store that is eventually consistent does not require coordinating client operations globally. Hence it provides better performance than a strongly consistent data store. If the update operations to an item are not commutative, eventual consistency still requires some local coordination on them for convergence, i.e., all replicas reaching the same state on that item.

Causal+ consistency [58] is a restriction of eventual consistency with the additional requirement of *causal consistency* [7, 50]: an update becomes visible to clients only after its causal dependency states are visible. For convenience, in this thesis we also refer causal consistency to causal+ consistency, which requires both causality and convergence. Essentially, causal consistency requires a partial order among causally related events.

Guaranteeing causality is useful to a broad range of applications, such as online social networks, which normally do not require strong consistency. Dynamo [26] and COPS [58] are examples of eventually and causally consistent data stores, respectively, which demonstrate the virtues of weak consistency models.

The fundamental difference between strong and weak consistency models is whether global coordination of operations is required. Obviously, global coordination is only possible if the replicas can communicate with each other through messages over the network. The CAP theorem shows that, when the network partitions, a replicated system can either have availability or consistency, but not both [17, 34]. In other words, a strongly consistent system does not provide high availability while a weakly consistent system does. In addition, a weakly consistent system also provides low update latency, which is important for geo-replication. This is one of the most important principles that guide the design of distributed systems. While respecting the CAP theorem, we provide more efficient implementations of existing consistency models by using loosely synchronized physical clocks.

1.2.2 Transaction Isolation Levels

The concept of transaction is an important primitive and model in data management. A transaction groups multiple read and/or update operations together while still providing properties similar to a single operation.

A transaction concurrency control mechanism provides *isolations* between concurrent transactions [15]. Two transactions are concurrent if they access the same data item(s) and their execution time overlaps. Isolation constraints what a transaction can read and write at execution. Similar to consistency models we introduced before, the isolation level of a transaction specifies the properties that a transaction needs to maintain when executing concurrently with other transactions. People sometimes also use consistency to refer to the isolation properties of a transaction.

We briefly explain two widely used transaction isolation levels in both commercial and open-source database systems. We assume a data store that is not replicated. A transaction consists of one or more read and update operations. It may complete atomically with a commit, or with an abort. A transaction commits only if it satisfies the properties specified by its isolation level.

Serializability [15] is the strongest level of isolation. It requires that the schedule for executing concurrent transactions is equivalent to one that executes the transactions serially in some order. Under serializability, a transaction does not interfere with other transactions, i.e., it is totally isolated as if all transactions run serially. When a serializable transactional data store is replicated, to still guarantee serializability, every replica should commit transactions in the same serial order.

Snapshot isolation (SI) [13] is another popular strong isolation level, although it is not serializable. An SI implementation must satisfy two properties: 1) A transaction reads the most recent committed version as of the time when the transaction starts. 2) The write sets of each pair of committed concurrent transactions do not intersect.

SI originates from multi-versioning concurrency control (MVCC), which naturally relies on the notion of time in the definition and implementation. A typical implementation of SI normally totally orders transaction commits and provides consistent snapshots using a logical clock. The commit of an update transaction advances the clock. By reading the clock, a transaction obtains a consistent snapshot that includes the updates of all transactions committed before the snapshot time.

Many other weak transaction isolation levels exist in the literature [4]. Similar to consistency models, a weaker isolation level is also less expensive to implement because it requires less coordination, i.e., looser ordering, among concurrent transactions.

1.2.3 Summary

Event ordering is one of the fundamental problems in distributed computing. Based on our introduction above, it is not hard to observe that event ordering plays a critical role in defining and implementing various consistency models and transaction isolation levels. As we show next, using logical clocks for event ordering can hurt the throughput, latency, or availability of a system. To avoid or mitigate these problems, we use loosely synchronized physical clocks for event ordering in the implementation of three consistency models in this thesis.

1.3 Problem Statement

In this section we identify problems in the implementations of three widely used consistency models. These problems are all related to the usage of logical clocks for event ordering.

1.3.1 Snapshot Isolation for Partitioned Data Stores

SI is one of the most widely used concurrency control schemes in both commercial systems and research prototypes. A typical implementation of SI uses a logical clock to order transactions and provide consistent snapshots. In a centralized data store, the logical clock can be easily implemented by an atomic counter. Accessing the counter is almost free because it is just local memory operations. However, when the managed data set grows beyond the capacity of a single server, the data store has to partition the data set across multiple servers. As a consequence, the logical clock has to be maintained by a centralized time service running at a single server. The time service then totally orders transactions from different partitions by assigning them monotonically increasing commit timestamps. Percolator [67] and Omid [33] are examples of distributed data stores that implements SI with such a centralized time authority.

Lying on the critical path of transaction execution, the centralized time service affects the throughput, latency, and availability of the system. When a transaction starts at a partition, it obtains a consistent snapshot by requesting a snapshot timestamp from the centralized time service. Similarly, when an update transaction commits, it requests a commit timestamp from the time service. When the number of partitions is large, the service becomes a performance bottleneck because every transaction needs to access it through network messages. Transaction latency also increases because of additional communication steps to the time service. Furthermore, it affects the availability of the system, because it is a single point of failure.

1.3.2 State Machine Replication across Data Centers

The state machine approach [71] is often used to replicate services consistently by implementing a strong consistency model, such as linearizability and sequential consistency. If replicas start from the same initial state and they behave in a deterministic fashion, their states are consistent after executing the same sequence of commands, because the state machine replication protocol guarantees that all replicas execute the same set of commands in the same order.

To improve service availability and data locality, many online services replicate their data consistently at multiple geographic locations. A geo-replicated system can tolerate the failure of replicas due to server, network, and data center outage. It also serves user requests using the nearby replica hence reducing service latency and improving user experiences.

Although many protocols have been proposed to replicate state machines, they are not well suited for geo-replication because of high replication latency. Multi-Paxos [52, 53], a variant of Paxos, is the most widely used state machine replication protocol in production. In this protocol, one replica is designated as the leader, which orders commands by contacting a majority of replicas in one round trip of messages. A non-leader replica forwards its command to the leader and later receives the commit notification, adding the latency of another round trip. In a wide-area environment, the latency of two message round trips is significant. Essentially, Multi-Paxos relies on a logical clock at the leader replica to totally order commands. This incurs high latency at the non-leader replicas because they have to contact the leader to order each command.

1.3.3 Scalable Causally Consistent Data Replication

Many applications, such as online social networks, require high availability but not strong consistency. For these applications, causal consistency is a good option because it preserves the virtues of eventual consistency, namely high availability and low update latency. In addition, it also provides stronger semantics than eventual consistency by guaranteeing causality.

For purely replicated systems, causal consistency is implemented with *version vectors* [7, 65]. Each replica totally orders its local updates and maintains a version vector. The size of a version vector is equal to the total number of replicas. An element of a version vector is a scalar value and denotes how many updates the replica has applied from another replica. To track the dependencies of a local update, a replica associates the latest version vector with the update as its *dependency vector*. When the

update is sent to another replica for replication, it is not applied and does not become visible until the version vector of that replica becomes equal to or greater than the dependency vector of the update.

Implementing causal consistency in a data store that is both partitioned and replicated is challenging [58]. As version vectors are designed for pure replicated system and are not scalable for a large number of partitions, recent solutions track every dependency at the client side and check during replication that the dependencies of an update are present at a remote replica before install the update at that replica [58, 59]. These solutions have a number of drawbacks. First, explicitly storing every dependency and transmitting it to other replicas during replication consumes CPU cycles and network bandwidth. Second, checking the dependencies of each replicated update requires additional network messages to other data partitions, which also consumes CPU cycles and bandwidth. Third, providing causally consistent read-only transactions dramatically increases the number of tracked and checked dependencies, because it cannot utilize the transitivity of causality and requires tracking the complete set of dependencies.

1.4 Solution Overview

In this section, we introduce protocols that efficiently implement three consistent models, snapshot isolation, linearizability, and causal consistency, in a distributed environment. The protocols rely on physical clocks to solve most of the problems described in the previous section. With a group of loosely synchronized physical clocks, they either replace the existing centralized logical clock or impose explicit partial orders on events. The precision of clock synchronization does not affect the correctness of our solutions, although it may affect performance in unlikely circumstances where clock skews are large.

1.4.1 Clock-SI

Clock-SI is a fully distributed implementation of SI for partitioned data stores. It does not rely on a single logical clock to order transactions and provides snapshots. A transaction obtains its snapshot timestamp by reading the physical clock of the first partition it accesses. A single-partition update transaction obtains its commit timestamp from the clock of the partition it updates. A transaction that updates multiple partitions uses an augmented two-phase commit protocol to derive its commit timestamp. By avoiding a centralized service on the critical path of transaction execution, Clock-SI improves the throughput and latency of both read-only and update

transactions. The scalability of the system is not limited by the centralized service any more. Clock-SI also improves the availability of the system by eliminating a single point of failure.

There are two challenges of using loosely synchronized physical clocks. First, a hardware physical clock only provides monotonically increasing timestamps. It is less flexible than a logical clock implemented by a counter variable. Second, clocks are loosely synchronized. The time indicated by a clock may drift from the standard time by a small value. A snapshot taken at one server may not be available at another server because the clock at that server is behind by some time. While these situations happen relatively rarely, they must be handled for correctness. Clock-SI addresses these challenges and preserves the properties of SI.

1.4.2 Clock-RSM

Clock-RSM is a low-latency linearizable state machine replication protocol that targets geo-replication. Replicas are placed at different data centers to provide low access latency and tolerate disastrous failures. Clock-RSM does not rely on a distinguished leader replica to orders commands. Instead, a replica assigns to each command from its own clients a timestamp using the local physical clock. Commands from all replicas are then totally ordered by their assigned physical timestamps. Hence, Clock-RSM is a multi-leader protocol.

Clock-RSM requires three steps to commit a command: 1) Logging the command at a majority of replicas. 2) Determining the stable order of the command. 3) Notifying the commit of the command to all replicas. Clock-RSM overlaps these steps as much as possible to reduce the replication latency. By avoiding a centralized logical clock for command ordering, for many real world replica placements across data centers, Clock-RSM needs only one round-trip latency to a majority of replicas to commit a command. Since Clock-RSM does not mask replica failures as in Paxos, we also introduce a reconfiguration protocol that automatically removes failed replicas and reintegrates recovered replicas to the system.

1.4.3 Orbe

Orbe is a partitioned and replicated key-value store that provides causal consistency. It is scalable because replicas of different partitions replicate their updates independently in parallel. More importantly, it uses three techniques to reduce the overhead of dependency tracking and checking in existing solutions.

First, Orbe totally orders updates at each replica of a partition. With this ordering,

it uses *dependency matrices*, a two-dimensional data structure, to compactly track dependencies at the client side. Each element in a dependency matrix is a scalar value that represents all dependencies from the corresponding data store server. The size of dependency matrices is bounded by the total number of servers in the system.

Second, Orbe orders causally dependent states using loosely synchronized physical clocks. It assigns to each state an update timestamp obtained from a local physical clock and guarantees that the update timestamp order of causally related states is consistent with their causal order. Orbe provides causally consistent snapshots of the data store to read-only transactions by assigning them a snapshot timestamp, which is also obtained from a local physical clock.

Third, Orbe proposes *dependency cleaning*, an optimization that further reduces the size of dependency metadata. It is based on the observation that once a state and its dependencies are fully replicated, any subsequent read on the state does not introduce new dependencies to the client session.

With the above techniques, Orbe effectively reduces the cost of implementing causal consistency in a distributed data store. The improvement comes from imposing orders on data store operations using both logical clocks and physical clocks.

1.5 Thesis Organization

The rest of this thesis is structured as follows: Chapter 2 describes Clock-SI, a scalable implementation of SI for partitioned data stores. Chapter 3 presents Clock-RSM, a low-latency state machine replication protocol for geographic replication. Chapter 4 describes Orbe, a scalable implementation of causal consistency. Chapter 5 describes prior work on the implementations of different consistency models. Finally, Chapter 6 concludes this thesis.

2 Clock-SI: Snapshot Isolation for Partitioned Data Stores

In this chapter, we introduce Clock-SI, a fully distributed implementation of snapshot isolation (SI) for partitioned data store systems.

2.1 Introduction

SI [13] is one of the most widely used concurrency control schemes. While allowing some anomalies not possible with serializability [15], SI has significant performance advantages. In particular, SI never aborts read-only transactions, and read-only transactions do not block update transactions. SI is supported in several commercial systems, such as Microsoft SQL Server, Oracle RDBMS, and Google Percolator [67], as well as in many research prototypes [25, 30, 42, 77, 80].

2.1.1 SI Overview

Intuitively, under SI, a transaction takes a snapshot of the database when it starts. A snapshot is equivalent to a logical copy of the database including all committed updates. When an update transaction commits, its updates are applied atomically and a new snapshot is created. Snapshots are totally ordered according to their creation order using monotonically increasing timestamps. Snapshots are identified by timestamps: The snapshot taken by a transaction is identified by the transaction's *snapshot timestamp*. A new snapshot created by a committed update transaction is identified by the transaction's *commit timestamp*.

Managing timestamps in a centralized system is straightforward. Most SI implementations maintain a global variable, the *database version*, to assign snapshot and commit timestamps to transactions.

When a transaction starts, its snapshot timestamp is set to the current value of the database version. All its reads are satisfied from the corresponding snapshot. To support snapshots, multiple versions of each data item are kept, each tagged with a

version number equal to the commit timestamp of the transaction that creates the version. The transaction reads the version with the largest version number smaller than or equal to its snapshot timestamp. If the transaction is read-only, it always commits without further checks. If the transaction makes updates, its writes are buffered in a private workspace not visible to other transactions. When the update transaction requests to commit, a certification check verifies that the transaction writeset does not intersect with the writesets of concurrent committed transactions. If the certification succeeds, the database version is incremented, and the transaction commit timestamp is set to this value. The transaction's updates are made durable and visible, creating a new version of each updated data item with a version number equal to the commit timestamp.

2.1.2 Problem Statement

Efficiently maintaining and accessing timestamps in a distributed system is challenging. We focus here on partitioning, which is the primary technique employed to manage large data sets. Besides allowing for larger data sizes, partitioned systems improve latency and throughput by allowing concurrent access to data in different partitions. With current large main memory sizes, partitioning also makes it possible to keep all data in memory, further improving performance.

Existing implementations of SI for a partitioned data store [42, 67, 78, 80] use a centralized time authority to manage timestamps. When a transaction starts, it requests a snapshot timestamp from the centralized authority. Similarly, when a successfully certified update transaction commits, it requests a commit timestamp from the centralized authority. Each partition does its own certification for update transactions, and a two-phase commit (2PC) protocol is used to commit transactions that update data items at multiple partitions. The centralized timestamp authority is a single point of failure and a potential performance bottleneck. It negatively impacts system availability, and increases transaction latency and messaging overhead. We refer to the implementations of SI using a centralized timestamp authority as *conventional SI*.

2.1.3 Solution Overview

We introduce Clock-SI, a fully distributed implementation of SI for partitioned data stores. Clock-SI uses loosely synchronized clocks to assign snapshot and commit timestamps to transactions, avoiding the centralized timestamp authority in conventional SI. Similar to conventional SI, partitions do their own certification, and a 2PC protocol is used to commit transactions that update multiple partitions.

Compared with conventional SI, Clock-SI improves system availability and performance. Clock-SI does not have a single point of failure and a potential performance bottleneck. It saves one round-trip message for a ready-only transaction (to obtain the snapshot timestamp), and two round-trip messages for an update transaction (to obtain the snapshot timestamp and the commit timestamp). These benefits are significant when the workload consists of short transactions as in key-value stores, and even more prominent when the data set is partitioned geographically across data centers.

We build on earlier work [5,6,71] to totally order events using physical clocks in distributed systems. The novelty of Clock-SI is to efficiently create consistent snapshots using loosely synchronized clocks. In particular, a transaction's snapshot timestamp is the value of the local clock at the partition where it starts. Similarly, the commit timestamp of a local update transaction is obtained by reading the local clock.

The implementation of Clock-SI poses several challenges because of using loosely synchronized clocks. The core of these challenges is that, due to a clock skew or pending commit, a transaction may receive a snapshot timestamp for which the corresponding snapshot is not yet fully available. We delay operations that access the unavailable part of a snapshot until it becomes available. As an optimization, we can assign to a transaction a snapshot timestamp that is slightly smaller than the clock value to reduce the possibility of delayed operations.

We build an analytical model to study the properties of Clock-SI and analyze the trade-offs of using old snapshots. We also verify the model using a system implementation. We demonstrate the performance benefits of Clock-SI on a partitioned key-value store using a micro-benchmark (YCSB [21]) and application-level benchmark (Twitter feed-following [73]). We show that Clock-SI has significant performance advantages. In particular, for short read-only transactions, Clock-SI improves latency and throughput by up to 50% over conventional SI. This performance improvement comes with higher availability as well.

In this chapter, we make the following contributions:

- We present Clock-SI, a fully distributed protocol that implements SI for partitioned data stores using loosely synchronized clocks (Section 2.4).
- We develop an analytical model to study the performance properties and trade-offs of Clock-SI (Section 2.5).
- We build a partitioned key-value store and experimentally evaluate Clock-SI to demonstrate its performance benefits (Section 2.6).

2.2 Model and Definition

In this section, we describe the system model and define SI.

2.2.1 System Model

We consider a multiversion key-value store, in which the dataset is partitioned and each partition resides on a single server. A server has a standard hardware clock. Clocks are synchronized by a clock synchronization protocol, such as Network Time Protocol (NTP) [2]. We assume that clocks always move forward, perhaps at different speeds as provided by common clock synchronization protocols [2, 56]. The absolute value of the difference between clocks on different servers is decided by the clock synchronization skew.

The key-value store supports three basic operations: `get`, `put`, and `delete`. A transaction consists of a sequence of basic operations. A client connects to a partition, selected by a load balancing scheme, and issues transactions to that partition. We call this partition the *originating partition* of these transactions. The originating partition executes the operations of a transaction sequentially. If the originating partition does not store a data item needed by an operation, it executes the operation at the remote partition that stores the item.

The originating partition assigns the snapshot timestamp to a transaction by reading its local clock. When an update transaction starts to commit, if it updates data items at a single partition, the commit timestamp is assigned by reading the local clock at that partition. We use a more complex protocol to commit a transaction that updates multiple partitions.

2.2.2 SI Definition

Formally, SI is a multiversion concurrency control scheme with three main properties [4, 13, 31] that must be satisfied by the underlying implementation:

(1) Each transaction reads from a consistent snapshot, taken at the start of the transaction and identified by a snapshot timestamp. A snapshot is consistent if it includes all writes of transactions committed before the snapshot timestamp, and if it does not include any writes of aborted transactions or transactions committed after the snapshot timestamp.

(2) Update transactions commit in a total order. Every commit produces a new database snapshot, identified by the commit timestamp.

(3) An update transaction aborts if it introduces a write-write conflict with a con-

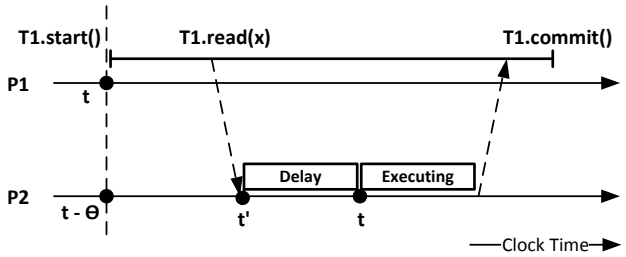

Figure 2.1 – Snapshot unavailability due to clock skew.

current committed transaction. Transaction T_1 is concurrent with committed update transaction T_2, if T_1 took its snapshot before T_2 committed and T_1 tries to commit after T_2 committed.

2.3 Challenges

Using loosely synchronized physical clocks to implement SI imposes a few challenges. We illustrate them in this section before presenting the details of the Clock-SI protocol.

From the SI definition, a consistent snapshot with snapshot timestamp t includes, for each data item, the version written by the transaction with the greatest commit timestamp smaller than or equal to t. This property holds independent of where a transaction starts and gets its snapshot timestamp, where an update transaction gets its commit timestamp, and where the accessed data items reside. Ensuring this property is challenging when assigning snapshot and commit timestamps using clocks as we illustrate here. While these situations happen relatively rarely, they must be handled for correctness. We show in detail how Clock-SI addresses these challenges in Section 2.4.

Example 1: First, we show that clock skew may cause a snapshot to be unavailable. Figure 2.1 shows a transaction accessing two partitions. Transaction T_1 starts at partition P_1, the originating partition. P_1 assigns T_1's snapshot timestamp to the value t. The clock at P_2 is behind by some amount θ, and thus at time t on P_1, P_2's clock value is $t-\theta$. Later on, T_1 issues a read for data item x stored at partition P_2. The read arrives at time t' on P_2's clock, before P_2's clock has reached the value t, and thus $t' < t$. The snapshot with timestamp t at P_2 is therefore not yet available. Another transaction on P_2 could commit at time t'', between t' and t, and change the value of x. This new value should be included in T_1's snapshot.

Example 2: Second, we show that the pending commit of an update transaction can cause a snapshot to be unavailable. Figure 2.2 depicts two transactions running

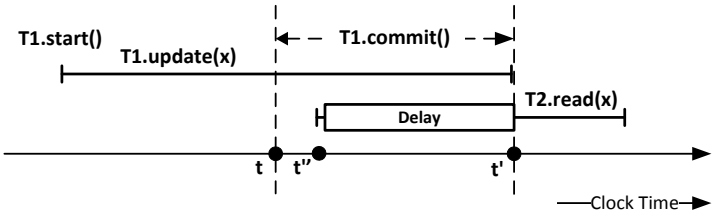

Figure 2.2 – Snapshot unavailability due to the pending commit of an update transaction.

in a single partition. T_2's snapshot is unavailable due to the commit in progress of transaction T_1, which is assigned the value of the local clock, say t, as its commit timestamp. T_1 updates item x and commits. The commit operation involves a write to stable storage and completes at time t'. Transaction T_2 starts between t and t', and gets assigned a snapshot timestamp t'', $t < t'' < t'$. If T_2 issues a read for item x, we cannot return the value written by T_1, because we do not yet know if the commit will succeed, but we can also not return the earlier value, because, if T_1's commit succeeds, this older value will not be part of a consistent snapshot at t''.

Both examples are instances of a situation where the snapshot specified by the snapshot timestamp of a transaction is not yet available. These situations arise because of using physical clocks at each partition to assign snapshot and commit timestamps in a distributed fashion. We deal with these situations by delaying the operation until the snapshot becomes available.

As an optimization, the originating partition can assign to a transaction a snapshot timestamp that is slightly smaller than its clock value, with the goal of reducing the probability and duration that an operation needs to be delayed, albeit at the cost of reading slightly stale data. Returning to Example 1, if we assign a snapshot timestamp by subtracting the expected clock skew from the local clock, then the probability of the snapshot not being available because of clock skew decreases substantially.

2.4 Clock-SI

We first present the read protocol, which provides transactions consistent snapshots across partitions, and the commit protocol. Next, we discuss correctness and other properties of Clock-SI.

2.4.1 Read Protocol

The read protocol of Clock-SI provides transactions consistent snapshots across multiple partitions. It has two important aspects: (1) the assignment of snapshot times-

Algorithm 1 Clock-SI read protocol.

```
1:  StartTransaction(transaction T)
2:      T.SnapshotTime ← GetClockTime() − Δ (Δ ≥ 0)
3:      T.State ← active
4:  ReadDataItem(transaction T, data item oid)
5:      if oid ∈ T.WriteSet return T.WriteSet[oid]
6:      // check if delay needed due to pending commit
7:      if oid is updated by T′ ∧
8:          T′.State = committing ∧
9:          T.SnapshotTime ≥ T′.CommitTime
10:     then wait until T′.State = committed
11:     if oid is updated by T′ ∧
12:         T′.State = prepared ∧
13:         T.SnapshotTime ≥ T′.PrepareTime ∧
14:         // Here T can obtain commit timestamp of T′
15:         // from its originating partition by a RPC.
16:         T.SnapshotTime ≥ T′.CommitTime
17:     then wait until T′.State = committed
18:     return latest version of oid created before T.SnapshotTime
19: upon transaction T arriving from a remote partition
20:     // check if delay needed due to clock skew
21:     if T.SnapshotTime > GetClockTime()
22:     then wait until T.SnapshotTime < GetClockTime()
```

tamps, and (2) delaying reads under certain conditions to guarantee that transactions access consistent snapshots identified by their snapshot timestamps. Algorithm 1 presents the pseudocode of the read protocol.

Timestamp assignment. When transaction T is initialized at its originating partition (lines 1-3), it receives the snapshot timestamp by reading the local physical clock, and possibly subtracting a parameter, Δ, to access an older snapshot as we explain in Section 2.4.3. The assigned timestamp determines the snapshot of the transaction.

Consistent Snapshot Reads. A transaction reads a data item by its identifier denoted by oid (lines 4-18). To guarantee that a transaction reads from a consistent snapshot, Clock-SI delays a read operation until the required snapshot becomes available in two cases.

Case 1: Snapshot unavailability due to pending commit. Transaction T tries to access an item that is updated by another transaction T' which has a commit timestamp smaller than or equal to T's snapshot timestamp but has not yet completed the commit. For example, T' is being committed locally but has not completely committed (lines 6-10) or T' is prepared in 2PC (lines 11-17). [1] We delay T's access to ensure that a snapshot includes only committed updates and all the updates committed before the snapshot timestamp. The delay is bounded by the time of synchronously writing the update transaction's commit record to stable storage, plus one round-trip network latency in the case that a transaction updates multiple partitions.

