Achieving Strong Consistency and Low-Cost Fault Tolerance using Logical Clocks in Distributed Transactional Systems

Achieving Strong Consistency and Low-Cost Fault Tolerance using Logical Clocks in Distributed Transactional Systems

by

Masoomeh Javidi Kishi

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Doctor of Philosophy

in

Computer Science

Lehigh University

August 2020

Approved and recommended for acceptance as a dissertation in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Date

Dr. Roberto Palmieri, Dissertation Advisor

Committee Members:

Dr. Roberto Palmieri, Committee Chair

Dr. Hank Korth

Dr. Michael Spear

Dr. Sebstiano Peluso

Acknowledgements

The completion of this work could not have been possible without the help and assistance of many amazing people in my life. First and foremost, is the warm of my heart, Mohammad Mostafavi. This work is dedicated to him.

My deepest and eternal appreciation goes to my parents Fatemeh Lavaei and Mohammadbagher Javidi Kishi and my brother Alireza Javidi Kishi who support me by sending their love all the time.

I owe a huge debt of gratitude to my advisor, Dr. Roberto Palmieri, who has inspired me to achieve a level higher than I thought possible. His guidance and encouragement was invaluable and without it I most surely would have been lost at sea.

I would like to express my deep acknowledgments to the members of my dissertation committee Dr. Hank Korth, Dr. Michael Spear, and Dr. Sebastiano Peluso for their support. I have been fortunate to have them in my PhD committee. I would also like to thank to Dr. Ahmed Hassan for providing helpful suggestions and comments during my research.

Thanks to the members of Scalable Systems Software Research Group for their support and friendship: Jacob Nelson, Pantea Zardoshti, dePaul Miller, Paul Grocholske, Yaodong Sheng and, Matthew Rodriguez.

Special thanks to my incredible friends Arash Amini, Saleh Teymouri, Abdolhamid Sadeghnejad and Mohammadhossein Mohammadi Siahroudi for listening, supporting, and encouraging me all the time. Last but not least, my infinite gratitude goes to my amazing friends Fatemeh Movafagh, Ghadir Asadi, Nasim Ebadi, Fatemeh Yazdandoust and, Ayoub Yari for their infinite kindness.

Contents

Acknowledgements **iv**

List of Tables **ix**

List of Figures **x**

List of Algorithms **xiii**

Abstract **1**

1 Introduction **4**

1.1 The Importance of Consistency in Concurrency Controls 6

1.1.1 External Consistency 8

1.1.2 Serializability 9

1.1.3 Snapshot Isolation, Generalized Snapshot Isolation, Parallel Snapshot Isolation 10

1.2 Fault Tolerance and Performance in Transactional Systems 11

1.3 Dissertation Motivation 13

1.4 SSS: Improving Serializability with External Consistency using Off-the-Shelf Hardware 15

1.5 FPSI: Improving Parallel Snapshot Isolation by Refreshing Reading Snapshot 17

1.6 Correctness of Transaction Processing with External Dependency 18

1.7 EPSI: Improving the performance and storage space in Replicated PSI Transactional Systems 19

1.8 Dissertation Organization . . . 20

2 Background, Terminologies and System Model **21**

2.1 Distributed Processes and Communication Primitives . . . 21

2.2 Data Organization . . . 22

2.3 Transaction Model . . . 22

2.4 History and Direct Serialization Graph . . . 23

3 Related Work **24**

3.1 Externally Consistent Protocols . . . 24

3.2 Serializable Protocols . . . 25

3.3 SI Protocols . . . 26

3.4 Protocols with other variants of SI . . . 26

3.5 Other Vector Clock/Physical Clock Protocols . . . 27

3.6 Erasure-coded Protocols . . . 28

4 SSS: Scalable Key-Value Store with External Consistent and Abort-free Read-only Transactions **30**

4.1 Overview . . . 30

4.2 SSS Concurrency Control . . . 31

4.2.1 Execution of Update Transactions . . . 33

4.2.2 Execution of Read-Only Transactions . . . 37

4.2.3 Examples . . . 41

4.3 Additional Considerations of SSS . . . 43

4.3.1 Garbage Collection . . . 43

4.3.2 Starvation . . . 43

4.3.3 Deadlock-Freedom . . . 43

4.3.4 Fault Tolerance . . . 44

4.4 Correctness Arguments . . . 44

4.5 Evaluation Study . . . 46

5 FPSI: Improving Read Guarantees in Parallel Snapshot Isolation 53

5.1 Overview 53

5.2 Background and Motivation 55

5.2.1 Walter and PSI 55

5.2.2 The Challenge of Updating Reading Snapshot in Walter 56

5.2.3 The Impact of Data Freshness in the Long Fork Anomaly 57

5.3 FPSI: Protocol Description 58

5.3.1 Metadata 58

5.3.2 Transactional Begin Operation 60

5.3.3 Transactional Write Operation 61

5.3.4 Transactional Read Operation 61

5.3.5 Commit protocol 65

5.3.6 Handling Asynchronous Messages 68

5.4 Correctness Arguments 68

5.5 Evaluation Study 70

5.5.1 YCSB 72

5.5.2 TPC-C 74

6 On the Correctness of Transaction Processing with External Dependency 77

6.1 Overview 77

6.2 Formalization 79

6.3 Designing a System using our Unified Model 81

7 EPSI: Efficient Erasure-coded Parallel Snapshot Isolation for Key-Value Stores 83

7.1 Overview 83

7.2 Background 85

7.2.1 Walter Protocol 85

7.2.2 Erasure Coding 89

7.3 System Model 90

7.4 Overview and General Architecture 91

7.5 Protocol Details . . . 96
7.5.1 Terminologies . . . 96
7.5.2 Transactional Begin Operation . . . 96
7.5.3 Transactional Write Operation . . . 97
7.5.4 Transactional Read Operation . . . 97
7.5.5 Transactional Commit Operation . . . 98
7.6 Evaluation Study . . . 100
7.6.1 Configuration and System Parameters . . . 100
7.6.2 Benchmarks and Workload . . . 100
7.6.3 Competitors . . . 101
7.6.4 Experimental Results . . . 101

8 Conclusions and Future Work **109**
8.1 Conclusion Remarks . . . 109
8.2 Future Works . . . 110

Bibliography **112**

Biography **124**

List of Tables

7.1 Application interfaces of EPSI and Walter. 91

7.2 Internal interfaces of Walter, used by EPSI's concurrency control. 92

List of Figures

1.1 The relation between external consistency, Serializability and SI variants. Each circle represents the set of accepted executions by each consistency level. External consistency is the strictest of the three. Serializability, and Snapshot Isolation both are strictly weaker than external consistency. 7

4.1 SSS execution in the presence of an anti-dependency