Case 2: Snapshot unavailability due to clock skew. When a transaction tries to access a data item on a remote partition and its snapshot timestamp is greater than the clock time at the remote partition, Clock-SI delays the transaction until the clock at the remote partition catches up (lines 19-22). The transaction, therefore, does not miss any committed changes included in its snapshot. The delay is bounded by the maximum clock skew allowed by the clock synchronization protocol minus one-way network latency.

In both cases, delaying a read operation does not introduce deadlocks: An operation waits only for a finite time, until a commit operation completes, or a clock catches up.

Clock-SI also delays an update request from a remote partition, under the same condition that it delays a read request, so that the commit timestamp of an update

1. The question arises how T knows that it is reading a data item that has been written by a transaction in the process of committing, since in SI a write is not visible outside a transaction until it is committed. The problem is easily solved by creating, after assignment of a prepare or a commit timestamp, a version of the data item with that timestamp as its version number, but by prohibiting any transaction from reading that version until the transaction is fully committed.

transaction is always greater than the snapshot timestamp (line 19).

Algorithm 2 Clock-SI commit protocol.

```
 1: CommitTransaction(transaction T)
 2:     if T updates a single partition
 3:     then LocalCommit(T)
 4:     else DistributedCommit(T)
 5: LocalCommit(transaction T)
 6:     if CertificationCheck(T) is successful
 7:         T.State ← committing
 8:         T.CommitTime ← GetClockTime()
 9:         log T.CommitTime and T.Writeset
10:         T.State ← committed
11: // two-phase commit
12: DistributedCommit(transaction T)
13:     for p in T.UpdatedPartitions
14:         send prepare T to p
15:     wait until receiving T prepared from participants
16:     T.State ← committing
17:     // choose transaction commit time
18:     T.CommitTime ← max(all prepare timestamps)
19:     log T.CommitTime and commit decision
20:     T.State ← committed
21:     for p in T.UpdatedPartitions
22:         send commit T to p
23: upon receiving message prepare T
24:     if CertificationCheck(T) is successful
25:         log T.WriteSet and T's coordinator ID
26:         T.State ← prepared
27:         T.PrepareTime ← GetClockTime()
28:         send T prepared to T's coordinator
29: upon receiving message commit T
30:     log T.CommitTime
31:     T.State ← committed
```

2.4.2 Commit Protocol

With Clock-SI, a read-only transaction reads from its snapshot and commits without further checks, even if the transaction reads from multiple partitions. An update transaction modifies items in its workspace. If the update transaction modifies a single partition, it commits locally at that partition. Otherwise, we use a coordinator to either commit or abort the update transaction at the updated partitions. One important aspect is how to assign a commit timestamp to update transactions. Algorithm 2

presents the pseudocode of the commit protocol.

Committing a single-partition update transaction. If a transaction updates only one partition, it commits locally at the updated partition (lines 5-10). Clock-SI first certifies the transaction by checking its writeset with concurrent committed transactions [31]. Before assigning the commit timestamp, the transaction state changes from *active* to *committing*. The updated partition reads its clock to determine the commit timestamp, and writes the commit record to stable storage. Then, the transaction state changes from *committing* to *committed*, and its effects are visible in snapshots taken after the commit timestamp.

Committing a distributed update transaction. A multi-partition transaction, which updates two or more partitions, commits using an augmented 2PC protocol (lines 11-31) [15].

The transaction coordinator runs at the originating partition. Certification is performed locally at each partition that executed updates for the transaction (line 24). Each participant writes its prepare record to stable storage, changes its state from *active* to *prepared*, obtains the prepare timestamp from its local clock, sends the prepare message with the prepare timestamp to the coordinator, and waits for the response (line 25-28). The 2PC coordinator computes the commit timestamp as the maximum prepare timestamp of all participants (line 18).

Choosing the maximum of all prepare timestamps as the commit timestamp for a distributed update transaction is important for correctness. Remember from the read protocol that, on a participant, reads from transactions with a snapshot timestamp greater than or equal to the prepare timestamp of the committing transaction are delayed. If the coordinator were to return a commit timestamp smaller than the prepare timestamp on any of the participants, then a read of a transaction with a snapshot timestamp smaller than the prepare timestamp but greater than or equal to that commit timestamp would not have been delayed and would have read an incorrect version (i.e., a version other than the one created by the committing transaction). Correctness is still maintained if a participant receives a commit timestamp greater than its current clock value. The effects of the update transaction will be visible only to transactions with snapshot timestamps greater than or equal to its commit timestamp.

2.4.3 Choosing Older Snapshots

In Clock-SI, the snapshot timestamp of a transaction is not restricted to the current value of the physical clock. We can choose the snapshot timestamp to be smaller

than the clock value by Δ, as shown on line 2 of Algorithm 1. We can choose Δ to be any non-negative value and make this choice on a per-transaction basis. If we want a transaction to read fresh data, we set Δ to 0. If we want to reduce the delay probability of transactions close to zero, we choose an older snapshot by setting Δ to the maximum of (1) the time required to commit a transaction to stable storage synchronously plus one round-trip network latency, and (2) the maximum clock skew minus one-way network latency between two partitions. These two values can be measured and distributed to all partitions periodically. Since networks and storage devices are asynchronous, such a choice of the snapshot age does not completely prevent the delay of transactions, but it significantly reduces the probability.

While substantially reducing the delay probability of transactions, taking a slightly older snapshot comes at a cost: The transaction observes slightly stale data, and the transaction abort rate increases by a small fraction. We study this trade-off using an analytical model in Section 2.5 and experiments on a prototype system in Section 2.6.

2.4.4 Correctness

We show that Clock-SI implements SI by satisfying the three properties that define SI [31]. Furthermore, we show that safety is always maintained, regardless of clock synchronization precision.

(1) *Transactions commit in a total order.* Clock-SI assigns commit timestamps to update transactions by reading values from physical clocks. Ties are resolved based on partition ids. The commit timestamp order produces a total order on transaction commits.

(2) *Transactions read consistent snapshots.* The snapshot in Clock-SI is consistent with respect to the total commit order of update transactions. The snapshot timestamp specifies a snapshot from the totally ordered commit history. A transaction reads all committed changes of transactions with a smaller commit timestamp. By delaying transaction operations in the read protocol, a transaction never misses the version of a data item it is supposed to read. A transaction does not read values from an aborted transaction or from a transaction that commits with a greater commit timestamp.

(3) *Committed concurrent transactions do not have write-write conflicts.* Clock-SI identifies concurrent transactions by checking whether their execution time overlaps using the snapshot and commit timestamps. Clock skews do not affect the identification of concurrent transactions according to their snapshot and commit timestamps. Clock-SI aborts one of the two concurrent transactions with write-write conflicts.

We also point out an important property of Clock-SI: The precision of the clock

synchronization protocol does not affect the correctness of Clock-SI but only the performance. Large clock skews increase a transaction's delay probability and duration; safety is, however, always maintained, satisfying the three properties of SI.

Although it does not affect Clock-SI's correctness, the transaction commit order in Clock-SI may be different from the real time commit order (according to global time) because of clock skews. This only happens to independent transactions at different partitions whose commit timestamp difference is less than the clock synchronization precision. This phenomenon also happens with conventional SI: A transaction commits on a partition after it obtains the commit timestamp from the timestamp authority. The asynchronous messages signaling the commit may arrive at the partitions in a different order from the order specified by the timestamps. Clock-SI preserves the commit order of dependent transactions when this dependency is expressed through the database as performed in a centralized system [15].

2.4.5 Discussion

Clock-SI is a fully distributed protocol. Compared with conventional SI, Clock-SI provides better availability and scalability. In addition, it also reduces transaction latency and messaging overhead.

Availability. Conventional SI maintains timestamps using a centralized service, which is a single point of failure. Although the timestamp service can be replicated to tolerate certain number of replica failures, replication comes with performance costs. In contrast, Clock-SI does not include such a single point of failure in the system. The failure of a data partition only affects transactions accessing that partition. Other partitions are still available.

Communication cost. With conventional SI, a transaction needs one round of messages to obtain the snapshot timestamp. An update transaction needs another round of messages to obtain the commit timestamp. In contrast, under Clock-SI, a transaction obtains the snapshot and commit timestamps by reading local physical clocks. As a result, Clock-SI reduces the latency of read-only transactions by one round trip and the latency of update transactions by two round trips. By sending and receiving fewer messages to start and commit a transaction, Clock-SI also reduces the cost of transaction execution.

Scalability. Since Clock-SI is a fully distributed protocol, the throughput of single-partition transactions increases as more partitions are added. In contrast, conventional SI uses a centralized timestamp authority, which can limit system throughput as it is on the critical path of transaction execution.

Session consistency. Session consistency [24, 47] guarantees that, in a workflow of transactions in a client session, each transaction sees (1) the updates of earlier committed transactions in the same session, and (2) non-decreasing snapshots of the data. Session consistency can be supported under Clock-SI using the following standard approach. When a client finishes a read-only transaction, its snapshot timestamp is returned. When a client successfully commits an update transaction, its commit timestamp is returned. $LatestTime$ maintained by a client is updated to the value returned by the last transaction completed. When a client starts a new transaction, it sends $LatestTime$ to the originating partition for that transaction. If $LatestTime$ is greater than the current clock value at that partition, it is blocked until the clock proceeds past $LatestTime$. Otherwise, it starts immediately.

Recovery. We employ traditional recovery techniques to recover a partition in Clock-SI. Each partition maintains a write-ahead log (WAL) containing the transaction update records (as redo records), commit records, as well as 2PC prepare records containing the identity of the coordinator. In addition, the partition uses checkpointing to reduce the recovery time. Taking a checkpoint is the same as reading a full snapshot of the partition state. If a partition crashes, it recovers from the latest complete checkpoint, replays the log, and determines the outcome of prepared but not terminated transactions from their coordinators.

2.5 Analytical Model

In this section, we assess the performance properties of Clock-SI analytically. Our objective is to reason about how various factors impact the performance of Clock-SI. We show the following: (1) With normal database configurations, the delay probability of a transaction is small and the delay duration is short. (2) Taking an older snapshot reduces the delay probability and duration, but slightly increases the abort rate of update transactions.

We derive formulas for transaction delay probability, delay duration, and abort rate. We verify the model predictions in Section 2.6.5 using a distributed key-value store. Readers who are not interested in the mathematical derivations of the analytical model may skip this section.

2.5.1 Model Parameters

Our model is based on prior work that predicts the probability of conflicts in centralized [36] and replicated databases [31, 35]. We consider a partitioned data store that runs Clock-SI. The data store has a fixed set of items. The total number of items is

DBSize. We assume all partitions have the same number of items. There are two types of transactions: read transactions and update transactions. Each read transaction reads R items and takes L_r time units to finish. Each update transaction updates W items and takes L_u time units to finish. The data store processes TPS_u update transactions per unit time.

The *committing window* of update transaction T_i, denoted as CW_i, is the time interval during which T_i is in the committing state. On average, an update transaction takes CW time units to persist its writeset to stable storage synchronously. We denote RRD as the message request-reply delay, which includes one round-trip network delay, data transfer time, and message processing time. We denote S as the time to synchronously commit an update transaction to stable storage. Δ, the snapshot age, is a non-negative value subtracted from the value read from the physical clock to obtain an older snapshot, as shown in Algorithm 1.

2.5.2 Delay due to Pending Commit

A transaction might be delayed when it reads an item updated by another transaction being committed to stable storage.

For a read transaction T_i, it takes its snapshot at time ST_i. Assume another update transaction T_j obtains its commit timestamp at time CT_j. With Clock-SI, at time CT_j, T_j still needs CW time to synchronously persist its writeset to stable storage. Assume T_j obtains its commit timestamp before T_i takes its snapshot, i.e., $CT_j < ST_i$. If $ST_i - CT_j < CW$, when T_i reads an item updated by T_j, it is delayed until T_j completes the commit to stable storage. Other update transactions T_k, $ST_i - CT_k > CW$, do not delay T_i, because at the time T_i reads items updated by T_k, T_k must have completed. Taking a Δ old snapshot is equivalent to shortening the committing window of update transactions to $CW - \Delta$. In this case, T_i is delayed by T_j if $ST_i - CT_j < CW - \Delta$. If $\Delta > CW$, $CW - \Delta$ effectively becomes zero.

During $CW - \Delta$, $(CW - \Delta) * TPS_u$ transactions update $(CW - \Delta) * TPS_u * W$ data items. The probability that T_i reads any particular item in the database is $R/DBSize$. If $CW - \Delta \geqslant L_r$, then the probability that T_i is delayed is $(CW - \Delta) * TPS_u * W * (R/DBSize)$. If $CW - \Delta < L_r$, then only reading the first $R * (CW - \Delta)/L_r$ items possibly delays T_i. The probability becomes $(CW - \Delta) * TPS_u * W * (R * (CW - \Delta)/L_r/DBSize)$.

For a transaction that only updates one partition, its committing window is $CW = S$ and its commit timestamp is available at the updated partition. For a transaction that updates multiple partitions, its commit timestamp is only available at its originating partition before 2PC completely finishes. Hence a delayed transaction may take one

extra round-trip network latency to obtain a commit timestamp from the update transaction's originating partition. We use $CW = S + RRD$ as an estimation of the committing window of a distributed update transaction.

We assume all update transactions update items at multiple partitions. Combining the above analysis, the delay probability of short read transactions, i.e., $CW - \Delta \geqslant L_r$, is

$$(S + RRD - \Delta) * TPS_u * W * R/DBSize$$

The delay probability of long read transactions, i.e., $CW - \Delta < L_r$, is

$$(S + RRD - \Delta)^2 * TPS_u * W * R/(DBSize * L_r)$$

The expected delay duration of read transactions is

$$0.5 * (S + RRD - \Delta)$$

The above results show that the delay duration is bounded and normally short. Although the delay probability depends on various factors, we show that, with normal database configurations and workloads, it is low by a numerical example in Section 2.5.5 and experiments in Section 2.6.5. By assigning an older snapshot to a transaction, we can reduce its delay probability and shorten its delay duration.

2.5.3 Delay due to Clock Skew

Imperfect time synchronization causes clock skew. A transaction is delayed when it accesses a remote partition and the remote clock time is smaller than its snapshot timestamp. As we show in Section 2.6, common clock synchronization protocols, such as NTP, work well in practice and the clock skew is very small. Hence, this type of delay rarely happens.

We assume the clock skew SK between each pair of clocks follows normal distribution [32] with mean μ and standard deviation δ. The delay probability when a transaction accesses a remote partition is

$$P(SK > 0.5 * RRD + \Delta) = 1 - \Phi((0.5 * RRD + \Delta - \mu)/\delta)$$

The expected delay duration is

$$\frac{\int_{0.5*RRD+\Delta}^{+\infty} x e^{-(x-\mu)^2/(2\delta^2)}\, dx}{\int_{0.5*RRD+\Delta}^{+\infty} e^{-(x-\mu)^2/(2\delta^2)}\, dx} - (0.5 * RRD + \Delta)$$

The results show that if the clock skew is greater than one-way network latency, it becomes possible that a transaction is delayed when accessing a remote partition, because the requested snapshot is not yet available when the transaction arrives. By assigning an older snapshot to a transaction, we can reduce its delay probability and shorten its delay duration.

Suppose the maximum clock skew is SK_{max} time units. Taking an older snapshot with $\Delta = max(CW, SK_{max} - 0.5 * RRD)$ eliminates almost all the delays due to either pending commits or clock skews.

2.5.4 Update Transaction Abort Probability

Assigning old snapshots to transactions reduces their delay probability and duration. However, this increases the abort rate of update transactions because the (logical) execution time of an update transaction is extended and more update transactions run concurrently with it.

We first compute the probability that a transaction T_i has to abort without adjusting the snapshot timestamp [31]. On average, the number of transactions that commit in L_u, the life time of T_i, is $TPS_u * L_u$. The number of data items updated by the these transactions is $W * TPS_u * L_u$. The probability that one particular item in the database is updated during L_u is $W * TPS_u * L_u / DBSize$. As T_i updates W items in total, the probability that T_i has conflicts with its concurrent transactions and has to abort is $W^2 * TPS_u * L_u / DBSize$. The abort probability is directly proportional to L_u, the duration of update transaction execution.

Clock-SI can assign to each transaction a snapshot that is Δ older than the latest snapshot. The transaction abort probability becomes

$$W^2 * TPS_u * (L_u + \Delta) / DBSize$$

With other parameters fixed, the longer a transaction executes, the more concurrent transactions run with it, increasing the likelihood of write-write conflicts with other transactions. Assigning an older snapshot to a transaction increases its abort probability.

2.5.5 Example

We provide a numerical example using the equations to calculate the transaction delay probability and duration due to pending commits of update transactions. We assume $DBSize = 10,000,000$, $R = 10$, $W = 10$, $TPS_u = 10,000$, $RRD = 0.2ms$, $\Delta = 0$,

$CW = 8ms$ and $L_u = 32ms$. If we use mechanical hard disks to persist transaction updates, it takes a few milliseconds (e.g., 8ms) to commit an update transaction. The delay probability is 0.08% and the expected delay time is 4.1ms. On average, each transaction is delayed for 3.2μs. If a transaction takes a snapshot that is 8.2ms earlier than the clock time, i.e., $\Delta = 8.2ms$, both delay probability and time are zero, and the transaction abort probability increases by 25% (from 0.0032 to 0.004). Therefore we can almost eliminate transaction delays at the cost of a slight increase in the abort rate.

2.6 Evaluation

We evaluate the performance benefits of Clock-SI using a micro-benchmark and an application-level benchmark with a partitioned key-value store in both LAN and WAN environments, and show the following: (1) Compared with conventional SI, Clock-SI reduces transaction response time and increases throughput. (2) Selecting a slightly older snapshot reduces the delay probability and duration of transactions. Furthermore, we verify the predictions of our analytical model using experimental measurements. We focus only on assessing the performance benefits of Clock-SI, rather than the improvement in availability stemming from avoiding a single point of failure.

2.6.1 Implementation and Setup

We build a partitioned multiversion key-value store in `C++`. It supports Clock-SI as well as conventional SI as implemented in other systems [67]. With conventional SI, a transaction communicates with a centralized timestamp authority running on a separate server to retrieve timestamps. A transaction needs one round of messages to obtain the snapshot timestamp. An update transaction needs another round of messages to obtain the commit timestamp.

The data set is partitioned among a group of servers. Servers have standard hardware clocks synchronized using NTP running in peer mode [2]. A key is assigned to a partition based on its hash. The default size of a key and its value are 8 and 64 bytes, respectively. We keep all key-value pairs in a group of hash tables in main memory. A key points to a linked list that contains all the versions of the corresponding value. The transaction commit log resides on the hard disk. The system performs group commit to write multiple transaction commit records in one stable disk write.

We conduct experiments in both LAN and WAN environments. For LAN experiments, we deploy our key-value store in a local cluster. Servers in our local cluster run

Linux 3.2.0. Each server has two Intel Xeon processors with 4GB DDR2 memory. The transaction log resides on a 7200rpm 160GB SATA disk. We disable the disk cache so that synchronous writes reach the disk media. The average time of a synchronous disk write is 6.7ms. The system has eight partitions. Each partition runs in one server and manages one million keys in main memory. All machines are connected to a single Gigabit Ethernet switch. The average round-trip network latency is 0.14 milliseconds. For WAN experiments, we deploy our system on Amazon EC2 using medium instances (size M1) at multiple data centers.

2.6.2 Micro-benchmarks

We implement a benchmark tool based on the Yahoo! Cloud Serving Benchmark (YCSB) [21], which is designed to benchmark key-value stores. We extend YCSB to support transactions.

Latency. Clock-SI takes shorter time to run a transaction because it does not communicate with a timestamp authority for transaction snapshot and commit timestamps. Figure 2.3 shows the latency of read-only transactions in our local cluster. Both the clients and servers are connected to the same switch. Each transaction reads eight items at each partition, with keys chosen uniformly randomly. We vary the number of partitions accessed by a transaction from one to three. Compared with conventional SI, Clock-SI saves approximately one round-trip latency. For a single-partition read-only transaction this amounts to about 50% latency savings. We also see consistent savings of one round-trip latency for the other two cases.

We run the same experiment in a WAN environment on Amazon EC2 and measure transaction latency. We place three data partitions at three data centers in different geographical locations: US West, US East and Europe. The timestamp server is co-located with the partition in US West. A client always chooses the nearest partition as the originating partition of its transaction.

First, we run clients at our local cluster in Europe and the number of partitions accessed by a transaction varies from one to three. Figure 2.4 shows the measured latency. Compared with conventional SI, Clock-SI reduces the latency of each transaction by about 160 milliseconds since it does not contact the remote timestamp server. Next, for transactions issued by the clients near US West and served by the partition co-located with the timestamp server, the saved latency by Clock-SI becomes sub-milliseconds and the latencies are similar to those in Figure 2.3. Last, for the clients near US East, Clock-SI still reduces their transaction latency by one round-trip between two data centers, which is about 170 milliseconds.

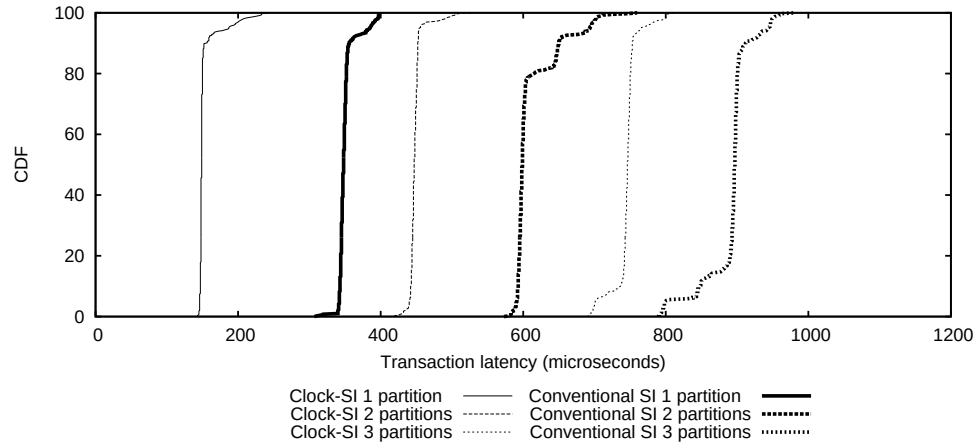

Figure 2.3 – Latency distribution of read-only transactions in a LAN.

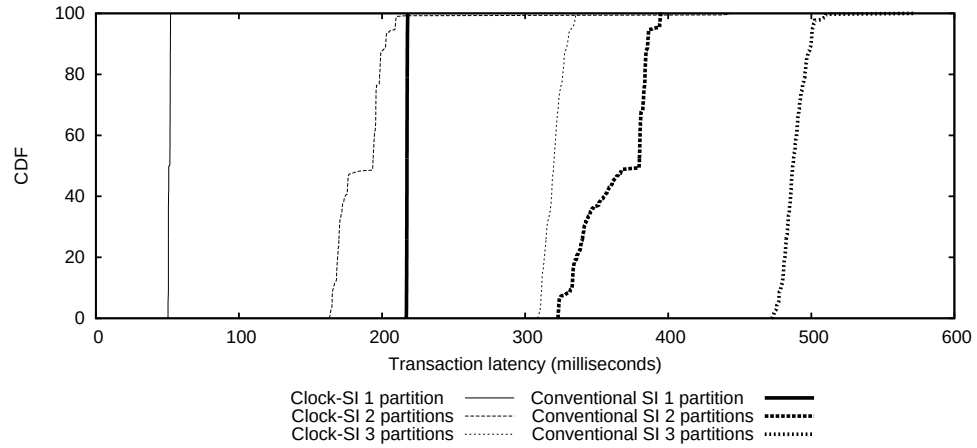

Figure 2.4 – Latency distribution of read-only transactions in a WAN. Partitions are in data centers in Europe, US West and US East. Clients are in our local cluster in Europe.

For update transactions, the latency is reduced similarly by two network round trips in both LAN and WAN environments.

Throughput. Next we compare the throughput of Clock-SI with conventional SI. We run a large number of clients to saturate the servers that host the key-value data partitions. Figure 2.5 shows the throughput of read-only and update-only single-partition transactions. The number of partitions serving client requests varies from one to eight. Each transaction reads or updates eight items randomly chosen at each partition.

We first analyze read-only transactions. Below five partitions, the throughput of Clock-SI is about twice of conventional SI. The cost of a read-only transaction stems mainly from sending and receiving messages. As Clock-SI does not contact

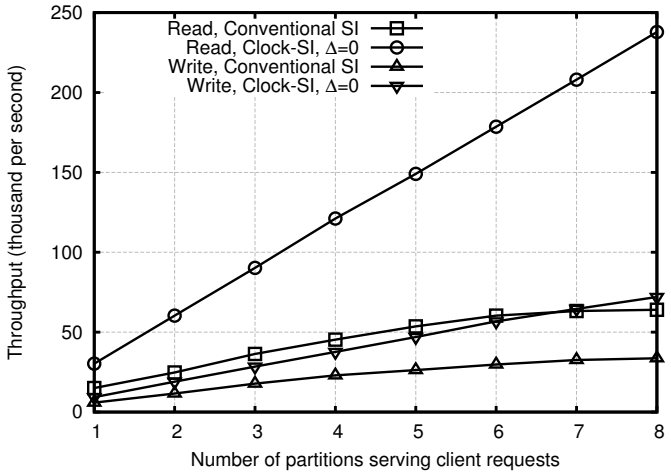

Figure 2.5 – Throughput comparison for single-partition read-only and update-only transactions. Each transaction reads/updates eight items.

the timestamp authority to obtain snapshot timestamps, it sends and receives one message for each transaction, while conventional SI sends and receives two messages. Beyond five partitions, the throughput gap becomes larger: The throughput of Clock-SI increases linearly as the number of partitions increases. For conventional SI, the throughput levels off at around 64k transactions/second. The timestamp authority becomes the bottleneck because it can only assign about 64k timestamps/second.

Update transactions show similar results in Figure 2.5. The throughput of update transactions is lower than that of read-only transactions, because an update transaction does more work, including creating new versions of items, updating version metadata, and performing I/O to make updates durable on the disk. Update transactions require two timestamps from the timestamp authority. Given its limit of 64k timestamps/second, the timestamp authority sustains only 32k update transactions/second. The timestamp authority again becomes the bottleneck.

Batching timestamp requests from the data partitions to the timestamp authority can improve the throughput of conventional SI. However, message batching comes with the cost of increased latency. In addition, in a system with large number of data partitions, even with message batching, the centralized timestamp authority can still become a bottleneck under heavy workloads.

The results of our micro-benchmarks show that Clock-SI has better performance than conventional SI in both LAN and WAN environments as it does less work per transaction, improving both latency and throughput. We show the results of transactions containing both read and update operations for an application benchmark in

the next section.

2.6.3 Twitter Feed-Following Benchmark

We build a Twitter-like social networking application on top of the distributed key-value store. The feed-following application supports *read-tweet* and *post-tweet* transactions on a social graph. The transactions guarantee that a user always reads consistent data. Each user in this application has a key to store the friend list, a key for the total number of tweets, and one key for each tweet. There is a trade-off between pushing tweets to the followers and pulling tweets from the followees [72]. We choose the push model, which optimizes for the more common read transactions.

We model 800,000 users. The followers of a user are located at three partitions. On average, each user follows 20 other users. The users accessed by the read-tweet and post-tweet transactions are chosen uniformly randomly. A post-tweet transaction pushes a tweet of 140 characters to the followers of a user at three different partitions. A read-tweet transaction retrieves the 20 latest tweets from the followees, which accesses one partition. The workload includes 90% read-tweet transactions and 10% post-tweet transactions, as used in prior work [73].

Figure 2.6 shows the throughput of Clock-SI and conventional SI. With eight partitions Clock-SI supports more than 38k transactions/second, while conventional SI supports 33k transactions/second. The average transaction latency of Clock-SI is also lower than that of conventional SI by one round-trip latency for the read-tweet transactions and two round-trip latency for the post-tweet transactions (figure not shown).

Since the transactions access a reasonably large data set uniformly randomly and clocks are well synchronized by NTP, transaction delays rarely occur and do not affect the overall performance. We discuss transaction delays with carefully designed workloads further below.

2.6.4 Effects of Taking Older Snapshots

In previous experiments, the delay probabilities are very small because there are few "conflicts" between reads and pending commits. Here we change the workload to create more conflicts using a small data set. We show how choosing a proper value for Δ, the snapshot age, in Clock-SI reduces the delay probability and duration. We run a group of clients, each of which issues update transactions that modify ten items. Another group of clients issue transactions that read ten of the items being updated with varying probability. We vary Δ and measure transaction throughput and latency.

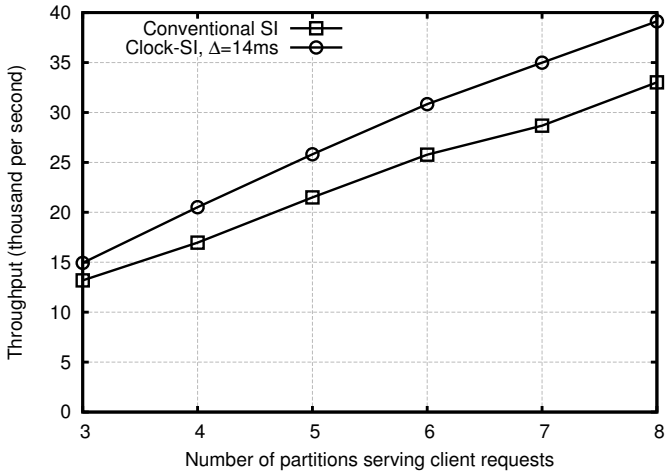

Figure 2.6 – Throughput of Twitter Feed-Following application. 90% read-tweets and 10% post-tweets.

Figure 2.7 shows the throughput of read-only transactions with different Δ values against the probability of reading hot-spot items. Figure 2.8 shows the corresponding latency. As Δ becomes larger, the throughput increases and the latency decreases. With Δ greater than the duration of a commit, reads are not delayed by updates at all (for curves with $\Delta = 14ms$ and $\Delta = 21ms$). With smaller Δ values, the probability that reads are delayed increases when the probability that a transaction reads the hot-spot items increases. As a result, the transaction latency increases and the throughput drops. Therefore, choosing a proper snapshot age in Clock-SI effectively reduces the probability of transaction delays.

2.6.5 Model Verification

We run experiments to verify our analytical model and the effects of using different Δ values in Clock-SI. Each transaction both reads and writes a fixed number of data items at one partition. We check whether the delay probability and duration change as the model in Section 2.5 predicts when we change the data set size and snapshot age.

Figure 2.9 shows that the delay probability decreases with the size of the data set. The numbers produced by the analytical model follow the same pattern as the experimental results. As each transaction accesses items uniformly randomly, the larger the data set, the less likely that a transaction reads an item updated by another committing transaction.

Next we show that taking older snapshots reduces both the transaction delay

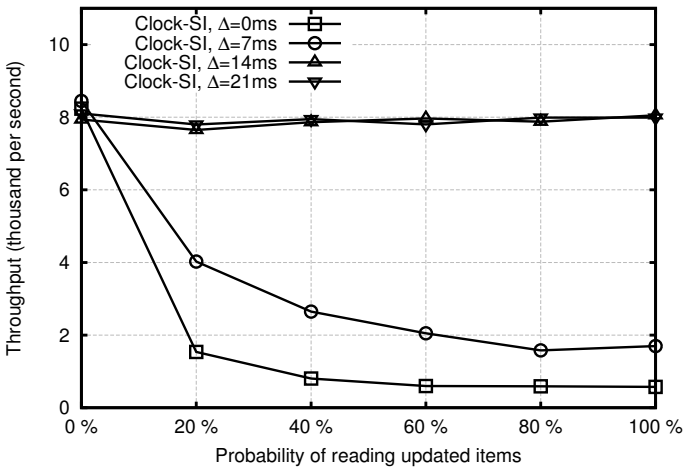

Figure 2.7 – Read transaction throughput with write hot spots.

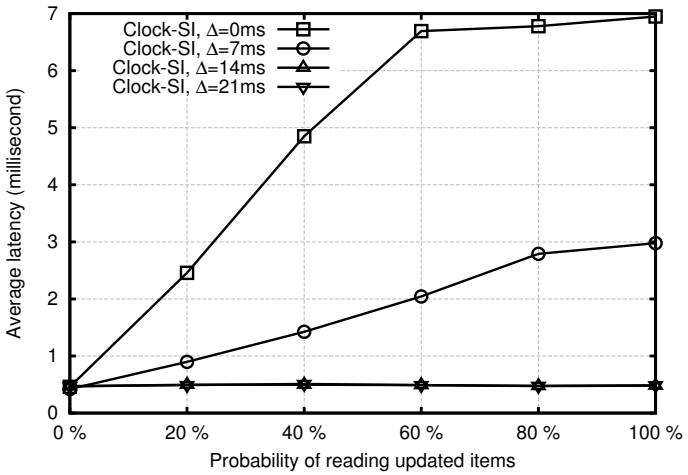

Figure 2.8 – Read transaction latency with write hot spots.

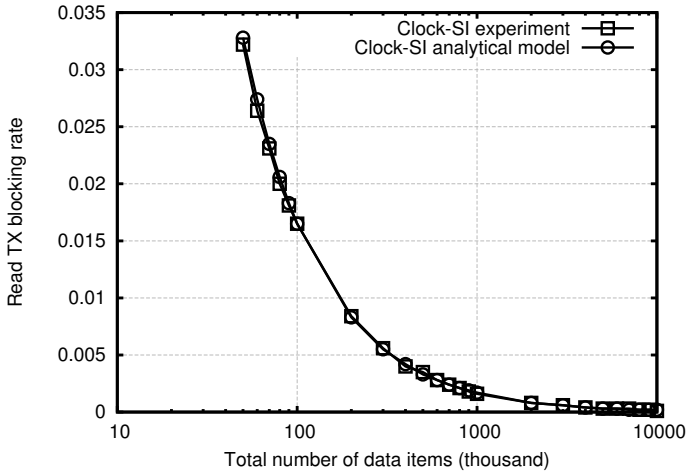

Figure 2.9 – Transaction delay rate when $\Delta = 0$ while varying the total number of data items.

probability and duration. We choose a small data set with 50,000 items to make the delays happen more often. Figure 2.10 and 2.11 demonstrate the effects of choosing different snapshot ages. The older a snapshot, the lower the delay probability and the shorter the delay duration. Figure 2.12 shows how the transaction abort rate changes. As the analytical model predicts, the transaction abort rate increases as the age of the snapshot increases.

2.6.6 NTP Precision

With Clock-SI, a transaction might be delayed when accessing a remote partition if the remote clock time is smaller than the snapshot timestamp of the transaction. The occurrence of this type of delay indicates that the clock skew is greater than one-way network latency. In all the experiments, we observe that most of the transaction delays are due to the pending commit of update transactions. Only very few delays are due to clock skew.

We measure the synchronization precision of NTP indirectly on our local cluster since it is difficult to measure the time difference of two clocks located at two different servers directly. We record the clock time, t_1, at server s_1 and immediately send a short message containing t_1 to another server s_2 over the network. After the message arrives at s_2, we record its clock time, t_2. $t_2 - t_1$ is the skew between the two clocks at s_2 and s_1 plus one-way network latency. If $t_2 - t_1 > 0$, Clock-SI does not delay transactions that originate from s_1 and access data items at s_2. Figure 2.13 shows the distribution of the clock skew between two clocks plus one-way network latency measured every 30

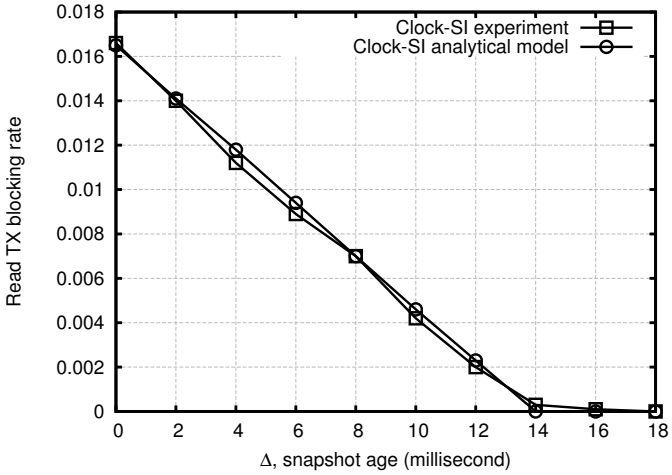

Figure 2.10 – Transaction delay rate in a small data set while varying Δ, the snapshot age.

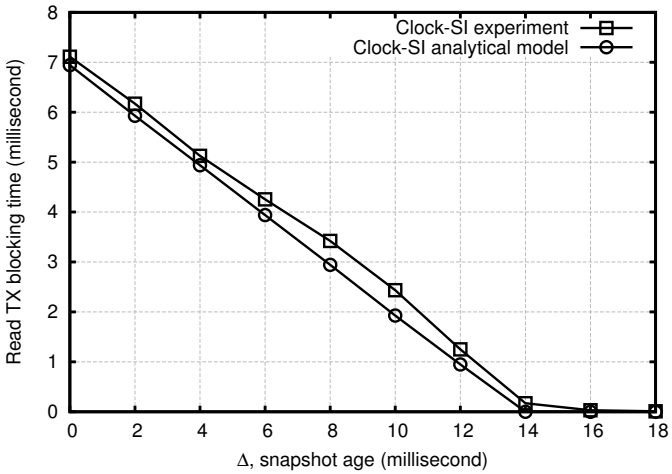

Figure 2.11 – Transaction delay time in a small data set while varying Δ, the snapshot age.

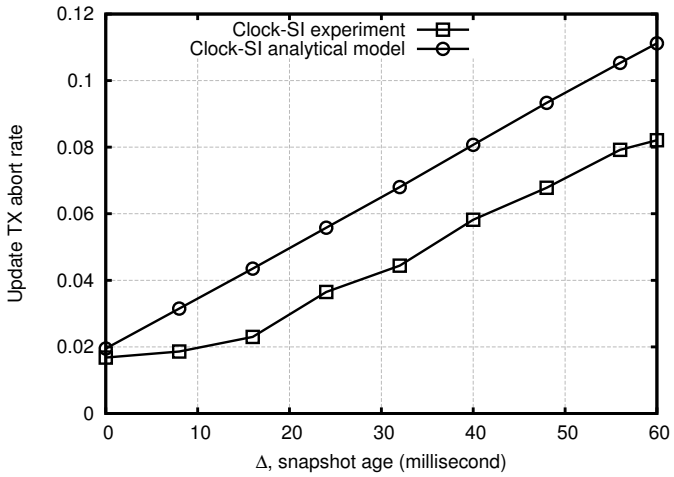

Figure 2.12 – Transaction abort rate in a small data set while varying Δ, the snapshot age.

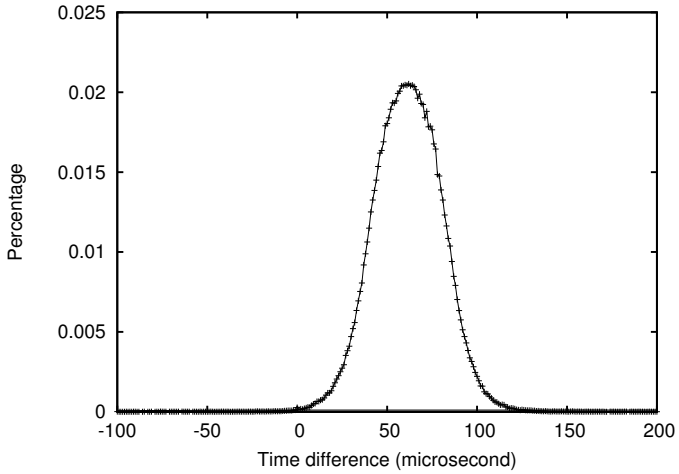

Figure 2.13 – Distribution of the clock skew between two clocks plus one-way network latency in LAN.

seconds in six weeks. A negative value on the x axis indicates the possibility of delaying a transaction when accessing a remote partition. As we see from the figure, the delay probability due to clock skew is very low and the delay duration is very short.

2.7 Summary

This chapter introduces Clock-SI, a fully distributed implementation of SI for partitioned data stores. The work described in this chapter appears in [28]. The novelty of Clock-SI is the provision of consistent snapshots using loosely synchronized clocks. Clock-SI uses physical clocks for assigning snapshot and commit timestamps. It improves over existing systems that use a centralized timestamp authority, by eliminating a single point of failure and a potential performance bottleneck. Moreover, Clock-SI avoids the round-trip latency between the partitions and the timestamp authority, showing better response times for both LAN and WAN environments.

3 Clock-RSM: Low-Latency Inter-Datacenter State Machine Replication

In this chapter, we describe Clock-RSM, a new state machine replication protocol. Clock-RSM provides linearizable low-latency replication across multiple data centers.

3.1 Introduction

Many online services replicate their data at multiple geographic locations to improve locality and availability [12,19,22]. User requests can be served at nearby replicas, thus reducing latency. By deploying replicas at multiple data centers, a system can tolerate the failure of some replicas due to server, network, and data center outage.

The state machine approach [71] is often used to replicate services consistently. All replicas execute the same set of commands in the same order deterministically. If replicas start from the same initial state, their states are consistent after executing the same sequence of commands.

3.1.1 Problem Statement

Although many protocols have been proposed to implement the state machine approach, they are not well suited for geo-replicated systems because of high replication latency. The latency of a protocol in this environment is dominated by message transmission, rather than message processing and logging. A round trip between two data centers can take hundreds of milliseconds. In contrast, message processing and logging to stable storage takes much less time, from microseconds to a few milliseconds. In addition, the latencies among data centers are not uniform and vary depending on the physical distance as well as the wide-area network infrastructure.

Multi-Paxos [52, 53] is one of the most widely used state machine replication protocols. One replica is designated as the leader, which orders commands by contacting a majority of replicas in one round of messages. A non-leader replica forwards its command to the leader and later receives the commit notification, adding the latency

of another message round, for a total of two round trips, which is significant in a wide-area setting. Mencius [61] addresses this problem by avoiding a single leader. It rotates the leader role among the replicas according to a predefined order. Committing a command in Mencius takes one round of messages. However, Mencius suffers from the *delayed commit* problem: A command can be delayed by another concurrent command from a different replica by up to one round-trip latency. In addition, in some cases a replica needs to contact the furthest replica to commit a command while Multi-Paxos only requires the leader plus a majority. With realistic non-uniform inter-data center latencies, Mencius exhibits in many cases higher latency than Multi-Paxos.

3.1.2 Solution Overview

We describe Clock-RSM, a state machine replication protocol that provides low latency for consistent replication across data centers. Clock-RSM uses loosely synchronized physical clocks at each replica to totally order commands. It avoids both the single leader required in Multi-Paxos and the delayed commit problem in Mencius. Clock-RSM requires three steps to commit a command: 1) logging the command at a majority of replicas, 2) determining the stable order of the command from the furthest replica, and 3) notifying the commit of the command to all replicas. It overlaps these steps to reduce replication latency. For many real world replica placements across data centers, Clock-RSM needs only one round-trip latency to a majority of replicas to commit a command. As Clock-RSM does not mask replica failures as in Paxos, we also introduce a reconfiguration protocol that removes failed replicas and reintegrates recovered replicas into the system.

We evaluate Clock-RSM analytically and experimentally, and compare it with Paxos-bcast and Mencius-bcast, variants of Multi-Paxos and Mencius with latency optimizations. We derive latency equations for these protocols. We also implement these protocols in a replicated key-value store and deploy it across three and five Amazon EC2 data centers. We find that Clock-RSM has lower latency than Paxos-bcast at the non-leader replicas, and similar or slightly higher latency at the leader replicas. In addition, Clock-RSM always provides lower latency than Mencius-bcast.

The key contributions of this chapter are the following:

- We describe Clock-RSM, a state machine replication protocol using loosely synchronized physical clocks (Section 3.3).
- We derive the commit latency of Clock-RSM, Multi-Paxos, Paxos-bcast, and Mencius-bcast analytically assuming non-uniform latencies (Section 3.4).
- We present a reconfiguration protocol for Clock-RSM that automatically re-

moves failed replicas and reintegrates recovered ones (Section 3.5).
– We evaluate Clock-RSM on Amazon EC2 and demonstrate its latency benefits by comparison with other protocols (Section 3.6).

3.2 Model and Definition

We describe our system model and define the properties that Clock-RSM guarantees.

3.2.1 System Model

We assume a standard asynchronous system that consists of a set of interconnected processes (clients and replicas). Processes communicate through message passing. We assume that messages are eventually delivered by their receivers, and that there is no bound on the time to deliver a message. To simplify the presentation of the protocols, we assume that messages are delivered in FIFO order.

Processes may fail by crashing and can later recover. We exclude byzantine failures. Processes have access to stable storage, which survives failures. Processes are equipped with a failure detector. We use failure detectors to ensure liveness. Failure detectors may provide wrong results, but eventually all faulty processes are suspected and at least one non-faulty process is not suspected. In practice, such a failure detector can be implemented by timeouts.

We assume the server where a replica runs is equipped with a physical clock. Clocks are loosely synchronized by a time synchronization protocol, such as NTP [2]. A clock provides monotonically increasing timestamps. The correctness of Clock-RSM does not depend on the synchronization precision.

3.2.2 State Machine Replication

State machine replication is a technique for implementing fault-tolerant services [71]. It replicates a state machine over a set of replicas. A state machine consists of a set of commands that may read and/or write the current state and produce an output. Replicas coordinate to execute commands issued by clients to achieve the desired consistency criteria.

Clients observe *linearizable* executions [39]. An execution σ is linearizable if there exists a permutation of all commands in σ such that: 1) it respects the semantics of the commands, as defined in their sequential specification; and 2) it respects the real-time ordering of commands across all clients. Linearizability can be achieved by ensuring that each replica executes commands in the *same order*. This, together with

the assumption that commands are deterministic and executed atomically, ensures that replicas transit through the same states and produce the same output for each command.

3.2.3 Geo-Replication

For a geo-replicated service, such as a data store, each replica is placed in a distinct data center. Users issue requests to their nearest data center, and the requests are handled by application servers, which contact the local replica of the service. Hence, from the point of view of the service, the clients are the application servers and they are local, i.e., they reside within the same data center as a replica of the geo-replicated service.

3.3 Clock-RSM

Clock-RSM is a multi-leader protocol. A client connects to its nearby replica within the same data center, and each replica coordinates the commands of its own clients. A replica assigns a unique timestamp to a client command, broadcasts it, and waits for the acknowledgements broadcast by other replicas once they have logged the command on their stable storage. Every replica executes commands serially in the timestamp order after they are committed. A replica knows that a command has committed if the following three conditions hold:

(1) **Majority replication.** A majority of replicas have logged the command;

(2) **Stable order.** The replica has received all commands with a smaller timestamp;

(3) **Prefix replication.** All commands with a smaller timestamp have been replicated by a majority.

Algorithm 3 gives the pseudocode of the Clock-RSM replication protocol. Table 3.1 defines the symbols used in the protocol.

3.3.1 Protocol States

Each replica maintains three hard states: (1) $Spec$, the specification of all replicas in the system; (2) $Config$, the current configuration that includes all active replicas in $Spec$, $Config \subseteq Spec$; (3) Log, the command log.

The system administrator specifies $Spec$ before the system starts, and we assume the specification of replicas is fixed during the lifetime of the system. Clock-RSM requires a majority of replicas in the specification to be non-faulty, which means $Config$ should contain at least a majority subset of $Spec$. The failure or recovery of a

Symbols	Definitions
$Spec$	all replicas, active or failed, in the system specified by the system administrator
$Config$	current configuration that includes all active replicas in $Spec$
Log	command log on stable storage
$Clock$	latest time of the physical clock
$PendingCmds$	commands pending to commit
$LatestTV$	latest clock timestamps from all replicas of $Config$, $\|LatestTV\| = \|Config\|$
$RepCounter$	command replication counter

Table 3.1 – Definition of symbols used in Algorithm 3.

replica triggers changes in $Config$. A reconfiguration protocol removes failed replicas from and adds recovered replicas to $Config$ (Section 3.5). Without loss of generality, our explanation and analysis of Clock-RSM assume that a failed replica recovers and joins the replication fast, i.e., $Spec = Config$.

Each replica also maintains some soft states when executing the protocol: (1) $PendingCmds$, a set containing the timestamps of the commands that have not been committed yet; (2) $RepCounter$, a dictionary that stores the number of replicas that have logged a command; (3) $LatestTV$, a vector of timestamps with the same size as $Config$. $LatestTV[k]$, the kth element of $LatestTV$, contains the latest known timestamp from replica r_k. It indicates that all commands originated from r_k with timestamp smaller than $LatestTV[k]$ have been received.

3.3.2 Protocol Execution

The Clock-RSM protocol is given in Algorithm 3. We explain how it totally orders and executes client commands.

1. When a replica receives ⟨REQUEST cmd⟩ from a client, where cmd is the requested command to execute, it assigns to the command its latest clock time. We call this replica the *originating* replica of the command. The replica then sends a prepare message, ⟨PREPARE cmd, ts⟩, to all replicas in $Config$ to replicate cmd. ts is the assigned timestamp that uniquely identifies and orders cmd. Ties are resolved by using the id of the command's originating replica. (lines 1-3)

2. When a replica receives ⟨PREPARE cmd, ts⟩, the logging request, from replica r_k, it adds the command to $PendingCmds$, the set of pending commands not committed yet. It updates the kth element of $LatestTV$ with ts, reflecting the latest time it knows of r_k. (lines 4-6)

Algorithm 3 Replication Protocol at Replica r_m

```
1:  upon receive ⟨REQUEST cmd⟩ from client
2:      ts ← Clock
3:      send ⟨PREPARE cmd, ts⟩ to replicas in Config

4:  upon receive ⟨PREPARE cmd, ts⟩ from r_k
5:      PendingCmds ← PendingCmds ∪ {⟨cmd, ts, k⟩}
6:      LatestTV[k] ← ts
7:      append ⟨PREPARE cmd, ts⟩ to Log
8:      wait until ts < Clock
9:      clockTs ← Clock
10:     send ⟨PREPAREOK ts, clockTs⟩ to replicas in Config

11: upon receive ⟨PREPAREOK ts, clockTs⟩ from r_k
12:     LatestTV[k] ← clockTs
13:     RepCounter[ts] ← RepCounter[ts] + 1

14: upon ∃⟨cmd, ts, k⟩ ∈ PendingCmds, s.t. COMMITTED(ts)
15:     append ⟨COMMIT ts⟩ to Log
16:     result ← execute cmd
17:     if k = m then
18:         send ⟨REPLY result⟩ to client
19:     remove ts from PendingCmds, RepCounter

20: function COMMITTED(ts)
21:     return RepCounter[ts] ≥ ⌊|Spec|/2⌋ + 1 ∧
22:         ts ≤ min(LatestTV) ∧
23:         ∄ts′ ∈ PendingCmds, s.t. ts′ < ts
```

It appends the message to its log on stable storage and acknowledges that it has logged the command with ⟨PREPAREOK $ts, clockTs$⟩, where $clockTs$ is the replica's clock time. Before acknowledging the command, the replica waits until its local clock time is greater than the timestamp of the command. That is, it promises not to send any message with a timestamp smaller than ts afterwards. To reduce the total commit latency, the acknowledgment is sent to all replicas. (lines 7-10)

The wait (at line 8) is highly unlikely with reasonably synchronized clocks, which normally provide much smaller clock skew than one-way message latency between data centers. Replicas send both PREPARE and PREPAREOK messages in timestamp order (ts in PREPARE and $clockTs$ in PREPAREOK). This guarantees replicas send monotonically increasing timestamps carried by the two types of messages.

3. When a replica receives ⟨PREPAREOK $ts, clockTs$⟩ from replica r_k, it learns its latest timestamp and updates $LatestTV$ with $clockTs$ accordingly. The replica then increments the replication counter $RepCounter[ts]$ to record the number of replicas

Algorithm 4 Periodic clock time broadcast at replica r_m

```
1: upon Clock ≥ LatestTV[m] + Δ
2:     ts ← Clock
3:     send ⟨ClockTime ts⟩ to all replicas in Config
4: upon receive ⟨ClockTime ts⟩ from replica r_k
5:     Latest[k] ← ts
```

that have logged the command with ts. $RepCounter[ts]$ has a default value of 0. (lines 11-13)

4. A replica knows that a command has committed when the following three conditions hold: (1) It receives replication acknowledgements from a majority of replicas. (2) It will not receive any message with a smaller timestamp from any replica. (3) All commands with a smaller timestamp have been replicated by a majority, and it has executed them. (lines 20-23)

When a replica learns the commit of a command with timestamp ts, it appends ⟨COMMIT ts⟩, the *commit mark*, to its log and executes the command. Commit marks are appended to the log in timestamp order. This helps a replica replay the log in the correct order during recovery, as we show in Section 3.5. If the command is from one of the replica's clients, it sends the result back to the client. Finally, the replica removes the command from $PendingCmds$ and $RepCounter$. (lines 14-19)

3.3.3 Extension

We present an extension to Algorithm 3 that further improves its latency. Algorithm 4 gives the pseudocode. Each replica periodically broadcasts its latest clock time if it does not receive frequent enough client requests. Δ is the minimum interval at which a replica broadcasts its latest clock time. If there are frequent enough client requests, a replica does not need to broadcast CLOCKTIME messages. When a replica receives a CLOCKTIME message from replica r_k, it updates $LatestTV[k]$ accordingly. This extension requires a replica to send PREPARE, PREPAREOK, and CLOCKTIME messages in timestamp order.

This extension improves latency *only* in one case: Among all replicas, only one replica serves very infrequent client requests while the other replicas do not serve client requests at all. We explain why this is the case in Section 3.4. This extension makes Clock-RSM non-quiescent, although it improves latency.

3.4 Latency Analysis

In this section we analyze the latency of Clock-RSM (Algorithm 3) and its extension (Algorithm 4). We also compare Clock-RSM with Paxos and Mencius analytically.

We assume N replicas deployed in different data centers, and denote the set of all replicas by R, which is $\{r_k \mid 0 \leq k \leq N-1\}$. We assume that latencies between replicas are non-uniform, and define $d(r_i, r_j)$ as the one-way message latency between replica r_i and r_j. We assume symmetric network latency between two replicas: $d(r_i, r_j) = d(r_j, r_i)$. Given high network latencies in a WAN, we ignore the latency introduced by local computation and disk I/O, as well as clock skew.

3.4.1 Clock-RSM

Assume that r_i is the originating replica of command cmd and assigns timestamp ts to it. As described in Section 3.3, a replica in Clock-RSM knows a command committed if three conditions hold. We analyze the latency requirement of each condition and derive the latency required to commit cmd at r_i below.

1) Majority replication requires cmd to be logged by a majority of replicas. To satisfy this condition, r_i needs to receive PREPAREOKs for cmd from a majority. The required latency is one round trip from r_i to a majority: $2 * median(\{d(r_i, r_k) \mid \forall r_k \in R\})$. We denote lc_1 the latency required to satisfy majority replication.

2) Stable order requires that cmd has the smallest timestamp among all commands that have not committed.

In the worst case, if no replica sends a message to r_i between the time that r_i assigns cmd a timestamp and cmd is logged at all replicas, r_i has to rely on the PREPAREOKs of cmd from all replicas to determine its stable order. The latency is $2 * max(\{d(r_i, r_k) \mid \forall r_k \in R\})$. We denote this latency by lc_2^{worst}.

In the best case, when r_i assigns cmd a timestamp, around the same time, if every replica sends a message with a timestamp greater than ts to r_i, then r_i knows that cmd is stable once all these messages arrive at r_i. The message can be either PREPARE or PREPAREOK. The latency is $max(\{d(r_i, r_k) \mid \forall r_k \in R\})$. We denote this latency by lc_2^{best}.

If the extension in Algorithm 4 is enabled, replicas broadcast their clock time every Δ time units. Therefore r_i determines the stable order of cmd after $lc_2^{best} + \Delta$ time units, regardless of commands being submitted concurrently. In practice, we expect Δ to be a small value. Hence lc_2^{worst} is roughly the same as lc_2^{best} with this extension enabled.

3) Prefix replication is satisfied when all commands with a smaller timestamp than

ts are replicated by a majority.

In the worst case, when r_i assigns cmd a timestamp, around the same time, if every other replica also assigns to its own command a slightly smaller timestamp than ts, then r_i needs to know that all of these commands are replicated by a majority, after the command becomes stable. That is, for each of these commands, r_i waits for its PREPAREOK message from a majority. The latency is $max(\{median(\{d(r_j, r_k) + d(r_k, r_i) \mid \forall r_k \in R\}) \mid \forall r_j \in R\})$. We denote this latency by lc_3^{worst}.

In the best case, when cmd becomes stable at r_i, if all commands with a smaller timestamp than ts have committed, this condition holds immediately. Hence it is dominated by the previous two conditions, and its latency can be ignored when computing the final commit latency. We denote by lc_3^{best} the latency in this case. We explain how this case happens later in the section.

We now derive the overall latency of committing a command under two different workloads.

Balanced workloads. Each replica serves client requests at moderate or heavy load. That is, every replica sends and receives PREPARE and PREPAREOK frequently.

This is the best case for condition 2 and the worst case for condition 3. For r_i, the latency of committing a command is $max(lc_1, lc_2^{best}, lc_3^{worst}) = max(2 * median(\{d(r_i, r_k) \mid \forall r_k \in R\}), max(\{d(r_i, r_k) \mid \forall r_k \in R\}), max(\{median(\{d(r_j, r_k) + d(r_k, r_i) \mid \forall r_k \in R\}) \mid \forall r_j \in R\}))$.

Imbalanced workloads. Only one replica serves client requests. If the workload is moderate or heavy, the replica sends PREPARE messages frequently. Since every replica broadcasts PREPAREOK, these messages of previous commands carry back the latest clock time of other replicas and help reduce the stable order duration of the current one. This is the best case for condition 2. As only one replica proposes commands, when the replica knows the current command is replicated by a majority and is stable, all previous commands must have committed. Hence it is also the best case for condition 3. The latency is $max(lc_1, lc_2^{best}, lc_3^{best}) = max(2 * median(\{d(r_i, r_k) \mid \forall r_k \in R\}), max(\{d(r_i, r_k) \mid \forall r_k \in R\}))$.

If the workload is light and the replica sends PREPARE messages infrequently, the PREPAREOK messages of previous commands do not help the stable order condition any more. This is the worst case for condition 2 and the best case for condition 3. The latency is $max(lc_1, lc_2^{worst}, lc_3^{best}) = 2 * max(\{d(r_i, r_k) \mid \forall r_k \in R\})$. With the extension in Algorithm 4 enabled, lc_2^{worst} is roughly the same as lc_2^{best}. Hence the latency becomes $max(lc_1, lc_2^{best} + \Delta, lc_3^{best}) = max(2 * median(\{d(r_i, r_k) \mid \forall r_k \in R\}), max(\{d(r_i, r_k) \mid \forall r_k \in R\}) + \Delta)$. This is the only case where enabling the extension given in Algorithm 4

helps latency.

In summary, the latency of Clock-RSM depends on the locations of the replicas and the workloads. lc_1 is the round-trip latency between the originating replica and a majority of all replicas. lc_2 is bounded by the maximum one-way latency between the originating replica and other replicas. lc_3 is bounded by the maximum two-hop latency between the originating replica and other replicas via a majority. The message complexity of Clock-RSM is $\mathcal{O}(N^2)$ as every replica broadcasts PREPAREOK messages.

3.4.2 Paxos

Multi-Paxos [52, 53] is the most widely used Paxos variant. We use Paxos to refer to Multi-Paxos in the rest of the chapter. With Paxos, one replica is designated as the leader, which coordinates replication and totally orders commands. We denote the leader in Paxos by r_l .

A non-leader replica r_i in Paxos experiences the following latency to commit a command. r_i needs $d(r_i, r_l)$ time to forward the command it proposes to leader r_l. r_l sends phase 2a messages to all replicas. All replicas reply to the leader with phase 2b messages. In order for r_l to learn that a command is committed, it waits for phase 2b messages from a majority. This takes $2 * median(\{d(r_l, r_k) \mid \forall r_k \in R\})$ time. Finally, the leader needs $d(r_l, r_i)$ time to notify r_i the commit of its command. Thus, the overall latency is $2 * d(r_i, r_l) + 2 * median(\{d(r_l, r_k) \mid \forall r_k \in R\})$. If the leader proposes a command, the latency reduces to $2 * median(\{d(r_l, r_k) \mid \forall r_k \in R\})$.

A well-known optimization allows Paxos replicas to broadcast phase 2b messages, thus saving the last message from the leader to the originating replica of a command. Replicas learn the outcome of a command without the assistance of the leader. In this case, r_i waits for phase 2b messages from a majority. The overall latency is $d(r_i, r_l) + median(\{d(r_l, r_k) + d(r_k, r_i) \mid \forall r_k \in R\})$. For the leader replica, the latency is still $2 * median(\{d(r_l, r_k) \mid \forall r_k \in R\})$. We use *Paxos-bcast* to refer to the Paxos variant that broadcasts its phase 2b message.

Although Paxos-bcast improves latency, it increases the message complexity of Paxos from $\mathcal{O}(N)$ to $\mathcal{O}(N^2)$.

3.4.3 Mencius

Mencius [61] rotates the leader role among all the replicas based on a predefined order. Each replica serves client requests at the rounds it coordinates. Mencius can also save the last step message, which is used to notify the commit of a command, by broadcasting the replication acknowledgement message. We use *Mencius-bcast* to

Protocol	Steps	Latency
Paxos	4 / 2	Leader: $2 * median(\{d(r_l, r_k) \mid \forall r_k \in R\})$ Non-leader: $2 * d(r_i, r_l) + 2 * median(\{d(r_l, r_k) \mid \forall r_k \in R\})$
Paxos-bcast	3 / 2	Leader: $2 * median(\{d(r_l, r_k) \mid \forall r_k \in R\})$ Non-leader: $d(r_i, r_l) + 2 * median(\{d(r_l, r_k) \mid \forall r_k \in R\})$
Mencius-bcast	2	Imbalanced: $2 * max(\{d(r_i, r_k) \mid \forall r_k \in R\})$ Balanced: $[q, q + max(\{d(r_i, r_k) \mid \forall r_k \in R\})]$, q is latency of Clock-RSM
Clock-RSM	2	Imbalanced: $max(2 * median(\{d(r_i, r_k) \mid \forall r_k \in R\}), max(\{d(r_i, r_k) \mid \forall r_k \in R\}))$ Balanced: $max(2 * median(\{d(r_i, r_k) \mid \forall r_k \in R\}), max(\{d(r_i, r_k) \mid \forall r_k \in R\}),$ $max(\{median(\{d(r_j, r_k) + d(r_k, r_i) \mid \forall r_k \in R\}) \mid \forall r_j \in R\}))$

Table 3.2 – Number of message steps, message complexity, and command commit latency of Paxos, Paxos-bcast, Mencius-bcast, and Clock-RSM.

refer to Mencius with this latency optimization.

Under imbalanced workloads when only one replica proposes commands, regardless of light or heavy load, Mencius-bcast always needs one round-trip message from the replica to *all* replicas to commit a command. Hence, it takes $2 * max(\{d(r_i, r_k) \mid \forall r_k \in R\})$ time to commit a command at replica r_i.

Under balanced workloads, due to the delayed commit problem, the commit latency at r_i is between q and $q + max(\{d(r_i, r_k) \mid \forall r_k \in R\})$, where q is the latency of Clock-RSM with the same workloads. Clock-RSM does not suffer from the delayed commit problem, because it uses physical clocks to assign command timestamps [1].

Therefore, compared with Clock-RSM, Mencius-bcast always requires higher latency or at most the same in "lucky" cases. Similar to Paxos-bcast, Mencius-bcast increases the message complexity of Mencius from $\mathcal{O}(N)$ to $\mathcal{O}(N^2)$.

3.4.4 Intuition and Comparison

We summarize our latency analysis of the four protocols in Table 3.2, and compare Clock-RSM and Paxos-bcast below. We do not discuss Mencius-bcast and Paxos because they have higher latency than Clock-RSM and Paxos-bcast, respectively.

At non-leader replicas, Paxos-bcast requires more message steps than Clock-RSM. If we assume that the latencies between any two replicas are the same, Clock-RSM provides lower latency. In practice, latencies among data centers are not uniform. Simply counting message steps is not sufficient to determine how a protocol performs.

1. This can be easily verified in Figure 3 in [61].

For both protocols, a replica needs to log a command at a majority. The replica that sends the logging requests, the replica that receives the logging confirmations, and the majority replicas that log the commands may be different. However, communicating with a majority means median latency, which avoids the paths of high latency between far away replicas. For Clock-RSM, the prefix replication condition also requires a majority, and it overlaps with majority replication, hence it does not increase the overall commit latency much. As a result, as long as replicas are not too far apart, such as one replica is far away from the rest, logging a command at a majority replicas does not differ much in these two protocols.

The two protocols differ mainly in command ordering. For Paxos-bcast, a non-leader replica needs one additional message to forward a command to the leader. We denote the forwarding latency by d_{fwd}. For Clock-RSM, the originating replica of a command requires a message from every other replica to determine the stable order of the command. Only a message with a greater timestamp than the command timestamp helps the stable order process. In the best case, this message is sent out roughly at the same time when the command is sent out for majority replication. Receiving the message from the furthest replica, taking $d_{max} = lc_2^{best}$ time, overlaps with the round-trip latency of majority replication, taking $2 * d_{median} = lc_1$ time. Hence, $d_{max} - 2 * d_{median}$ is the latency introduced by command ordering in Clock-RSM.

Combining the above analysis, Clock-RSM provides lower latency than Paxos-bcast at a non-leader replica as long as $d_{max} - 2 * d_{median} < d_{fwd}$, which means the highest latency is smaller than the sum of twice median latency and one forwarding latency. This condition is not demanding. Our measurements of latencies among Amazon EC2 data centers in Section 3.6 show that it holds in most cases. At the leader replica, Clock-RSM provides the same latency if $d_{max} - 2 * d_{median} \leq 0$, which means the highest latency is smaller than or equal to twice of median latency. In this case, majority replication dominates the overall commit latency. Our measurements show that this condition does not hold all the time. However, even when it does not hold, d_{max} is not much greater than $2 * d_{median}$, meaning that Clock-RSM does not lose much in those cases.

In summary, Clock-RSM provides lower latency than Paxos-bcast under conditions that are easy to satisfy in practice. In general, the more uniform the latencies among replicas are, the more likely Clock-RSM provides lower latency.

3.5 Failure Handling

To order commands, replicas rely on knowing the latest clock time of every replica in the current configuration. As a consequence, Clock-RSM may stall in case of failure of a replica or network partitions in the current configuration. In this section, we describe a reconfiguration protocol for Clock-RSM, which removes failed replicas and reintegrates recovered ones.

3.5.1 Reconfiguration

The reconfiguration protocol is given in Algorithm 5. It exposes a RECONFIGURE function that is triggered when a failure detector suspects that a replica has failed, or a recovered replica asks to rejoin. RECONFIGURE takes a new configuration as the argument, which specifies the new membership of the system after reconfiguration. The protocol uses primitives PROPOSE(k, m_p) and DECIDE(k, m_d) of consensus. In practice one can use a protocol like Paxos [52,53] to implement the primitives. A replica proposes value m_p as kth consensus instance. Eventually, all correct processes decide on the same final value m_d, among the ones proposed. We use consensus because two or more replicas may trigger RECONFIGURE with a different configuration. The protocol also introduces a new hard state: a monotonically increasing $Epoch$ number. $Epoch$ is initially 0, and is incremented after each reconfiguration. This allows us to ignore messages from older epochs, issued by replicas which have not reconfigured yet. Notice that we do not assume $Config = Spec$ in this section, because reconfiguration changes $Config$. The protocol works as follows.

1. A replica r_k that triggers RECONFIGURE sends a ⟨SUSPEND e, cts⟩ message to all replicas in $Spec$. e is the next epoch number and cts is the timestamp of the last commit mark in r_k's Log. r_k then waits for SUSPENDOK replies from a majority in $Spec$. The purpose of this phase is two-fold. First, replicas that receive a SUSPEND message stop handling PREPARE requests from other replicas and REQUEST messages from clients, essentially freezing their logs. Second, r_k collects all logged commands with a timestamp greater than cts, from a majority in $Spec$. This includes all commands that could have been committed by a failed replica. Finally, r_k invokes the eth consensus instance over all replicas in $Spec$ by proposing $config_{new}$, the next configuration to use, timestamp cts, and the above set of commands.

2. Eventually all non-faulty replicas learn about the decision for the eth consensus instance. The decision includes enough information to start a new epoch such that all replicas in $config_{new}$ start from the same state. To do so, replicas remove all entries with timestamp greater than ts from their Log, where ts is the timestamp in

the consensus decision and apply all commands that could have been committed in timestamp order. Replicas then install the new epoch number and configuration. They also resize $LatestTV$ based on the new configuration and update its elements. Finally, the normal case replication protocol can resume. Notice that some replicas may lag behind when the last commit mark in their Log is smaller than the decided timestamp. In such a case, a replica initiates a state transfer to fetch all commands up to ts, before applying the commands decided by consensus.

Algorithm 5 Reconfiguration protocol at Replica r_m

```
1:  function RECONFIGURE(config_new)
2:      e ← Epoch + 1
3:      cts ← timestamp of the last commit mark in Log
4:      send ⟨SUSPEND e, cts⟩ to all replicas in Spec
5:      wait for ⟨SUSPENDOK e, cmds_k⟩ from a majority of Spec
6:      PROPOSE(e, config_new, cts, ⋃_k cmds_k)

7:  upon receive ⟨SUSPEND e, cts⟩ from r_k
8:      stop processing REQUEST and PREPARE messages
9:      cmds ← {∀⟨cmd, ts⟩ ∈ Log | ts > cts}
10:     send ⟨SUSPENDOK e, cmds⟩ to r_k

11: upon DECIDE(e, config_new, ts, cmds)
12:     cts ← timestamp of the last commit mark in Log
13:     if ts > cts then
14:         cmds ← cmds ∪ STATETRANSFER(cts, ts)
15:     remove all ⟨PREPARE c, t⟩ from Log, s.t. t > ts and c is not executed yet
16:     for all ⟨cmd, ts⟩ ∈ cmds do                          ▷ in order of ts
17:         if ⟨PREPARE cmd, ts, k⟩ ∉ Log then
18:             append ⟨PREPARE cmd, ts⟩ to Log
19:             append ⟨COMMIT ts⟩ to Log
20:             execute cmd
21:     Epoch ← e
22:     Config ← config_new
23:     resize and update LatestTV
24:     resume processing REQUEST and PREPARE messages

25: function STATETRANSFER(from, to)
26:     send ⟨RETRIEVECMDS from, to⟩ to all replicas in Spec
27:     wait for ⟨RETRIEVEREPLY cmds_k⟩ from majority of Spec
28:     return ⋃_k cmds_k

29: upon receive ⟨RETRIEVECMDS from, to⟩ from r_k
30:     cmds ← {∀⟨cmd, ts⟩ ∈ Log | from < ts ≤ to}
31:     send ⟨RETRIEVEREPLY cmds⟩ to r_k
```

3.5.2 Recovery and reintegration

We next discuss how a replica recovers from its *Log* and is reintegrated to the existing active replicas. A replica may fail and recover in a short time, without triggering the reconfiguration process. The *Epoch* number is used to determine whether a reconfiguration has meantime happened or not. *Log* is used to replay the history of commands up to the point in which the replica failed. Recall that log entries are of two types in Clock-RSM, either PREPARE or COMMIT. A PREPARE entry contains a command with the timestamp, but they do not necessarily appear in timestamp order in *Log*. A COMMIT entry contains a timestamp only and is logged in timestamp order. In addition, Clock-RSM guarantees that a COMMIT entry is appended to *Log* after the corresponding PREPARE.

Recovery from the log proceeds as follows. The failed replica scans log entries serially, starting from the head of the log. While scanning, each PREPARE entry in the *Log* is inserted into a hash table, indexed by the entry's timestamp. Whenever a COMMIT entry with the timestamp is encountered, the corresponding PREPARE entry is removed from the hash table and executed. When the replica finishes scanning the log, it has executed all committed commands in timestamp order. However, towards the end of *Log* there might be some PREPARE entries which do not have the corresponding COMMIT in *Log*. To recover these entries, a replica sends RETRIEVECMDS to a majority of replicas in *Spec*, and only executes commands that have been logged by a majority. Finally, the replica triggers reconfiguration to join the current configuration, using Algorithm 5. Checkpointing can be used to avoid replaying the whole log and speed up the recovery process.

3.5.3 Discussion

The overall time to exclude a failed replica from the current configuration is the time to detect the failure, plus the time to reconfigure. Reconfiguration requires one initial exchange with a majority for SUSPENDing the replicated state machine, one exchange with a majority to agree on a timestamp and the set of commands that could have been committed. Some replicas might require an additional exchange for STATETRANSFER. When replicas are deployed at multiple data centers, the timeout for failure detection normally dominates the reconfiguration duration, similar to other existing protocols.

In practice failures within a replication group do not happen often, because the replication degree of a service is normally small, such as five or seven, and data centers have redundant network connections to the Internet. Therefore, we do not expect

reconfiguration to be triggered frequently and affect the availability of the replicated service.

Temporary latency variations due to network congestions on the paths between data centers may affect the commit latency of Clock-RSM. A managed WAN among data centers, which provides stable network latency, can solve this problem [41].

3.6 Evaluation

We evaluate the latency of Clock-RSM and other protocols with experiments on Amazon EC2 and numerical analysis. Our evaluation shows that: 1) Clock-RSM provides lower latency than Mencius-bcast. 2) With five and seven replicas, Clock-RSM provides lower latency than Paxos-bcast at the non-leader replicas in most cases, and it provides similar or slightly higher latency at the leader replicas. 3) With three replicas, a special case for Paxos-bcast, Clock-RSM provides similar or slightly higher (about 6% on average) latency than Paxos-bcast at all replicas.

We also evaluate the throughput of the four protocols on a local cluster. Our results show that: 1) Clock-RSM and Mencius have similar throughput for all command sizes. 2) They provide higher throughput than Paxos and Paxos-bcast for commands of large size while their throughput is lower for small and medium size commands.

3.6.1 Implementation

We implement Clock-RSM, Paxos, Paxos-bcast and Mencius-bcast in `C++`, using Google's Protocol Buffers library for message serialization. The implementation is event-driven and fully asynchronous. The protocol is divided into steps, and each step is executed by a different thread. When a thread of one step finishes processing a command, it pushes the command to the input queue of the next step. The thread executing a step batches the same type of messages being processed whenever possible. However, to avoid increasing latency, if no messages are available to be processed, it does not delay to try to batch more messages.

To evaluate the protocols, we also implement an in-memory key-value store which we replicate using the above protocols. In all experiments, clients send commands to replicas of the key-value store to update the value of a randomly selected key. Each protocol replicates the update commands and executes them in total order.

For Clock-RSM, we run NTP to keep the physical clock at each replica synchronized with a nearby public NTP server. With the proper configuration of NTP and the assistance of `clock_gettime` system call in Linux, we obtain monotonically increasing timestamps.

	VA	IR	JP	SG	AU	BR
CA	83	170	125	171	187	212
VA	-	101	215	254	220	137
IR	-	-	280	216	305	216
JP	-	-	-	77	129	368
SG	-	-	-	-	188	369
AU	-	-	-	-	-	349

Table 3.3 – Average round-trip latencies (ms) between EC2 data centers. CA, VA, EU, JP, SG, and BR correspond to California, Virginia, Ireland, Japan (Tokyo), Singapore, Australia, and Brazil (São Paulo), respectively.

3.6.2 Latency in Wide Area Replication

We evaluate the latency of Clock-RSM and compare it with other protocols with three and five replicas. We place the replicas at Amazon EC2 data centers in California (CA), Virginia (VA) and Ireland (IR), plus Japan (JP) and Singapore (SG) for the five-replica experiments. To help reason about the experimental results and numerical analysis later, we measure the round-trip latencies between data centers using `ping` and report them in Table 3.3.

We run both replicas and clients on large EC2 instances that run Ubuntu 12.04. The typical RTT in an EC2 data center is about 0.6ms. There are 40 clients issuing requests of 64B to a replica at each data center. Clients send requests in a closed loop with a think time selected uniformly randomly between 0 and 80ms. We enable the extension of Clock-RSM given in Algorithm 4 and set Δ to 5ms. We consider two types of workloads: With balanced workloads all replicas serve client requests; with imbalanced workloads only one replica serves client requests.

Balanced Workloads

The first group of experiments use balanced workloads, where clients of each replica simultaneously generate requests.

Figure 3.1 and 3.2 shows the average and 95%ile (percentile) latency at each of five replicas. Designating the replica at VA as the leader gives the best overall latency for Paxos and Paxos-bcast. Clock-RSM provides lower latency at all replicas except the leader of Paxos and Paxos-bcast.

For Paxos and Paxos-bcast, a leader replica needs only one round trip to a majority replicas to commit a command. For Clock-RSM, a replica requires contacting a majority plus the extra latency possibly introduced by the overlapping the stable order and prefix replication processes. In these two experiments, the stable order process con-

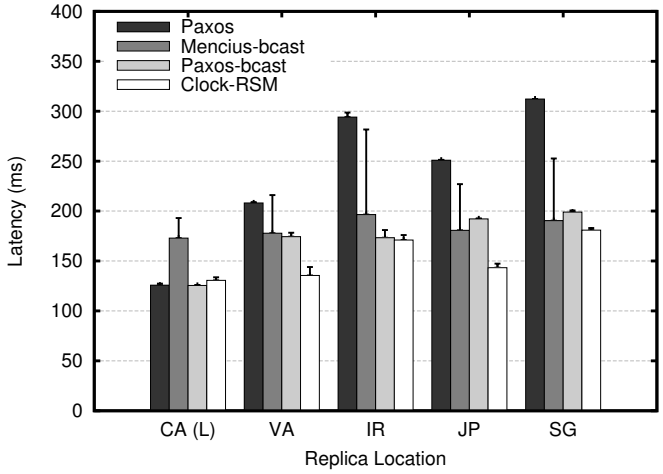

Figure 3.1 – Average (bars) and 95%ile (lines atop bars) commit latency at each of five replicas. Workload is balanced. Leader is placed at CA.

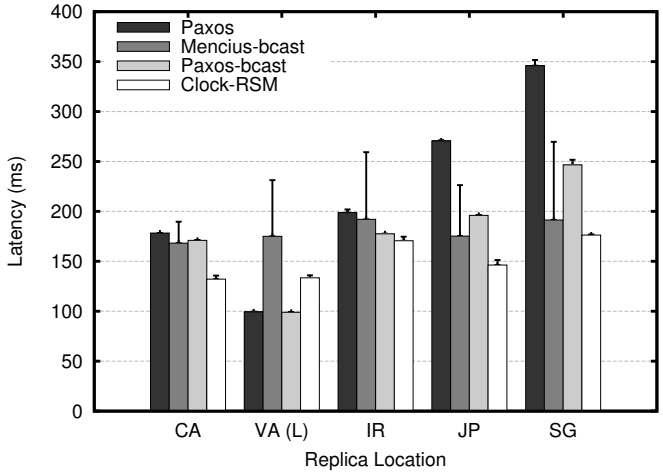

Figure 3.2 – Average (bars) and 95%ile (lines atop bars) commit latency at each of five replicas. Workload is balanced. Leader is placed at VA.

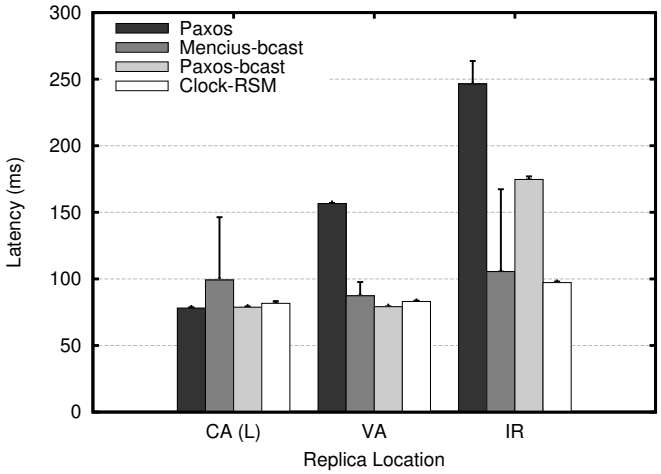

Figure 3.3 – Average (bars) and 95%ile (lines atop bars) commit latency at each of three replicas. Workload is balanced. Leader is placed at CA.

tributes to the commit latency, because the highest latencies between two replicas are quite significant. The round-trip latency between JP and IR is up to 280ms. This means the command latency at JP and IR is at least 140ms. As a consequence, Clock-RSM has higher latency at leader replicas. However, the extra latency contributed by command ordering in Clock-RSM is smaller than the latency of forwarding a command from a non-leader replica to the leader in Paxos and Paxos-bcast. Hence, Clock-RSM provides lower latency at all non-leader replicas.

We point out that the highest latency of Clock-RSM at all replicas is lower than Paxos and Paxos-bcast. The latencies of Clock-RSM at all replicas are more uniform. For the average latency of all replicas, Clock-RSM is also better.

Clock-RSM provides lower latency than Mencius-bcast at all replicas. The 95%ile latency of Mencius-bcast is much higher than its average, because the commit of a command may be delayed by another concurrent command from a different replica. The delay varies from zero up to the one-way latency between two replicas. Paxos-bcast is also better than Mencius-bcast in most cases. Mencius-bcast sometimes provides lower latency than Paxos at non-leader replicas, because it requires fewer steps and concurrent commands from all replicas help the stable order process.

Figure 3.3 and 3.4 shows the average commit latency and 95%ile latency at each of three replicas. Designating the replica at VA as the leader gives the best overall latency for Paxos and Paxos-bcast.

A three-replica setup is a *special case* for both Clock-RSM and Paxos-bcast. For Paxos-bcast, the leader replica commits a command after it receives the logging

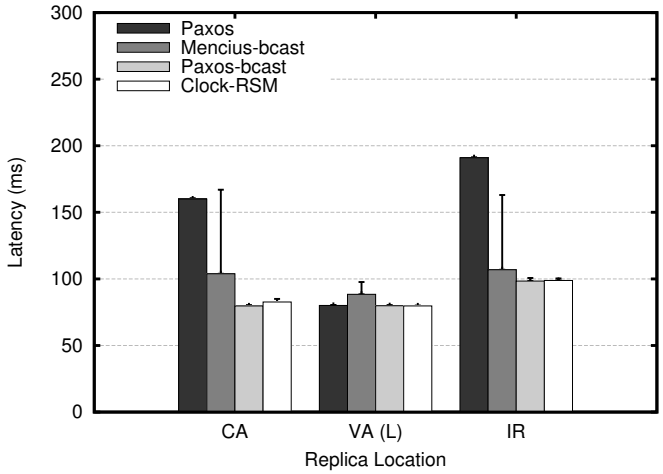

Figure 3.4 – Average (bars) and 95%ile (lines atop bars) commit latency at each of three replicas. Workload is balanced. Leader is placed at VA.

confirmation from the nearest replica. For a non-leader replica, it first forwards the command to the leader and then waits for the logging confirmation from a majority. Most likely after it logs its own command, the logging confirmation of the leader arrives, if triangle inequality still holds with latencies. Hence, all replicas in Paxos-bcast require one round trip to another replica to commit a command. When we designate the replica with the smallest weighted degree as the leader, Paxos-bcast always needs one round trip to the nearest replica to commit a command.

For Clock-RSM, prefix replication does not affect latency anymore, because it is dominated by stable order. A replica commits a command after it receives the corresponding PREPAREOK from the nearest replica (majority replication) and a greater timestamp from the furthest replica (stable order). For the three locations in this experiment, the highest latency (between VA and IR) is roughly twice of the lowest (between CA and VA). Hence, Clock-RSM also needs one round trip to the nearest replica to commit a command.

In Figure 3.3, the Paxos-bcast leader (CA) and its nearest non-leader replica (VA) have similar commit latency to Clock-RSM since they both require one round-trip messages to the nearest replica. For the other non-leader replica (IR), it has to use the longest path. Hence, the commit latency is much higher than Clock-RSM. In Figure 3.4, the Paxos-bcast leader (VA) avoids the longest path. Both protocols require one round trip to the nearest replica. Hence, they have similar latencies at all replicas.

To further clarify the latency characteristics of each protocol, we also present their latency distributions. Figure 3.5 shows the latency distribution at JP when there are

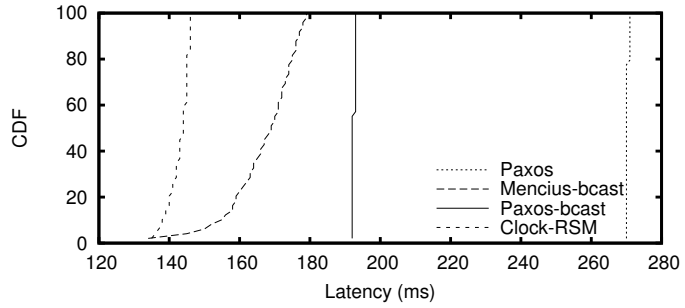

Figure 3.5 – Latency distribution at JP with five replicas. The leader is at CA. Workload is balanced.

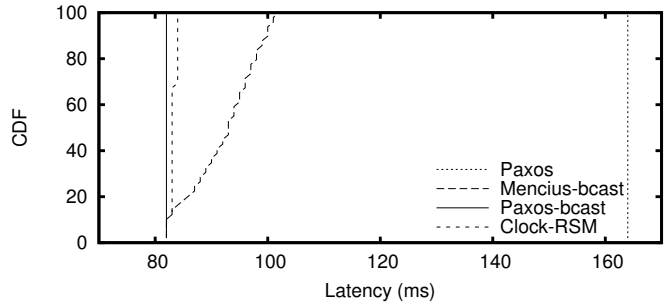

Figure 3.6 – Latency distribution at CA with three replicas. The leader is at VA. Workload is balanced.

five replicas and the leader is at CA. Both Paxos and Paxos-bcast have very predictable latency as the commit of a command is not affected by other commands. The latency of Mencius-bcast varies from 134ms to 230ms because of the delayed commit problem. Clock-RSM has some variance because, with this particular layout, the latency required by prefix replication sometimes dominates. Figure 3.6 shows the latency distribution at CA when there are three replicas and the leader is at VA. The results are similar to the ones in Figure 3.5 except that, with this replica layout, the latency of Clock-RSM almost does not vary, because prefix replication is dominated by the stable order process.

Imbalanced Workloads

We next evaluate the latency of the four protocols under imbalanced workloads. For each run of the experiment, clients issue requests to only one replica. Figure 3.7 shows the results for five replicas. This experiment is the same as the one used in Figure 3.1 except that the workload is imbalanced.

Paxos and Paxos-bcast provide the same latency for both balanced and imbalanced

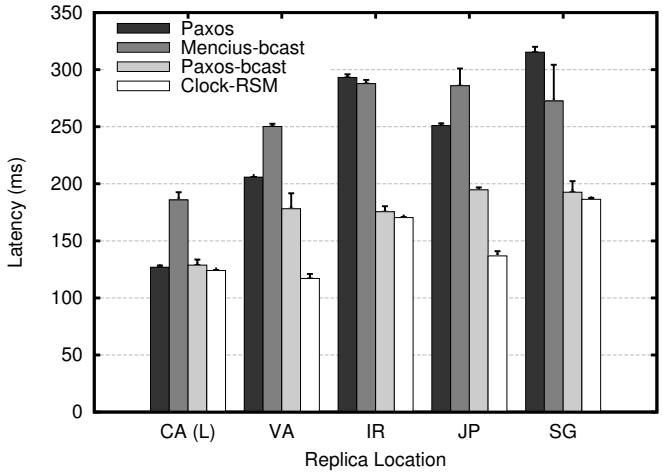

Figure 3.7 – Average (bars) and 95%ile (lines atop bars) commit latency at each of five replicas. The leader of Paxos and Paxos-bcast is at CA. Workload is imbalanced.

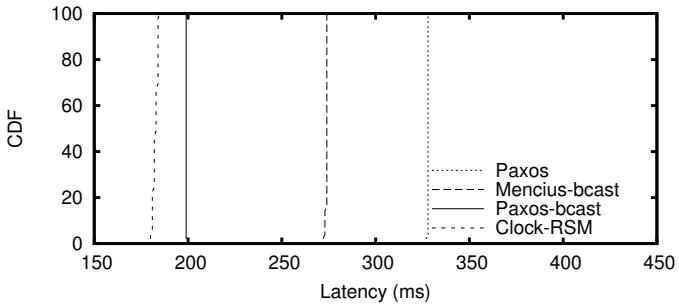

Figure 3.8 – Latency distribution at SG with five replicas. The leader of Paxos and Paxos-bcast is at CA. Workload is imbalanced.

workloads. Clock-RSM also provides similar predictable latency to both imbalanced and balanced workloads, because of the PREPAREOKs of previous commands and CLOCKTIMEs that carry the latest clock time of other replicas. The average latency of Mencius-bcast becomes much higher when it has imbalanced workloads, because Mencius-bcast needs to receive logging acknowledgements with skipped rounds from every replica to make sure that other replicas do not propose a command in a previous round. The 95%ile latency is close to the average because the delayed commit problem does not happen when there is no concurrent command at any replica. Figure 3.8 shows the latency distribution.

In summary, with realistic latencies among data centers, Clock-RSM provides lower latency in most cases. The experiment results above also confirm our analysis in Section 3.4.

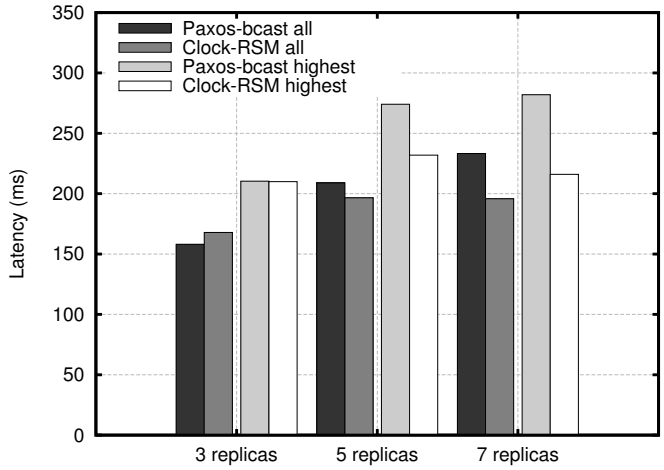

Figure 3.9 – Average commit latency. *all* includes latencies at all replicas of a group while *highest* only includes the latency at one replica that provides the highest latency.

3.6.3 Numerical Comparison of Latency

Our above experiments on EC2 show that Clock-RSM provides lower latency than other protocols in most cases with two groups of replicas of different sizes. To complete the evaluation, we compare Clock-RSM with Paxos-bcast numerically with all possible data center combinations on EC2.

We use all combinations of three, five, and seven replicas located at different EC2 data centers from Table 3.3. We plug the measured latencies in Table 3.3 into the latency formulas in Table 3.2. Paxos-bcast always chooses the best leader replica that provides the lowest average latency of all replicas in the group.

Figure 3.9 shows the average latency of replicas from all groups of the same size. We compute two types of average latency: average *all* latency includes latencies at all replicas of a group while average *highest* latency only includes the latency at one replica that provides the highest latency in the group. As the figure shows, Clock-RSM provides lower latency for both five and seven replicas. Its improvement for the average highest latency is greater, because for Paxos-bcast, latencies at different replicas are more spread out. The latency at a non-leader replica is much higher than the leader replica, as it needs one extra message to forward a command to the leader. In contrast, latencies in Clock-RSM are closer to each other because every replica requires the same number of steps to commit a command.

With three replicas, Paxos-bcast is slightly better than Clock-RSM, because we always choose the best leader for it, which leads to optimal commit latency at all replicas. This also validates our previous analysis of the protocols with three replicas.

	Replica Percentage	Absolute Reduction	Relative Reduction
3 replicas	0.0%	0.0ms	0.0%
	100.0%	-9.9ms	-6.2%
5 replicas	68.6%	31.9ms	15.2%
	31.4%	-30.6ms	-14.6%
7 replicas	85.7%	50.2ms	21.5%
	14.3%	-39.4ms	-16.9%

Table 3.4 – Latency reduction of Clock-RSM over Paxos-bcast. Negative latency reduction means Clock-RSM provides higher latency.

Table 3.4 shows the latency reduction of Clock-RSM over Paxos-bcast at all replicas for different replication groups. For instance, for all replicas in the groups with five replicas, the latency of Clock-RSM at 68.6% of the replicas is lower than Paxos-bcast. On average, it reduces the latency by 31.9ms, i.e., by 15.2%, at those replicas. For 31.4% of the replicas, the latency of Clock-RSM is higher. On average, it increases the latency by 30.6ms, i.e., by 14.3%, at those replicas. We look into all these 31.4% replicas and find that most of them are the leader replica in their group and a few are the non-leader replica that are very close to the leader. For groups with seven replicas, we have similar results. For groups with three replicas, the latency of Paxos-bcast is slightly better, because it provides optimal latency in this special case.

3.6.4 Throughput on A Local Cluster

Although the goal of Clock-RSM is to provide low commit latency in a WAN environment, for completeness, we also evaluate its throughput and compare it with other protocols. Our experimental results show that Clock-RSM has competitive throughput.

To avoid the network bandwidth limit on EC2 across data centers, we run experiments on a local cluster. Each server has two Intel Xeon processors with 4GB DDR2 memory. All servers are connected to a single Gigabit Ethernet switch. A replica runs on one server exclusively. Replicas log commands to main memory to avoid the disk being the bottleneck. Clients send commands to all replicas with sufficient frequency to saturate them.

Figure 3.10 reports the throughputs for five replicas and command sizes of 10B, 100B, and 1000B. In all cases, CPU is the bottleneck and message sending and receiving is the major consumer of CPU cycles.

For 10B and 100B commands, Paxos and Paxos-bcast have higher throughput than Mencius-bcast and Clock-RSM, because all the non-leader replicas forward

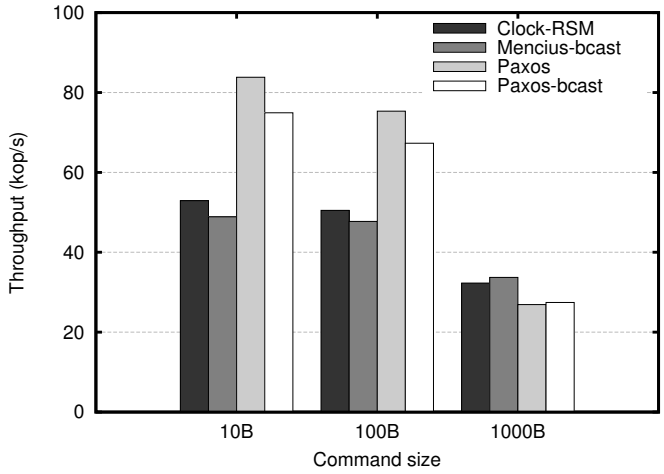

Figure 3.10 – Throughput for small (10B), medium (100B), and large (1000B) commands with five replicas on a local cluster.

commands to the leader in Paxos and Paxos-bcast, which can batch more commands when sending and receiving messages. For large commands of size 1000B, Paxos and Paxos-bcast have lower throughput. The leader replica becomes the performance bottleneck, because batching large messages does not help throughput very much anymore.

Clock-RSM and Mencius have similar throughput as they have the same communication pattern and message complexity. Paxos provides better throughput than Paxos-bcast because its message complexity is lower, but the improvement is not significant since Paxos requires one more message step.

Our measurements do not support the claim, made in other works [61, 63], that a multi-leader protocol always provides better throughput than Paxos because the leader in Paxos is the performance bottleneck. When replicas batch messages opportunistically, without waiting intentionally, the leader replica of Paxos has more chances to batch and hence increases throughput in the case of small and medium commands. Prior work evaluates throughput using different implementations or configurations. For the evaluations in Mencius [61], replicas do not batch messages. For the experiments in Egalitarian Paxos [63], although replicas batch messages, the Paxos leader handles all client messages, while the other replicas only process replication messages. Hence, the leader limits the throughput of the whole system because it processes considerably more messages than the other replicas.

3.7 Correctness

We provide a proof sketch for the correctness of Clock-RSM (Algorithm 3) and its reconfiguration protocol (Algorithm 5, Section 3.5). We first argue that replicas execute commands in the same order and that every correct replica executes every command (agreement). Then, we show that linearizability, as defined in Section 3.2.2, follows from the above properties.

Claim 1. *If any replica r_m executes command c_i followed by command c_j with timestamps ts_i and ts_j respectively, then $ts_i < ts_j$.*

Proof sketch. Suppose replica r_m executes command c_i (at line 16 of alg. 1). We distinguish two cases:

Case (a): r_m's $PendingCmds$ contains $\langle c_j, ts_j, k\rangle$. Notice that function COMMITTED(ts) evaluates to *true* only if the given timestamp ts is the smallest timestamp among all commands in $PendingCmds$ (line 23, alg. 1). Since c_i is executed, the invocation COMMITTED(ts_i) must have returned *true*. Therefore $ts_i < ts_j$.

Case (b): r_m's $PendingCmds$ does not contain $\langle c_j, ts_j, k\rangle$. Assume c_j originated at replica r_k. By line 22 of alg. 1, $ts_i \leq LatestTV[k]$, which implies r_m received a PREPARE or a PREPAREOK message from r_k tagged with a timestamp greater than ts_i. Since channels are FIFO and messages are sent in timestamp order, $LatestTV[k] < ts_j$. Therefore $ts_i < ts_j$.

Claim 2. (Total order). *If a replica executes commands c_i and c_j, in this order, then no replica executes c_j before c_i.*

Proof sketch. By Claim 1, any replica executes commands in clock timestamp order. What remains to show is that the timestamp order is a total order. Replicas assign monotonically increasing clock timestamps to each command. The clock timestamp, together with the unique replica *id* forms a total order. Moreover, timestamps are assigned once by one replica, and never change.

Claim 3. *If any replica executes command c in epoch e, then every correct replica in epoch $e+1$ has executed c.*

Proof sketch. Assume that some replica has executed command c with timestamp ts in epoch e. It means that a majority of replicas has acknowledged a message $\langle$PREPARE $c, ts\rangle$ and logged it to their stable storage.

Let r be the replica that triggers function RECONFIGURE (Algorithm 5) in epoch e and whose proposal (line 6, alg. 3) is decided by all correct replicas. We next consider cts, the largest commit timestamp in r's Log, and distinguish two cases:

Case (a): $ts > cts$. That is, r has not executed c yet. In this case r fetches all commands with timestamps greater than cts from a majority of replicas (lines 4-5, alg. 3). Since any two majorities intersect it must be that at least one replica returned command c to replica r. Replica r included command c in its consensus proposal (line 6, alg. 3), and all correct replicas eventually deliver c.

Case (b): $ts \leq cts$. In this case r has already executed c and therefore does not include c in its proposal. All correct replicas eventually deliver the consensus decision for epoch $e+1$. If a replica has not executed c yet, its last commit mark in the log is smaller than ts (line 13, alg. 3), in which case it fetches c from a majority of replicas in function STATETRANSFER.

In both cases, a replica has either already executed c or its set of commands $cmds$ after line 14 of alg. 3 includes c, in which case c is executed in lines 16-20 of alg. 3 before transitioning to epoch $e+1$.

Claim 4. (Agreement) *If a replica executes command c, then every correct replica eventually executes c.*

Proof sketch. Suppose replica r executes command c. We have to show that every correct replica eventually executes c, both during normal case operation and across subsequent epochs. Algorithm 3 ensures that during normal case operation every replica eventually delivers PREPARE message and therefore replicas include c in their set of pending commands. Since timestamps are monotonically increasing, it must be that c eventually becomes the command with smallest timestamp (line 23, alg. 1), and that all replicas have proposed or reported higher timestamps (line 22, alg. 1). Finally, every replica will receive enough acknowledgments (line 21, alg. 1) to execute c. In case of failures, we rely on a correct replica that triggers function RECONFIGURE. And by Claim 3, any command that has committed in the current epoch will be executed by every correct replica before transitioning to the subsequent epoch.

Claim 5. (Linearizability) *Clock-RSM is linearizable.*

Proof sketch. Let σ be an execution of client commands that consists of $\langle$REQUEST $cmd\rangle$ and their corresponding $\langle$REPLY $result\rangle$. We have to show that there exists a permutation π of σ such that: 1) π respects the semantics of the commands, as defined in their sequential specification; and 2) π respects the real-time ordering of commands across all clients. Let π be a permutation of σ ordered according to the clock timestamp ordering provided by Algorithm 3.

We first show that π respects the sequential semantics of the commands. The replication protocol executes commands in total order (Claim 2), and replicas execute

each and every command (Claim 4). Which means that every replica executes the *upon* clause of lines 14-19 of alg. 1 for each command in the same order. Moreover, every command is execute serially, one command at a time, thus satisfying the semantics of commands.

We next claim that π satisfies the real-time ordering of commands in σ. Suppose command c_i finishes before command c_j begins in σ. This implies that c_i's client has received a reply for c_i before c_j is submitted to a replica. Obviously, Algorithm 3 orders c_j after c_i. Thus c_i precedes c_j in π.

3.8 Summary

In this chapter, we introduce Clock-RSM, a state machine replication protocol that pushes the latency limit for strongly consistent replication. The work described in this chapter appears in [29]. Clock-RSM relies on loosely synchronized clocks to reduce latency. We evaluate our protocol extensively with realistic workloads, where latencies among data centers are non-uniform. We show that, compared with state of the art protocols, Clock-RSM reduces latency in most cases with real world replica placements.

4 Orbe: Scalable Causal Consistency

In this chapter, we introduce Orbe, a partitioned and replicated data store that implements causal consistency in a scalable and efficient fashion.

4.1 Introduction

Distributed data stores are a critical infrastructure component of many online services. Choosing a consistency model for such data stores is difficult. The CAP theorem [17, 34] shows that among Consistency, Availability, and (network) Partition-tolerance, a replicated system can only have two properties out of the three.

A strong consistency model, such as *linearizability* [39] and *sequential consistency* [51], does not allow high availability under network partitions. In contrast, *eventual consistency* [76], a weak model, provides high availability and partition-tolerance, as well as low update latency. It guarantees that replicas eventually converge to the same state provided no updates take place for a long time. However, it does not guarantee any order on applying replicated updates. *Causal consistency* [7] is weaker than sequential consistency but stronger than eventual consistency. It guarantees that replicated updates are applied at each replica in an order that preserves causality [7,50] while providing availability under network partitions. Furthermore, client operations have low latency because they are executed at a local replica and do not require coordination with other replicas.

4.1.1 Problem Statement

The problem addressed in this chapter is providing a scalable and efficient implementation of causal consistency for both partitioned and replicated data stores. Most existing causally consistent systems [48, 60, 68] adopt variants of *version vectors* [7, 65], which are designed for purely replicated data stores. Version vectors do not scale when partitioning is added to support a data set that is too large to fit on a single server.

They still view all partitions as one logical replica and require a single serialization point across partitions for replication, which limits the replication throughput [58]. A scalable solution should allow replicas of different partitions to exchange updates in parallel without serializing them at a centralized component.

COPS [58] identifies this problem and provides a solution that explicitly tracks causal dependencies at the client side. A client stores every accessed item as dependency metadata and associates this metadata with each update operation issued to the data store. When an update is propagated from one replica to another for replication, it carries the dependency metadata. The update is applied at the remote replica only when all its dependencies are satisfied at that replica. COPS provides good scalability. However, tracking every accessed item explicitly can lead to large dependency metadata, which increases storage and communication overhead and affects throughput. Although COPS employs a number of techniques to reduce the size of dependency metadata, it does not fundamentally solve the problem. When supporting causally consistent read-only transactions, the dependency metadata overhead is still high under many workloads.

4.1.2 Solution Overview

In this chapter, we present two protocols and one optimization that provide a scalable and efficient implementation of causal consistency for both partitioned and replicated data stores.

The first protocol uses two-dimensional *dependency matrices* (DMs) to compactly track dependencies at the client side. We call it *the DM protocol*. This protocol supports basic read and update operations. It associates with each update the dependency matrix of its client session. Each element in a dependency matrix is a scalar value that represents all dependencies from the corresponding data store server. The size of dependency matrices is bounded by the total number of servers in the system. Furthermore, the DM protocol resets the dependency matrix of a client session after each update operation, because prior dependencies need not to be tracked due to the transitivity of causality. With sparse matrix encoding, the DM protocol keeps the dependency metadata of each update small.

The second protocol extends the DM protocol to support causally consistent read-only transactions by using loosely synchronized physical clocks. We call it *the DM-Clock protocol*. In addition to the dependency metadata required by the DM protocol, this protocol assigns to each state an update timestamp obtained from a local physical clock and guarantees that the update timestamp order of causally related states is

consistent with their causal order. With this property, the DM-Clock protocol provides causally consistent snapshots of the data store to read-only transactions by assigning them a snapshot timestamp, which is also obtained from a local physical clock.

We also propose *dependency cleaning*, an optimization that further reduces the size of dependency metadata in the DM and DM-Clock protocols. It is based on the observation that once a state and its dependencies are fully replicated, any subsequent read on the state does not introduce new dependencies to the client session. More messages need to be exchanged, however, for a server to be able to decide that a state is fully replicated, which leads to a tradeoff which we study. Dependency cleaning is a general technique and can be applied to other causally consistent systems.

We implement the two protocols and dependency cleaning in *Orbe*, a distributed key-value store, and evaluate them experimentally. Our evaluation shows that Orbe scales out as the number of data partitions increases. Compared with an eventually consistent system, it incurs relatively little performance overhead for a large spectrum of workloads. It outperforms COPS under many workloads when supporting read-only transactions.

In this chapter, we make the following contributions:

- The DM protocol that provides scalable causal consistency and uses dependency matrices to keep the size of dependency metadata under control.
- The DM-Clock protocol that provides read-only transactions with causal snapshots using loosely synchronized physical clocks by extending the DM protocol.
- The dependency cleaning optimization that reduces the size of dependency metadata.
- An implementation of the above protocols and optimization in Orbe as well as an extensive performance evaluation.

4.2 Model and Definition

4.2.1 Architecture

We assume a distributed key-value store that manages a large set of data items. The key-value store provides two basic operations to the clients:

- PUT(key, val): A PUT operation assigns value val to an item identified by key. If item key does not exist, the system creates a new item with initial value val. If key exists, a new version storing val is created.
- val ← GET(key): The GET operation returns the value of the item identified by key.

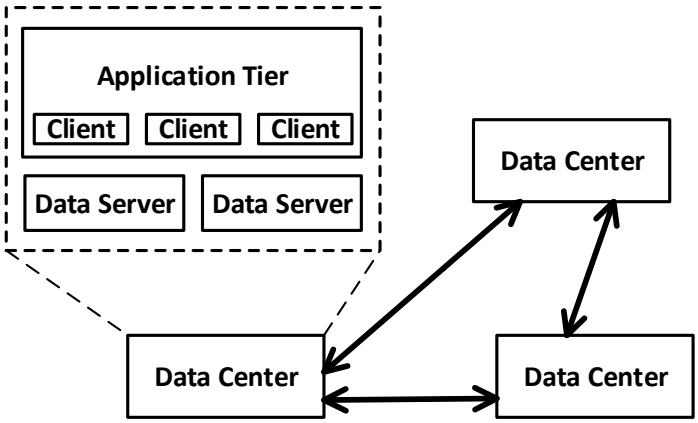

Figure 4.1 – System architecture. The data set is replicated by multiple data centers. Clients are collocated with the data store in the data center and are used by the application tier to access the data store.

An additional operation that provides read-only transactions is introduced later in Section 4.4.

The data store is partitioned into N partitions, and each partition is replicated by M replicas. A data item is assigned to a partition based on the hash value of its key. In a typical configuration, as shown in Figure 4.1, the data store is replicated at M different data centers for high availability and low operation latency. The data store is fully replicated. All N partitions are present at each data center.

The application tier relies on the clients to access the underlying data store. A client is collocated with the data store servers in a particular data center and only accesses those servers in the same data center. A client does not issue the next operation until it receives the reply to the current one. Each operation happens in the context of a client session. A client session maintains a small amount of metadata that tracks the dependencies of the session.

4.2.2 Causal Consistency

Causality is a *happens-before* relationship between two events [7, 50]. We denote causal order by $\rightsquigarrow$. For two operations a and b, if $a \rightsquigarrow b$, we say b depends on a or a is a dependency of b. $a \rightsquigarrow b$ if and only if one of the following three rules holds:

1. Thread-of-execution. a and b are in a single thread of execution. a happens before b.
2. Reads-from. a is a write operation and b is a read operation. b reads the state created by a.

3. Transitivity. There is some other operation c that $a \rightsquigarrow c$ and $c \rightsquigarrow b$.

We define the *nearest dependencies* of a state as all the states that it directly depends on, without relying on the transitivity of causality.

To provide causal consistency when replicating updates, a replica does not apply an update propagated from another replica until all its causal dependency states are installed locally. The original definition of causal consistency does not require that replicas of the same partition eventually reach the same state [7]. *Causal+ consistency*, a stronger consistency model, requires convergent conflict handling on top of causal consistency [58]. For convenience, we use causal consistency to refer to causal+ consistency in this thesis.

4.3 DM Protocol

In this section, we present the DM protocol that provides scalable causal consistency using dependency matrices. This protocol is scalable because replicas of different partitions exchange updates for replication in parallel, without requiring a global serialization point.

Dependency matrices are, for systems that support both replication and partitioning, the natural extension of version vectors, for systems that support only replication. A row of a dependency matrix is a version vector that stores the dependencies from replicas of a partition.

A client stores the nearest dependencies of its session in a dependency matrix and associates it with each update request to the data store. After a partition at the client's local data center executes the update, it propagates the update with its dependency matrix to the replicas of that partition at remote data centers. Using the dependency matrix of the received update, a remote replica waits to apply the update until all partitions at its data center store the dependency states of the update.

4.3.1 Definitions

The DM protocol introduces dependency tracking data structures at both the client and server side. It also associates dependency metadata for each item. Table 4.1 provides a summary of the symbols used in the protocol. We explain their meanings in detail below.

Client States. Without losing generality, we assume a client has one session to the data store. A client c maintains for its session a dependency matrix, DM_c, which consists of $N \times M$ non-negative integer elements. DM_c tracks the nearest dependencies of a client session. $DM_c[n][m]$ indicates that the client session potentially depends

Symbols	Definitions
N	number of partitions
M	number of replicas per partition
c	a client
DM_c	dependency matrix of c, $N \times M$ elements
PDT_c	physical dependency timestamp of c
p_n^m	a server that runs m^{th} replica of n^{th} partition
VV_n^m	(logical) version vector of p_n^m, M elements
PVV_n^m	physical version vector of p_n^m, M elements
$Clock_n^m$	current physical clock time of p_n^m
d	an item, tuple $\langle k, v, ut, put, dm, rid \rangle$
k	item key
v	item value
ut	(logical) update timestamp
put	physical update timestamp
dm	dependency matrix, $N \times M$ elements
rid	source replica id
t	a read-only transaction, tuple $\langle st, rs \rangle$
st	(physical) snapshot timestamp
rs	readset, a set of read items

Table 4.1 – Definition of symbols.

on the first $DM_c[n][m]$ updates at partition p_n^m, the mth replica of the nth partition.

Server States. Each partition maintains a *version vector* (VV) [7, 65]. The version vector of partition p_n^m is VV_n^m, which consists of M non-negative integer elements. $VV_n^m[m]$ counts the number of updates p_n^m has executed locally. $VV_n^m[i]$ $(i \neq m)$ indicates that p_n^m has applied the first $VV_n^m[i]$ updates propagated from p_n^i, a replica of the same partition.

A partition updates an item by either executing an update request from its clients or by applying a propagated update from one of its replicas at other data centers. We call the partition that updates an item to the current value by executing a client request the *source partition* of the item.

Item Metadata. We represent an item d as a tuple $\langle k, v, ut, dm, rid \rangle$. k is a unique key that identifies the item. v is the value of the item. ut is the *update timestamp*, the logical creation time of the item at its source partition. dm is the dependency matrix, which consists of $N \times M$ non-negative integer elements. $dm[n][m]$ indicates that d potentially depends on the first $dm[n][m]$ updates at partition p_n^m, a prefix of its update history. rid is the *source replica id*, the replica id of the item's source partition.

We use sparse matrix encoding to encode dependency matrices. Zero elements in

a dependency matrix do not use up any space after encoding. Only non-zero elements contribute to the actual size.

4.3.2 Protocol

We now describe how the DM protocol executes GET and PUT operations and replicates PUTs.

GET. Client c sends a request $\langle \text{GET } k \rangle$ to a partition at the local data center, where k is the key of the item to read. Upon receiving the request, partition p_n^m obtains the read item, d, and sends a reply $\langle \text{GETREPLY } v_d, ut_d, rid_d \rangle$ back to the client. Upon receiving the reply, the client updates its dependency matrix: $DM_c[n][rid_d] \leftarrow max(DM_c[n][rid_d], ut_d)$. It then hands the read value, v_d, to the caller of GET.

PUT. Client c sends a request $\langle \text{PUT } k, v, DM_c \rangle$ to a partition at the local data center, where k is the item key and v is the update value. Upon receiving the request, p_n^m performs the following steps: 1) Increment $VV_n^m[m]$; 2) Create a new version d for the item identified by k; 3) Assign item key: $k_d \leftarrow k$; 4) Assign item value: $v_d \leftarrow v$; 5) Assign update timestamp: $ut_d \leftarrow VV_n^m[m]$; 6) Assign dependency matrix: $dm_d \leftarrow DM_c$; 7) Assign source replica id: $rid_d \leftarrow m$. These steps form one atomic operation, and none of them is blocking. p_n^m stores d on stable storage and overwrites the existing version if there is one. It then sends a reply $\langle \text{PUTREPLY } utd_d, rid_d \rangle$ back to the client. Upon receiving the reply, the client updates its dependency matrix: $DM_c \leftarrow \mathbf{0}$ (reset all elements to zero) and $DM_c[n][rid_d] \leftarrow ut_d$.

Update Replication. A partition propagates its local updates to its replicas at remote data centers in update timestamp order. To replicate a newly updated item, d, partition p_n^s sends an update replication request $\langle \text{REPLICATE } k_d, v_d, ut_d, dm_d, rid_d \rangle$ to all other replicas.

A partition also applies updates propagated from other replicas in their update timestamp order. Upon receiving the request, partition p_n^m guarantees causal consistency by performing the following steps:

1. p_n^m checks if it has installed the dependency states of d specified by $dm_d[n]$. p_n^m waits until $VV_n^m \geqslant dm_d[n]$, i.e., $VV_n^m[i] \geqslant dm_d[n][i]$, for $0 \leqslant i \leqslant M-1$.
2. p_n^m checks if causality is satisfied at the other local partitions. p_n^m waits until, $VV_j^m \geqslant dm_d[j]$, for $0 \leqslant j \leqslant N-1$ and $j \neq n$. It needs to send a message to p_j^m for dependency checking if $dm_d[j]$ contains at least one non-zero element.
3. If there is currently no value stored for item k_d at p_n^m, it simply stores d. If p_n^m has an existing version d' such that $k_{d'} = k_d$, it orders the two versions deterministically by concatenating the update timestamp (high order bits) and

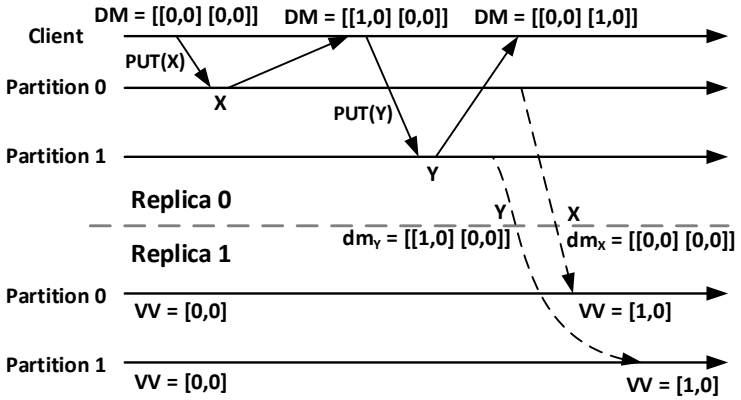

Figure 4.2 – An example of the DM protocol with two partitions replicated at two data centers. A client updates item X and then item Y at two partitions. X and Y are propagated concurrently, but their installation at the remote data center is constrained by causality.

source replica id (low order bits). If d is ordered after d', p_n^m overwrites d' with d. Otherwise, d is discarded.

4. p_n^m updates its version vector: $VV_n^m[s] \leftarrow ut_d$. As updates are propagated in order, we have the invariant before $VV_n^m[s]$ is updated: $VV_n^m[s]+1 = ut_d$.

Example. We give an example to explain the necessity for dependency matrices. Consider a system with two partitions replicated at two data centers ($N = 2$ and $M = 2$) in Figure 4.2. The client first updates item X at the local data center. The update is handled by partition p_0^0. Upon receiving a response the client updates item Y at partition p_1^0. Causality through the client session dictates that $X \rightsquigarrow Y$. At the other data center, p_1^1 applies Y only after p_0^1 applies X. The figure shows the dependency matrices propagated with the updates. p_1^1 waits until the version vector of p_0^1 is no less than $[1,0]$, which guarantees that X has been replicated.

Correctness. The DM protocol uses dependency matrices to track the nearest dependencies of a client session or an item. It resets the dependency matrix of a client session after each PUT to keep its size after encoding small. This does not affect correctness. By only remembering the update timestamp of the PUT, the protocol utilizes the transitivity of causality to track dependencies correctly.

An element in a dependency matrix, a scalar value, is the maximum update timestamp of all the nearest dependency items from the corresponding partition. Since a partition always propagates local updates to and applies remote updates from other replicas in their update timestamp order, once it applies an update from another

replica, it must have applied all the other updates with a smaller update timestamp from the same replica. Therefore, if a partition satisfies the dependency requirement specified by the dependency matrix of an update, it must have installed all the dependencies of the update.

4.3.3 Cost over Eventual Consistency

Compared with typical implementations of eventual consistency, the DM protocol introduces some overhead to capture causality. This is a reasonable price to pay for the stronger semantics.

Storage Overhead. The protocol keeps a dependency matrix and other metadata for each item. The per-item dependency metadata is the major storage overhead of causal consistency. We keep it small by only tracking the nearest dependencies and compressing it by sparse matrix encoding. In addition, the protocol requires that a client session maintains a dependency matrix and a partition maintains a version vector. These global states are small and negligible.

Communication Overhead. Checking dependencies of an update during replication requires a partition to send a maximum of $N-1$ messages to other local partitions. If a row in the dependency matrix contains all zeros, then the corresponding partition does not need to be checked, since the update does not *directly* depend on any states managed by all replicas of that partition. In addition, the dependency metadata carried by update replication messages also contributes to inter-datacenter network traffic.

4.3.4 Conflict Detection and Resolution

The above protocol orders updates of the same item deterministically by using the update timestamp and source replica id. If there are no updates for a long enough time, replicas of the same partition eventually converge to the same state while respecting causality. However, the dependency metadata does not indicate whether two updates from different replicas are conflicting or not.

To support conflict detection, we extend the DM protocol by introducing one more element to the dependency metadata: the item dependency timestamp. We denote the item dependency timestamp of an item d by idt_d. When d's source partition p_n^s creates d, it assigns to idt_d the update timestamp of the existing version of item k_d if it exists or -1 otherwise. When partition p_n^m applies d after dependency checking, it handles the existing version d' as below. If d is created after d' is replicated at p_n^s, $idt_d = ut_{d'}$, then d and d' do not conflict. p_n^m overwrites d' with d. If d and d' are

created concurrently by different replicas, $idt_d \neq ut_{d'}$, they conflict. In that case, either the application can be notified to resolve the conflict using application semantics, or p_n^m orders the two conflicting updates deterministically as outlined in Section 4.3.2.

4.4 DM-Clock Protocol

The DM-Clock protocol extends the DM protocol to support causally consistent read-only transactions. Many applications can benefit from a programming interface that provides a causally consistent view on multiple items, as an example later in this section shows.

Compared with the DM protocol, the DM-Clock protocol keeps multiple versions of each item. It also requires access to physical clocks. It assigns each item version a physical update timestamp, which imposes on causally related item versions a total order consistent with the (partial) causal order. A read-only transaction obtains its snapshot timestamp by reading the physical clock at the first partition it accesses, its *originating partition*. The DM-Clock protocol then provides a causally consistent snapshot of the data store, including the latest item versions with a physical update timestamp no greater than the transaction's snapshot timestamp.

4.4.1 Read-Only Transaction

With the DM-Clock protocol, the key-value store also provides a transactional read operation:

– ⟨vals⟩ ← GET-TX(⟨keys⟩): This operation returns the values of a set of items identified by $keys$. The returned values are causally consistent.

A read-only transaction provides a causally consistent snapshot of the data store. Assume x_r and y_r are two versions of items X and Y, respectively. If a read-only transaction reads x_r and y_r, and $x_r \rightsquigarrow y_r$, then there does not exist another version of X, x_o, such that $x_r \rightsquigarrow x_o \rightsquigarrow y_r$.

We give a concrete example to illustrate the application of read-only transactions. Assume Alice wants to share some photos with friends through an online social network such as Facebook. She first changes the permission of an album from “public" to “friends-only" and then uploads some photos to that album. When these two updates are propagated to and applied at remote replicas, causality ensures that their occurrence order is preserved: the permission update operation happens before the photo upload operation. However, it is possible that Bob, not a friend of Alice, first reads the permission of the album as “public” and then sees the photos that were uploaded after the album was changed to “friends-only”. Enclosing the album

permission check and the viewing of the photos in a causally consistent read-only transaction prevents this undesirable outcome. In a causally consistent snapshot, the photos cannot be viewed if the permission change that causally precedes their uploads is not observed.

4.4.2 Definitions

Physical Clocks. The DM-Clock protocol uses loosely synchronized physical clocks. We assume each server is equipped with a hardware clock that increases monotonically. A clock synchronization protocol, such as the Network Time Protocol (NTP) [2], keeps the clock skew under control. The clock synchronization precision does not affect the correctness of our protocol. We use $Clock_n^m$ to denote the current physical clock time at partition p_n^m.

Dependency Metadata. Compared with the DM protocol, the DM-Clock protocol introduces additional dependency metadata. Each version of an item d has a *physical update timestamp*, put_d, which is the physical clock time at the source partition of d when it is created. Different versions of an item are sorted in the item's version chain using their physical update timestamps. A client c maintains a *physical dependency time*, PDT_c, for its session. This variable stores the greatest physical update timestamp of all the states a client session depends on. Each partition p_n^m maintains a *physical version vector*, PVV_n^m, a vector of M physical timestamps. $PVV_n^m[i]$ ($0 \leqslant i \leqslant M-1, i \neq m$) indicates the physical time of p_n^i seen by p_n^m. This value comes either from replicated updates or from heartbeat messages.

4.4.3 Protocol

We now describe how the DM-Clock protocol extends the DM protocol to support read-only transactions.

GET. When a partition returns the read version d back to the client, it also includes its physical update timestamp put_d. Upon receiving the reply, the client updates its physical dependency time: $PDT_c \leftarrow max(PDT_c, put_d)$.

PUT. An update request from client c to partition p_n^m also includes PDT_c. When p_n^m receives the request, it first checks whether $PDT_c < Clock_n^m$. If not, it delays the update request until the condition holds. When p_n^m creates a new version d for the item identified by k_d, it also assigns d the physical update time: $put_d \leftarrow Clock_n^m$. It then inserts d to the version chain of item k_d using ut_d. The reply message back to the client also includes put_d. Upon receiving the reply message, the client updates its physical dependency time: $PDT_c \leftarrow max(PDT_c, put_d)$.

With the above read and update rules, our protocol provides the following property on causally related states: *For any two item versions x and y, if* $x \rightsquigarrow y$*, then* $put_x < put_y$.

Update Replication. An update replication request of d also includes put_d. When partition p_n^m receives d from p_n^s and after d's dependencies are satisfied, it inserts d into the version chain of item k_d using put_d. p_n^m then updates its physical version vector: $PVV_n^m[s] \leftarrow put_d$.

Heartbeat Broadcasting. A partition periodically broadcasts its current physical clock time to its replicas at remote data centers. It sends out heartbeat messages and updates in the physical timestamp order.

When p_n^m receives a heartbeat message with physical time pt from p_n^s, it updates its physical version vector: $PVV_n^m[s] \leftarrow pt$. We use Δ to denote the heartbeat broadcasting interval. A partition skips sending a heartbeat message to a replica if there was an outgoing update replication message to that replica within the previous Δ time. Hence, heartbeat messages are not needed when replicas exchange updates frequently enough.

GET-TX. A read-only transaction t maintains a physical *snapshot timestamp* st_t and a *readset* rs_t. Client c sends a request $\langle$GETTX $kset\rangle$ to a partition by some load balancing algorithm, where $kset$ is the set of items to read.

When t is initialized at p_o^m, the originating partition, it reads the local hardware clock to obtain its snapshot timestamp: $st_t \leftarrow Clock_o^m - \Delta$. We provide a slightly older snapshot to a transaction, by subtracting some amount of time from the latest physical clock time, to reduce the probability of a transaction being delayed and the duration of the delay. p_o^m reads the items specified by $kset$ one by one. If p_o^m does not store an item required by t, it reads the item from another local partition that stores the item.

Before t reads an item at partition p_n^m, it first waits until two conditions hold: 1) $Clock_n^m \geqslant st_t$; 2) $min(\{PVV_n^m[i] \mid 0 \leq i \leq M-1, i \neq m\}) \geqslant st_t$. t then chooses the latest version d such that $put_d \leqslant st_t$ from the version chain of the read item and adds d to its readset rs_t. After t finishes reading all the requested items, it sends a reply $\langle$GETTXREPLY $rs_t\rangle$ back to the client. The client handles each retrieved item version in rs_t one by one in the same way as for a GET operation.

By delaying a read-only transaction under the above conditions, our protocol achieves the following property: *The snapshot of a transaction includes all item versions with a physical update timestamp no greater than its snapshot timestamp, if no failure happens.*

If failure happens, a read-only transaction may be blocked. We provide solutions

to this in Section 4.5, where we discuss failure handling.

Correctness. To see why our protocol provides causally consistent read-only transactions, consider $x_r \rightsquigarrow y_r$ in the definition in Section 4.4.1 again. With the first property, if a transaction t reads x_r and y_r, then $put_{x_r} < put_{y_r} \leq st_t$. With the second property, t only reads the latest version of an item with a physical update timestamp no greater than st_t. Hence there does not exist x_o such that $put_{x_r} < put_{x_o} \leq st_t$. Therefore, it is impossible that there exists x_o such that $x_r \rightsquigarrow x_o$. Our protocol provides causally consistent read-only transactions.

Conflict Detection and Resolution. The above DM-Clock protocol cannot tell whether two updates conflict or not. We employ the same technique used by the DM protocol in Section 4.3.4 to detect conflicts. We do not need the conflict resolution part here since the DM-Clock protocol uses physical update timestamps to totally order different versions of the same item.

4.4.4 Garbage Collection

The DM-Clock protocol stores multiple versions of each item. We briefly describe how to garbage-collect old item versions to keep the storage footprint small. Partitions within the same data center periodically exchange snapshot timestamps of the oldest active transactions. If a partition does not have any active read-only transactions, it sends out the latest physical clock time. At each round of garbage collection, a partition chooses the minimum one among the received timestamps as the *safe garbage collection timestamp*. With this timestamp, a partition scans the version chain of each item it stores. It only keeps the latest item version created before the safe garbage collection timestamp (if there is one) and the versions created after the timestamp. It removes all the other versions that are not needed by active and future read-only transactions.

4.5 Failure Handling

4.5.1 DM Protocol

Client Failures. When a client fails, it stops issuing new requests to the data store. The failure of a client does not affect other clients and the data store. Recovery is not needed since a client only stores soft states for dependency tracking.

Partition Server Failures. A partition maintains a redo log on stable storage, which stores all installed update operations in the update timestamp order. A failed partition recovers by replaying the log. Checkpointing can be used to accelerate the recovery

process. The partition then synchronizes its state with other replicas at remote data centers by exchanging locally installed updates.

The current design of the DM protocol does not tolerate partition failures within a data center. However, it can be extended to tolerate failures by replicating each data partition within the same data center using standard techniques, such as primary copy [9, 64], Paxos [52] and chain replication [75].

Data Center Failures. The DM protocol tolerates the failure of an entire data center, for example due to power outage, and network partitions among data centers. If a data center fails and recovers later, it rebuilds the data store states by recovering each partition in parallel. If the network partitions and heals later, it updates the data store states by synchronizing the operation logs within each replication group in parallel. If a data center fails permanently and cannot recover, any updates originated in the failed data center, which are not propagated out, will be lost. This is inevitable due to the nature of causally consistent replication, which allows low-latency local updates without requiring coordination across data centers.

4.5.2 DM-Clock Protocol

The DM-Clock protocol uses the same failure handling techniques as the DM protocol, except that it treats read-only transactions specially.

With the DM-Clock protocol, before reading an item at a partition, a read-only transaction requires that the partition has executed all local updates and applied all remote updates with update timestamps no greater than the snapshot timestamp of the transaction. However, if a remote replica fails or the network among data centers partitions, a transaction might be delayed for a long time because it does not know whether there are any updates from a remote replica that should be included in its snapshot but have not been propagated.

Two approaches can solve this problem. If a transaction is delayed longer than a certain threshold, its originating partition re-executes it using a smaller snapshot timestamp to avoid blocking on the (presumably) failed or disconnected remote partition. With this approach, the transaction provides relatively stale item versions until the remote partition reconnects. A transaction delayed for enough long time can also switch to a two-round protocol similar to the one used in Eiger [59]. In this case, the transaction returns relatively fresh data but may need two rounds of messages to finish.

4.6 Dependency Cleaning

Dependency cleaning is a technique that further reduces the size of dependency metadata in the DM and DM-Clock protocols. This idea is general and can be applied to other causally consistency systems.

4.6.1 Intuition

Although our DM and DM-Clock protocols effectively reduce the size of dependency metadata by using dependency matrices and loosely synchronized physical clocks, for big data sets managed by a large number of servers, it is still possible that the dependency matrix of an update has many non-zero elements. For instance, a client may scan a large number of items located at many different partitions and update a single item in some statistics workloads. In this case, the dependency metadata can be many times bigger than the actual application payload, which incurs more inter-datacenter traffic for update replication. In addition, checking dependencies of such an update during replication requires a large number of messages.

The hidden assumption behind tracking dependencies at the client side is that a client does not know whether a state it accesses has been fully replicated by all replicas. To guarantee causal consistency, the client has to remember all the nearest dependency states it accesses and associate them to subsequent update operations. Hence when an update is propagated to a remote replica, the remote replica uses its dependency metadata to check whether all its dependency states are present there. This approach is *pessimistic*, because it assumes the dependency states are not replicated by all replicas. Most existing solutions for causal consistency are built on this assumption. This is a valid assumption if the network connecting the replicas fails or disconnects often, which the early works are based upon [65, 68]. However, it is not realistic for modern data center applications. Data centers of the same organization are often connected by high speed and reliable fiber links. Most of the time, network partitions among data centers only happen because of accidents, and they are rare. Therefore, we argue that one should not be pessimistic about dependency tracking for this type of applications. If a state and its dependencies are known to be fully replicated by all replicas, a client does not need to include it in the dependency metadata when reading it. With this observation, we can substantially reduce the size of the dependency metadata.

4.6.2 Protocol Extension

We describe how to extend the DM and DM-Clock protocols to support dependency cleaning. To track updates that are replicated by all replicas, we introduce a *full replication version vector* (RVV) at each partition. At partition p_n^m, RVV_n^m indicates that the first $RVV_n^m[i]$ updates of p_n^i ($0 \leqslant i \leqslant M-1$) have been fully replicated.

Update Replication. We add the following extensions to the update replication process. After p_n^m propagates an item version d to all other replicas, it requires them to send back a *replication acknowledgment* message after they apply d. Similar to propagating local updates in their update timestamp order, a partition also sends replication acknowledgments in the same order. Once p_n^m receives replication acknowledgments of d from all other replicas, it increments $RVV_n^m[m]$. p_n^m then sends a *full-replication completion* message of d to other replicas. Similarly, the full-replication completion messages are also sent in the update timestamp order. When partition p_n^i receives the full-replication completed message for d, it increments $RVV_n^i[m]$ ($0 \leqslant i \leqslant M-1$ and $i \neq m$). Since partition p_n^m increments an element of VV_n^m when applying a replicated update, but increments the corresponding element in RVV_n^m only after that update is fully replicated, $RVV_n^m \leqslant VV_n^m$ always holds.

GET and GET-TX. We now describe how RVV is used to perform dependency cleaning when the data store handles read operations. Assume a client sends a read request to partition p_n^m and an item version d is selected. If $RVV_n^m[rid_d] \geqslant ut_d$, p_n^m knows that d and all its dependency states have been fully replicated and there is no need to include d in the dependencies of the client session. p_n^m sends a reply message back to the client without including d's update timestamp. Upon receiving the reply, the client keeps its dependency matrix unchanged. This technique can also be applied to read-only transactions. Therefore, by marking an item version as fully replicated, this technique "cleans" the dependency introduced to the client that reads the item.

Normally, the duration that $RVV_n^m[rid_d] < ut_d$ holds is short. Under moderate system loads, this duration is roughly 1.5 WAN round-trip latency plus two times the write latency of stable storage , which is normally a few hundreds milliseconds. As a consequence, for a broad spectrum of applications, most read operations do not generate dependencies, which keeps the dependency metadata small.

4.6.3 Message Overhead

Dependency cleaning has a tradeoff. It reduces the size of dependency metadata and the cost of dependency checking at the expense of more network messages for sending replication acknowledgments and full-replication completions.

Assume an update depends on states from k partitions except its source partition. For a system with M replicas, without dependency cleaning, it takes $M-1$ WAN messages to propagate the update to all other replicas at remote data centers. The dependency matrix in each of the $M-1$ WAN messages has at least k non-zero rows. $(M-1)k$ LAN messages are required for checking dependencies of the propagated update at remote replicas. With dependency cleaning, it requires $3(M-1)$ WAN messages to replicate an update. For many workloads, the dependency matrix of an update contains mostly zero elements. Very few LAN messages are needed for dependency checking.

4.7 Evaluation

We evaluate the DM protocol, DM-Clock protocol, and dependency cleaning in Orbe, a multiversion key-value store that supports both partitioning and replication. In particular, we answer the following questions:

- Does Orbe scale as the number of partitions increases?
- What is the overhead of providing causal consistency compared with eventual consistency?
- How does Orbe compare with COPS [58]?
- Is dependency cleaning an effective technique for reducing the size of dependency metadata?

4.7.1 Implementation and Setup

We implement Orbe in `C++` and use Google's Protocol Buffers for message serialization. We partition the data set to a group of servers using consistent hashing [44]. We run NTP to keep physical clocks loosely synchronized. We configure NTP to gradually catch up or fall back to a target during synchronization. Hence, physical clocks always move forward, a requirement for correctness in Orbe.

As part of the application tier, servers that run Orbe clients are in the same data center with Orbe partition servers. A client chooses a partition in its local data center as its originating partition by a load balancing scheme. The client then issues all its operations to its originating partition. If the originating partition does not store an item required by a client request, it executes the operation at the local partition that manages the required item.

Orbe's underlying key-value store keeps all key-value pairs in main memory. A key points to a linked list that contains different versions of the same item. The operation log resides on disk. The system performs group commit to write multiple updates

in one disk write. A PUT operation inserts a new version to the version chain of the updated item and adds a record to the operation log. During replication, replicas of the same partition exchange their operation logs. Each replica replays the log from other replicas and applies the updates one by one after dependency checking.

We run the DM-Clock protocol in all experiments, even where the DM protocol would suffice, because it is a superset of the DM protocol. We set the heartbeat broadcasting interval Δ to 10ms. By default, dependency cleaning is disabled. We enable it in one experiment, where we mention its use explicitly (see Section 4.7.6).

We run experiments on a local cluster where all servers are connected by a single GigE switch. All servers in the cluster are Dell PowerEdge SC1425 running Linux 3.2.0. Each server has two Intel Xeon processors, 4GB of DDR2 memory, one 7200rpm 160GB SATA disk, and one GigE network port. The round-trip network latency in our local cluster is between 120 to 180 microseconds. We enable the hardware cache of the disk. The latency of writing a small amount of data (64B) to the disk is around 450 microsecond. We partition the local cluster into multiple logical "data centers" as necessary. We introduce an additional 120 milliseconds network latency for messages among replicas of the same partition located at different logical data centers.

4.7.2 Microbenchmarks

We first evaluate the basic performance characteristics of Orbe through microbenchmarks. We replicate the data set in two data centers. At each data center, the data set is partitioned on eight servers. Each partition loads one million data items during initialization. For each preloaded item, the size of a key is eight bytes and the value is ten bytes.

In the first experiment, we examine the capability of a single partition server. We launch enough clients to saturate the server. A GET operation reads a randomly selected item from its originating partition. The PUT operation also operates on the originating partition by updating a random item with different sizes of update values. For comparison, we also introduce an Echo operation, which simply returns the operation argument to the clients.

As shown in Table 4.2, a partition server can process Echo operations at about 70K ops/s, GET operations at about 60K ops/s, and PUT operations at about 30K ops/s. The throughput of Echo indicates the message processing capability of our hardware. As the update value size increases in PUT, the throughput drops slightly due to the increased cost of memory copies. In all cases, CPU is the bottleneck.

In the second experiment, we measure operation latencies. For this experiment,

Operation	Echo	GET-10B	PUT-1B	PUT-16B	PUT-128B
Throughput (K op/s)	71.3	61.4	36.8	36.4	30.2

Table 4.2 – Maximum throughput of client operations on a single partition server without replication.

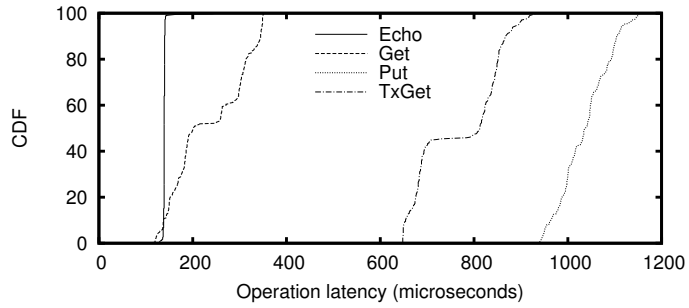

Figure 4.3 – Latency distribution of client operations.

GET and PUT choose items located at the originating partition with a probability of 50% and at other local partitions with the other 50%. A GET-TX operation reads six items in total. One is from its originating partition while the other five are from other local partitions.

Figure 4.3 shows the latency distribution of the four operations. The Echo operation shows the baseline as it only takes one round-trip latency to finish. Each GET and PUT requires either one or two rounds of messages within the same data center (depending on the location of the requested item), which results in two clear groups of latencies. The latency of executing a client operation is low, because Orbe does not require a partition server to coordinate with its replicas at other data centers to process GET and PUT operations. None of the (microsecond-scale) client operations depend on replication operations that incur the 120ms latency between data centers.

4.7.3 Scalability

We now examine the scalability of Orbe with an increasing number of partitions. We set up two data centers with two to eight partitions at each.

We first use three workloads which are a configurable mix of PUTs and GETs. Items are selected from the originating partition with a probability of 50% and from other local partitions for the other 50%. PUT operations update items with ten bytes values. For the workload of read-only transactions, each GET-TX reads one item from its originating partition and five from other local partitions. Figure 4.4 shows the through-

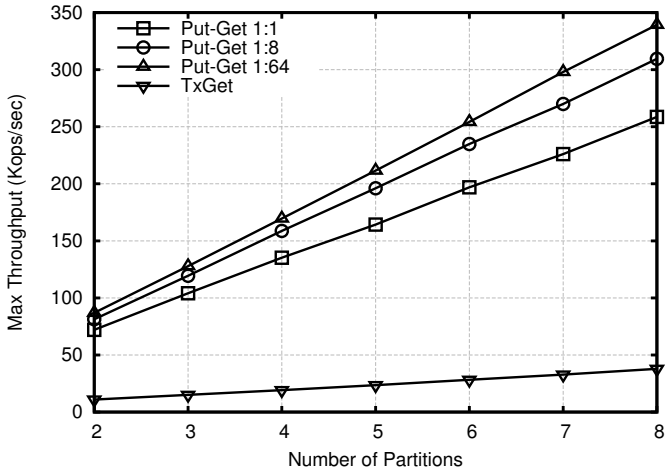

Figure 4.4 – Maximum throughput of varied workloads with two to eight partitions. The legend gives the put:get ratio.

put of Orbe as the number of partitions increases. Regardless of the put:get ratio, Orbe scales out with an increasing number of partitions. Because Orbe propagates updates across partitions in parallel, it is able to utilize more servers to provide higher throughput.

4.7.4 Comparison with Eventual Consistency

To show the overhead of providing causal consistency in our protocols, we compare Orbe with an eventually consistent key-value store, which is implemented in Orbe's codebase. We set up two data centers of three partitions each. A client accesses items randomly selected from the three local partitions with different put:get ratios. PUT updates an item with a value of 60 bytes.

Figure 4.5 shows the throughput of Orbe and the eventually consistent key-value store. For an almost read-only workload, they have similar throughputs. For an almost update-only workload, Orbe's throughput is about 24% lower. The minor degradation in throughput is a reasonable price to pay for the much improved semantics over eventual consistency.

The major overhead of implementing causal consistency comes from 1) network messages for dependency checking and 2) processing, storing and transmitting the dependency metadata. Figure 4.6 shows this overhead with two curves. The first is the average number of dependency checking messages per replicated update. The second is the percentage of the dependency metadata in the update replication traffic in Orbe. When the workload is almost update-only, the metadata percentage is small and so

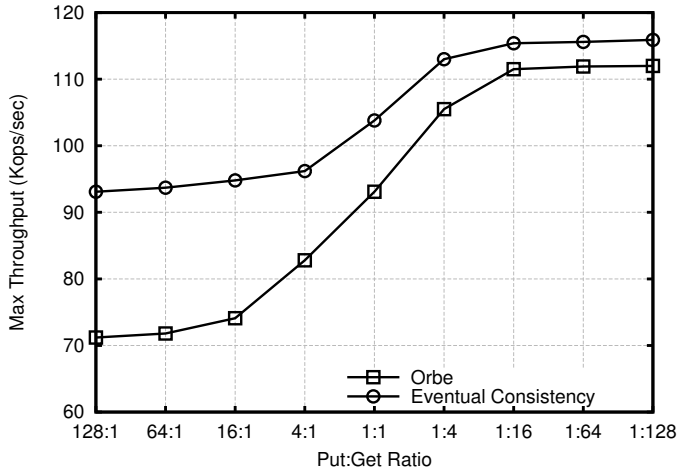

Figure 4.5 – Maximum throughput of workloads with varied put:get ratios for both Orbe (causal consistency) and eventual consistency.

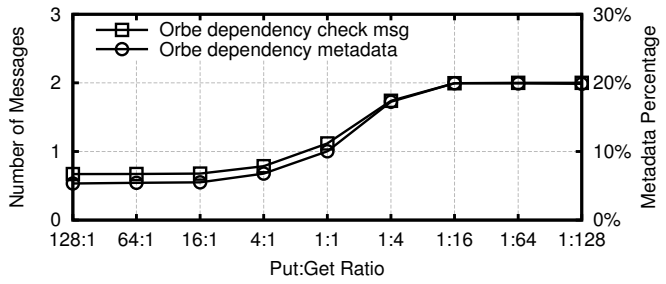

Figure 4.6 – Average number of dependency checking messages per replicated update and percentage of dependency metadata in the update replication traffic with varied put:get ratios in Orbe.

is the number of dependency checking messages per replicated update. When the workload becomes read-heavy, the numbers go up, but level off after GETs dominate the workload.

The contents of dependency matrices explain the numbers in Figure 4.6. As Orbe tracks only the nearest dependencies, an update depends only on the previous update and the reads since the previous update in the same client session. With a high put:get ratio, the dependency matrix contains only a few non-zero elements. With a low put:get ratio, reads generate a large number of dependencies, but the total number of elements in a dependency matrix is bounded by the number of partition servers in the data store.

4.7.5 Comparison with COPS

We compare Orbe with COPS [58], which also provides causal consistency for both partitioned and replicated data stores. We implement COPS in Orbe's codebase. For an apples-to-apples comparison, we enable read-only transaction support in both Orbe and COPS (which is called COPS-GT in prior work [58]).

COPS explicitly tracks each item version read and updated at the client side. It associates a set of dependency item versions with each update. A dependency matrix in Orbe plays the same role as a set of dependency item versions in COPS, but most of the time it takes less space using sparse encoding.

Although COPS relies on a number of techniques to reduce the size of dependency metadata, it can still become considerable since COPS has to track the *complete* dependencies to support read-only transactions while Orbe only tracks the nearest dependencies. In addition, the execution time of the two-round transactional reading protocol in COPS limits the frequency of garbage-collecting the dependency metadata. During the fixed interval between two successive garbage collections, the more operations a client issues, the more dependency states it creates. Hence, the size of dependency metadata is highly related to the inter-operation delays at each client. COPS sets the garbage collection interval to six seconds [58]. We use the same value in our COPS implementation.

We set up two data centers of three partitions each. A client accesses data items randomly selected from the three partitions with a configurable put:get ratio. Figure 4.7 illustrates the throughput of Orbe and COPS with different client inter-operation delays. Figure 4.8 shows the average number of states on which an update depends. For COPS, this is the number of dependency item versions. For Orbe, this is the number of non-zero elements in the dependency matrix. Orbe provides consistently higher throughput than COPS as it tracks fewer dependency states and spends fewer CPU cycles on message serialization and transmission. Figure 4.8 suggests that Orbe and COPS should have similar throughput when the inter-operation delays are longer than one second.

By tracking fewer dependency states and efficiently encoding the dependency matrix, Orbe also reduces the inter-datacenter network traffic for update replication among replicas of the same partition. Figure 4.9 compares the aggregated replication traffic (transmission only) between Orbe and COPS.

Tracking fewer states at the client side reduces the client's memory footprint, but also consumes fewer CPU cycles as fewer temporary objects are created and destroyed for tracking dependencies. Figure 4.10 shows the CPU utilization of a server that

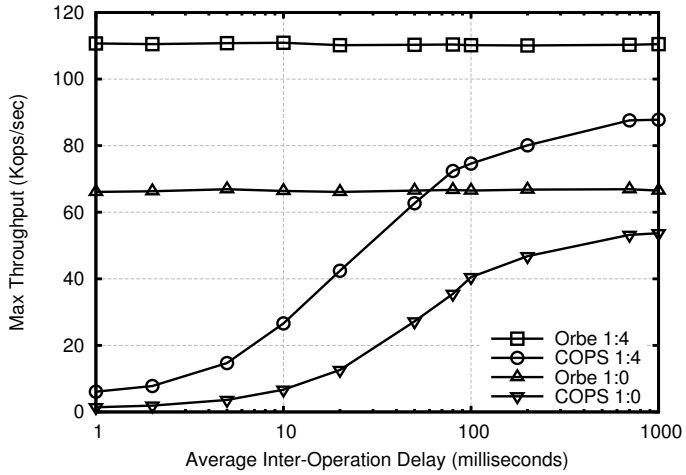

Figure 4.7 – Maximum throughput of operations with varied inter-operation delays for both Orbe and COPS. The legend gives the put:get ratio.

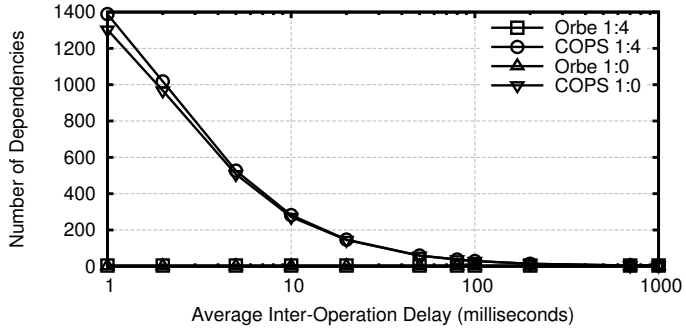

Figure 4.8 – Average number of dependencies for each PUT operation with varied inter-operation delays for both Orbe and COPS. The legend gives the put:get ratio.

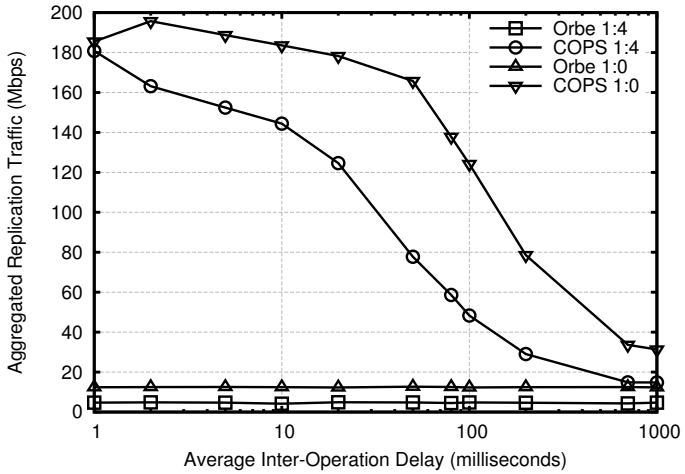

Figure 4.9 – Aggregated replication transmission traffic across data centers with varied inter-operation delays for both Orbe and COPS. The legend gives the put:get ratio.

runs a group of clients for Orbe and COPS, separately. For this measurement, we run all clients at a single powerful server to saturate the data store and record the CPU utilization of the server. Orbe is more efficient. It uses fewer CPU cycles per operation as it manages fewer states.

4.7.6 Dependency Cleaning

Dependency cleaning removes the necessity for dependency tracking when a client reads a fully replicated item version. However, it increases the cost of update replication as it requires additional inter-datacenter messages to mark an item version as fully replicated.

In this experiment, we show the benefits of dependency cleaning for workloads that read from a large number of partitions and update only a few. We set up two data centers and vary the number of partitions from one to eight at each data center. A client reads a randomly selected item from each of the local partitions and updates one random item at its originating partition.

Figure 4.11 shows the maximum throughput of Orbe with and without dependency cleaning enabled. When the system has only one partition, all reads and updates go to that partition. Dependency cleaning does not help as no network message is required for dependency checking. In this case, the throughput of Orbe with dependency cleaning enabled is slightly lower, because it requires more messages for update replication. However, the throughput drop is small, because we let update replication messages piggyback replication acknowledgement and full-replication completion

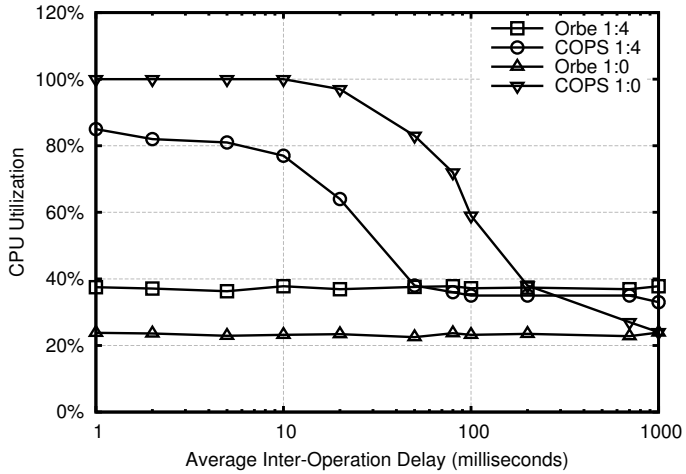

Figure 4.10 – CPU utilization of a server that runs a group of clients with varied inter-operation delays for both Orbe and COPS. The legend gives the put:get ratio.

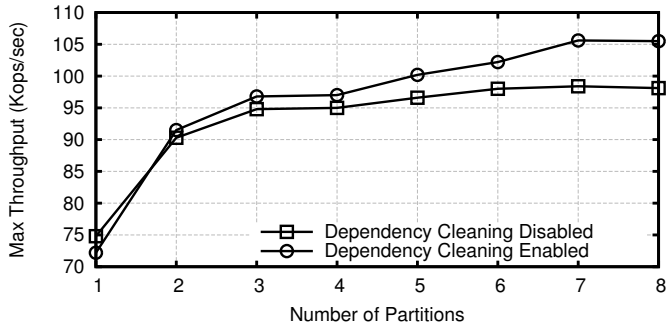

Figure 4.11 – Maximum throughput of operations with and without dependency cleaning enabled.

messages and batch these messages whenever possible.

As we increase the number of partitions, the throughput of Orbe with dependency cleaning enabled is higher. The throughput gap increases as the system has more partitions. With dependency cleaning, an update does not depend on states from other partitions most of the time, although it reads states from those partitions. As a result, dependency checking on replicated updates does not incur network messages at the remote data center. By removing this part of the overhead, dependency cleaning helps the overall throughput.

Dependency cleaning also reduces the size of dependency metadata for an update. Figure 4.12 shows the aggregated replication traffic from all partitions. The traffic decreases as the number of partitions increases because the put:get ratio decreases. Figure 4.13 shows the average percentage of metadata in the update replication traffic.

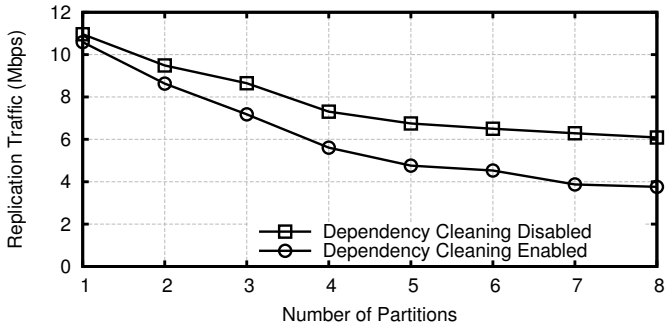

Figure 4.12 – Aggregated replication transmission traffic with and without dependency cleaning enabled.

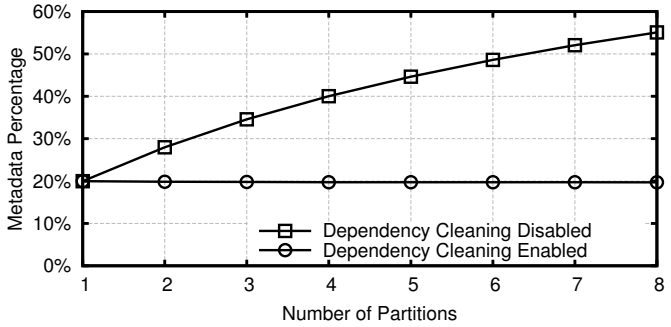

Figure 4.13 – Average percentage of dependency metadata in update replication traffic with and without dependency cleaning enabled.

4.8 Summary

In this chapter, we introduce two scalable protocols that efficiently provide causal consistency for partitioned and replicated data stores. The DM protocol extends version vectors to two-dimensional dependency matrices and relies on the transitivity of causality to keep dependency metadata small and bounded. The DM-Clock protocol relies on loosely synchronized physical clocks to provide causally consistent read-only transactions. We implement the two protocols in a distributed key-value store. We show that they incur relatively small overhead for tracking causal dependencies and outperform a prior approach based on explicit dependency tracking. The work described in this chapter appears in [27].

5 Related Work

This thesis introduces protocols that efficiently implement three consistency models by using loosely synchronized physical clocks to order events. In this chapter we present the related work, which can be generally divided into two categories: (1) using different techniques to implement the same or similar consistency models and (2) relying on physical clocks to implement other consistency models.

5.1 Distributed Transactions

SI in distributed systems. A number of large-scale distributed data stores use SI to support distributed transactions. These systems all rely on a centralized service for timestamp management.

Percolator [67] adds distributed SI transactions to Bigtable [20] to support incremental building of web search index. It assigns snapshot and commit timestamps to transactions using a centralized timestamp oracle. It adds additional columns to each row to store transaction metadata such as update timestamps and locks. Percolator is not designed for OLTP applications and a transaction can potentially be blocked for tens of seconds due to its lazy approach to clean up locks left by failed machines.

Zhang and Sterck [80] implement SI on top of HBase [1] in a system that stores all the transaction metadata in a number of global tables. The system manages timestamps by using a centralized shared table. ReTSO [42] implements lock-free SI for HBase using a centralized service to detect write-write conflicts and to assign timestamps. Similarly, an implementation of write snapshot isolation (WSI) using a centralized transaction certifier is given in [78].

The centralized timestamp services in these systems affect their throughput, latency, and availability. In contrast, Clock-SI uses a group of loosely synchronized physical clocks rather than a centralized authority for timestamp management, with the attendant benefits shown in this thesis.

Relaxing SI in distributed systems. Saeida Ardekani et al. [10] study the scalability of SI. They decouple SI to a few simple properties and show that a system cannot have both SI and genuine partial replication (GPR) at the same time. In other words, SI is not scalable in a partitioned and replicated system when transactions are ordered by logical clocks. Hence, prior work proposes relaxing the total order property of SI to achieve better performance in partitioned and replicated systems.

Walter [73] is a transactional geo-replicated key-value store that uses parallel snapshot isolation (PSI), which orders transactions only within a site and tracks causally dependent transactions using a vector clock at each site, leaving independent transactions unordered among sites. PSI relaxes SI by not enforcing a total commit order among all transactions. With PSI, data items are partitioned into different sites. Within a site, transaction commit is totally ordered. Across sites, PSI enforces only causal ordering which allows the system to replicate transactions asynchronously across sites.

Non-monotonic snapshot isolation (NMSI) [69, 70] provides non-monotonic snapshots. NMSI does not explicitly order transactions but uses dependency vectors to track causally dependent transactions. There is no global time that totally orders all transaction commits. This reduces the message complexity of SI in a partitioned system. Compared with PSI, NMSI supports genuine partial replication and reduces abort rates by reading fresher data.

The relaxations of SI may fail to provide application developers with the familiar isolation levels and requires extra effort to guarantee that transactions read consistent and monotonic snapshots as in SI. In contrast, Clock-SI provides a complete implementation of SI, including a total order on transaction commits and a guarantee that transactions read consistent and monotonic snapshots across partitions.

Relaxing freshness. Some systems relax freshness to improve performance at the cost of serving stale data. Relaxed currency models [14, 37, 38] allow each transaction to have a freshness constraint. Continuous consistency [79] bounds staleness using a real-time vector. These systems do not use physical clocks to assign transaction snapshot and commit timestamps. Clock-SI provides each transaction with either the latest snapshot or a slightly older snapshot (tunable per transaction) to reduce the delay probability and duration of transactions. In both cases, all the properties of SI are maintained.

Generalized snapshot isolation (GSI) [31] generalizes SI to replicated databases. It uses older snapshots to avoid the delay of waiting for committed updates to be propagated from the certifier to the replicas. In contrast, Clock-SI targets concur-

rency control for partitioned data stores using physical time. It allows assigning older snapshots to reduce the delay probability and duration of transactions.

5.2 State Machine Replication

In addition to Multi-Paxos [52, 53] and Mencius [61], which we compare with Clock-RSM in details in Chapter 3, Clock-RSM is also related to the following work.

Fast Paxos [55] allows clients to send commands directly to all replicas to reduce commit latency. In good runs, it requires two message delays to commit a command. However, under collisions due to concurrent proposals, Fast Paxos requires at least two additional messages for collision recovery. Collisions are frequent in a geo-replicated environment with balanced workloads, and thus Fast Paxos results in significantly higher latency than Clock-RSM.

Some protocols relax the total order property of state machine replication. Generalized Paxos [54] and Generic Broadcast [66] commits commands that do not interfere out of order in one round trip. It requires a stable leader to order interfering commands, which takes at least two additional round trips. Egalitarian Paxos [63], also called EPaxos, does not require a designated leader. Every replica in EPaxos can serve client requests and submit commands. EPaxos also commits non-interfering commands out of order in two message delays, which is one round-trip latency to (at least) a majority of replicas. The slow path, which resolves conflicts, requires one additional round trip. EPaxos provides linearizability. However, local reads in EPaxos may see updates in different orders at different replicas. In contrast, Clock-RSM provides linearizability and maintains an explicit total order over updates. Database replication, one of the most popular applications of state machine replication, often requires total order replication to maintain strong transaction isolation levels, such as serializability [15] and snapshot isolation [13], as in a single-copy database [45].

MDCC [46] uses Generalized Paxos to build a replicated partitioned key-value store across data centers. MDCC reduces replication latency by running one instance of Generalized Paxos per key. An update to a key commits in one round trip using the fast path, under the assumption that conflicting updates on the same key are rare. However, MDCC provides “read committed” guarantee to transactions, a weaker isolation level than both serializability and snapshot isolation. In contrast, Clock-RSM can be used for total order replication across all keys while providing low latency and strong transaction isolation.

Using physical clocks for state machine replication is discussed in [50] and [71]. However, neither of them provides a complete solution. Clock-RSM is a clearly speci-

fied protocol with reconfiguration for failure handling. It relies on physical clocks to reduce replication latency across data centers.

An atomic broadcast algorithm using physical clocks is introduced in [81]. This algorithm relies on ordinary broadcast and generic broadcast and delivers a message with two message delays in good runs. Similar to Clock-RSM, each replica coordinates its own commands and commands are totally ordered by physical time intervals. In contrast, Clock-RSM is a simpler and more efficient state machine replication protocol that includes recovery and reconfiguration and targets realistic geo-replication environments.

S-Paxos [16] optimizes Multi-Paxos for throughput. It offloads the command distribution work from the leader to all replicas. The leader only handles command ordering. With aggressive message batching at the leader, S-Paxos alleviates the performance bottleneck at the leader. However, the commit latency remains the same because the ordering metadata of each command still goes through the same message steps as in Multi-Paxos.

5.3 Causal Consistency

There have been many causally consistent systems in the literature, such as lazy replication [48], Bayou [68], and WinFS [60]. They use various techniques derived from the causal memory algorithm [7]. However, these systems target only full replication. None of them considers scalable causal consistency for partitioned and replicated data stores, to which Orbe provides a solution.

COPS [58] identifies the problem of causal consistency for both partitioned and replicated data stores and gives a solution. It tracks every accessed state as dependency metadata at the client side. To support causally consistent read-only transactions, a client has to track the complete dependencies explicitly. Although COPS garbage-collects the dependency metadata periodically, the metadata size may still be large under many workloads and can affect performance. In comparison, Orbe relies on dependency matrices to track the nearest dependencies and keeps the dependency metadata small and bounded. Orbe provides causally consistent read-only transactions using loosely synchronized clocks. Orbe requires one round of messages to execute a read-only transaction in the failure-free mode while COPS requires maximum two rounds.

Eiger [59] is recent follow-up work on COPS and provides causal consistency for a distributed column store. It proposes a new protocol for read-only transactions using Lamport clock [50]. Although Eiger also needs maximum two rounds of messages

to execute a read-only transaction, a client tracks only the nearest dependencies. In addition, Eiger provides causally consistent update-only transactions. Eiger still tracks every accessed state at the client side. All its improvements to COPS are different from the techniques Orbe uses.

ChainReaction [8] implements causal consistency on top of chain replication [75]. ChainReaction also tracks every accessed state at the client side. To solve the problem of large dependency metadata when providing read-only transactions, it uses a global sequencer service at each data center to totally order update operations and read-only transactions. However, the sequencer service increases the latency of all update operations by one round-trip network latency within the data center and is a potential performance bottleneck. In comparison, Orbe does not have any centralized component. It provides causal consistency for update-anywhere replication and relies on loosely synchronized physical clocks to implement read-only transactions.

Bolt-on causal consistency [11] provides causal consistency to existing eventually consistent data stores. It inserts a shim-layer between the data store and the application layer to insure the safety properties of causal consistency. It relies on the application to maintain explicit causality relationships. Tracking causality based on application semantics is precise but requires the application developers to specify causal relationships among application operations. In contrast, Orbe provides causal consistency directly in the data store, without requiring coordination from the application.

5.4 Physical Clocks in Distributed Systems

Liskov provides a survey of the use of loosely synchronized clocks in distributed systems long time ago [56]. Since then, physical clocks have been used in a few new research and commercial systems for data management. We briefly describe them below.

The Thor project explores the use of loosely synchronized clocks for distributed concurrency control [5, 6, 57]. Transactions in Thor execute on cached objects at the client side. They commit/abort at the server side after a validation phase. AOCC [5] assigns to committed transactions unique commit timestamps from physical clocks. Transactions running under AOCC may, however, read inconsistent states of the database. Therefore, read-only transactions need to be validated on commit and therefore may need to abort, an undesirable situation which does not happen in Clock-SI.

An extension to AOCC lets running transactions read *lazily consistent states* [6]

according to a dependency relation providing *lazy consistency* (LC). LC is weaker than SI [4]. Some read histories allowed under LC are forbidden by SI. For example, assume two items x_0 and y_0. Transaction T_1 writes x_1 and commits. Then transaction T_2 writes y_1 and commits. Next, transaction T_3 reads the two items. Under LC, T_3 is allowed to read x_0 and y_1, which is not serializable and also not allowed under SI. Therefore, even with AOCC+LC read-only transactions need to be validated and may have to abort.

Both AOCC and AOCC+LC do not provide consistent snapshots for running transactions. In comparison, transactions in Clock-SI always receive consistent snapshots and read-only transactions do not abort.

Granola [23] runs single-partition and independent multi-partition transactions serially at each partition to remove the cost of concurrency control. Such a transaction obtains a timestamp before execution, and transactions execute serially in timestamp order. Coordinated multi-partition transactions use traditional concurrency control and commit protocols. To increase concurrency on multicore servers, Granola partitions the database among CPU cores. This increases the cost of transaction execution, because transactions that access multiple partitions on the same node need distributed coordination. In contrast, Clock-SI runs all transactions concurrently and does not require partitioning the data set among CPU cores.

Spanner [22] is a geographically replicated and partitioned data store built at Google. It provides linearizability or external consistency based on synchronized clocks with bounded uncertainty, called TrueTime, requiring access to GPS and atomic clocks. Spanner executes update transactions at the leader replica using conventional two-phase locking to provide serializability. In addition, transactions can be annotated as read-only and executed according to SI. In comparison, Clock-SI relies solely on physical time to implement SI. Spanner relies on conventional Multi-Paxos to replicate each partition. Replica leaders order transactions with physical timestamps. In comparison, Clock-RSM is a new state machine replication protocol and uses physical clocks to improve latency. Spanner's provision of external consistency requires high-precision clocks, and its correctness depends on clock synchrony. In contrast, both Clock-SI and Clock-RSM use conventional physical clocks available on today's commodity servers, and its correctness does not depend on clock synchrony.

VoltDB [3] uses physical clocks to pre-order transactions and executes update transactions serially on each replica of the same partition. Cassandra [49] implements the last-writer-wins rule by using physical clocks to order concurrent update operations on the same items.

6 Conclusion

In this thesis, we introduce protocols that implement three widely used consistency models efficiently and correctly by using loosely synchronized physical clocks. We demonstrate the feasibility and benefits of using loosely synchronized physical clocks to order events in a distributed system. Because physical clocks advance automatically at roughly the same speed and can be well synchronized by standard protocols, they can be used to order events across a large number of servers at very low cost. Although physical clocks are less flexible than logical clocks and have synchronization skews, we show that they can be used to implement the required consistency properties correctly.

We first introduce Clock-SI for transaction management in partitioned data stores. Clock-SI is fully distributed implementation of SI. It uses a group of distributed physical clocks at each partition for snapshot and commit timestamp assignment. Clock-SI improves over existing solutions that use a centralized timestamp service, by eliminating a potential performance bottleneck and a single point of failure. Moreover, it also avoids the round-trip messages between the partitions and the timestamp service, resulting in lower transaction commit latency.

We then introduce Clock-RSM for strongly consistent geo-replication. Clock-RSM is a new state machine replication protocol that provides linearizability. It uses the physical clocks at each replica to totally order state machine commands. Compared with existing solutions that rely on the leader replica to order commands, Clock-RSM eliminates the messages to the leader for command ordering at the follower replicas. It hence reduces the latency of consistent geo-replication across multiple data centers.

We also describe Orbe, which provides a scalable and efficient implementation of causal consistency for both partitioned and replicated data stores. Compared with existing solutions that explicitly track every dependency, Orbe extends version vectors to two-dimensional dependency matrices to keep the dependency metadata small

and bounded. It also relies on physical clocks to assign a total order over all operations and guarantees that the order is consistent with causality. With these two techniques, Orbe efficiently provides causally consistent read-only transactions in a distributed key-value that also supports basic read and write operations.

Bibliography

[1] Apache hbase. https://hbase.apache.org/, 2014.

[2] The network time protocol. http://www.ntp.org, 2014.

[3] VoltDB. http://voltdb.com, 2014.

[4] Atul Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, Massachusetts Institute of Technology, 1999.

[5] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, 1995.

[6] Atul Adya and Barbara Liskov. Lazy consistency using loosely synchronized clocks. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, 1997.

[7] Mustaque Ahamad, Gil Neiger, James E Burns, Prince Kohli, and Phillip W Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[8] Sérgio Almeida, João Leitão, and Luís Rodrigues. ChainReaction: a causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013.

[9] Peter A Alsberg and John D Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 562–570, 1976.

[10] Masoud Saeida Ardekani, Pierre Sutra, Marc Shapiro, and Nuno Preguiça. On the scalability of snapshot isolation. In *Euro-Par 2013 Parallel Processing*, pages 369–381. 2013.

[11] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 761–772, 2013.

[12] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[13] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, 1995.

[14] Philip A. Bernstein, Alan Fekete, Hongfei Guo, Raghu Ramakrishnan, and Pradeep Tamma. Relaxed-currency serializability for middle-tier caching and replication. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, 2006.

[15] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[16] Martin Biely, Zarko Milosevic, Nuno Santos, and André Schiper. S-Paxos: Offloading the leader for high throughput state machine replication. In *SRDS*, 2012.

[17] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the 19th Annual ACM symposium on Principles of Distributed Computing*, 2000.

[18] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006.

[19] Brad Calder, Ju Wang, Aaron Ogus, et al. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, 2011.

[20] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. BigTable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[21] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*.

[22] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, et al. Spanner: Google's globally-distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[23] James Cowling and Barbara Liskov. Granola: Low-overhead distributed transaction coordination. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*.

[24] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with ordering guarantees. In *Proceedings of the 20th International Conference on Data Engineering*, 2004.

[25] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with snapshot isolation. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, 2006.

[26] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220, 2007.

[27] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. Orbe: scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the 4th annual Symposium on Cloud Computing*, 2013.

[28] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Proceedings of the 32nd International Symposium on Reliable Distributed Systems*, 2013.

[29] Jiaqing Du, Daniele Sciascia, Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. Clock-RSM: Low-latency inter-datacenter state machine replication using loosely synchronized physical clocks. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[30] Sameh Elnikety, Steven Dropsho, and Fernando Pedone. Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In *Proceedings of the 1st ACM European Conference on Computer Systems*, 2006.

[31] Sameh Elnikety, Fernando Pedone, and Willy Zwaenepoel. Database replication using generalized snapshot isolation. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, 2005.

[32] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained network time synchronization using reference broadcasts. *ACM SIGOPS Operating Systems Review*, 36, 2002.

[33] Daniel Gómez Ferro, Flavio Junqueira, Ivan Kelly, Benjamin Reed, and Maysam Yabandeh. Omid: Lock-free transactional support for distributed data stores. In *ICDE*, pages 676–687, 2014.

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

[35] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, 1996.

[36] Jim Gray and Andreas Reuter. *Transaction processing*. Kaufmann, 1993.

[37] Hongfei Guo, Per-Ake Larson, and Raghu Ramakrishnan. Caching with good enough currency, consistency and completeness. In *Proceedings of the 31st International Conference on Very Large Data Bases*, 2005.

[38] Hongfei Guo, Per-Ake Larson, Raghu Ramakrishnan, and Jonathan Goldstein. Relaxed currency and consistency: how to say good enough in SQL. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, 2004.

[39] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[40] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. 2010.

[41] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 3–14, 2013.

[42] Flavio Junqueira, Benjamin Reed, and Maysam Yabandeh. Lock-free transactional support for large-scale storage systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops*, 2011.

[43] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks*, 2011.

[44] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th Annual ACM symposium on Theory of computing*, 1997.

[45] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proceedings of the 26th International Conference on Very Large Data Bases*, 2000.

[46] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. Mdcc: Multi-data center consistency. In *EuroSys*, 2013.

[47] Konstantinos Krikellas, Sameh Elnikety, Zografoula Vagena, and Orion Hodson. Strongly consistent replication for a bargain. In *Proceedings of the 2010 IEEE 26th International Conference on Data Engineering*, 2010.

[48] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, 1992.

[49] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[50] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[51] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 100(9):690–691, 1979.

[52] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[53] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[54] Leslie Lamport. Generalized consensus and paxos. 2004.

[55] Leslie Lamport. Fast paxos. *Distributed Computing*, 2006.

[56] Barbara Liskov. Practical uses of synchronized clocks in distributed systems. *Distributed Computing*, 6(4):211–219, 1993.

[57] Barbara Liskov, Miguel Castro, Liuba Shrira, and Atul Adya. Providing persistent objects in distributed systems. *ECOOP'99 Object-Oriented Programming*, pages 667–667, 1999.

[58] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, 2011.

[59] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, 2013.

[60] Dahlia Malkhi and Doug Terry. Concise version vectors in WinFS. *Distributed Computing*, 20(3):339–353, 2005.

[61] Yanhua Mao and Flavio P Junqueira. Mencius: Building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008.

[62] Friedemann Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1(23):215–226, 1989.

[63] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013.

[64] Brian M. Oki and Barbara Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, 1988.

[65] D Stott Parker Jr, Gerald J Popek, Gerard Rudisin, Allen Stoughton, Bruce J Walker, Evelyn Walton, Johanna M Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, (3):240–247, 1983.

[66] Fernando Pedone and André Schiper. Generic broadcast. In *Distributed Computing*. 1999.

[67] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[68] Karin Petersen, Mike J Spreitzer, Douglas B Terry, Marvin M Theimer, and Alan J Demers. Flexible update propagation for weakly consistent replication. In *ACM SIGOPS Operating Systems Review*, volume 31, pages 288–301, 1997.

[69] Masoud Saeida Ardekani, Pierre Sutra, Nuno Preguiça, and Marc Shapiro. Non-Monotonic Snapshot Isolation. Technical Report RR-7805, Jun. 2013.

[70] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *SRDS*, 2013.

[71] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[72] Adam Silberstein, Jeff Terrace, Brian F. Cooper, and Raghu Ramakrishnan. Feeding frenzy: Selectively materializing users' event feeds. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 2010.

[73] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, 2011.

[74] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[75] Robbert van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design*, pages 91–104, 2004.

[76] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[77] Shuqing Wu and Bettina Kemme. Postgres-R (SI): Combining replica control with concurrency control based on snapshot isolation. In *Proceedings of the 2005 IEEE 21th International Conference on Data Engineering*, 2005.

[78] Maysam Yabandeh and Daniel Gómez Ferro. A critique of snapshot isolation. In *Proceedings of the 7th ACM European Conference on Computer Systems*, 2012.

[79] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the 4th Conference on Symposium on Operating System Design and Implementation*, 2000.

[80] Chen Zhang and Hans De Sterck. Supporting multi-row distributed transactions with global snapshot isolation using bare-bones hbase. In *Proceedings of the 11th IEEE/ACM International Conference on Grid Computing*, 2010.

[81] Piotr Zieliński. Low-latency atomic broadcast in the presence of contention. *Distributed Computing*, 20(6):435–450, 2008.

Jiaqing Du

CONTACT

EPFL IC IIF LABOS
BC 122 (Bâtiment BC), Station 14
1015 Lausanne, Switzerland

Office : +41 21 6936 673
Email : jiaqing.du@epfl.ch

INTERESTS

Distributed systems, operating systems, database systems

EDUCATION

Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Ph.D. candidate in Computer Science, September 2008 - present

Korea Advanced Institute of Science and Technology (KAIST), South Korea

M.Sc. in Computer Science, August 2008

Xidian University, Xi'an, China

B.Eng. in Telecommunications Engineering, July 2005

WORK EXPERIENCE

Microsoft Research, Research Intern, September 2011 - December 2011

Microsoft Research, Research Intern, June 2010 - September 2010

PUBLICATIONS

Clock-RSM: **Low-Latency Inter-Datacenter State Machine Replication Using Loosely Synchronized Physical Clocks**. Jiaqing Du, Daniele Sciascia, Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (DSN 2014)

Closing The Performance Gap between Causal Consistency and Eventual Consistency. Jiaqing Du, Calin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. *1st Workshop on Principles and Practice of Eventual Consistency* (PaPEC 2014)

Orbe: **Scalable Causal Consistency Using Dependency Matrices and Physical Clocks**. Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. *The ACM Symposium on Cloud Computing 2013* (SoCC 2013)

Clock-SI: **Snapshot Isolation for Partitioned Data Stores Using Loosely Synchronized Clocks**. Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. *32nd International Symposium on Reliable Distributed Systems* (SRDS 2013)

Performance Profiling of Virtual Machines. Jiaqing Du, Nipun Sehrawat, and Willy Zwaenepoel. *7th International Conference on Virtual Execution Environments* (VEE 2011)

Performance Profiling in a Virtualized Environment. Jiaqing Du, Nipun Sehrawat, and Willy Zwaenepoel. *2nd USENIX Workshop on Hot Topics in Cloud Computing* (HotCloud 2010)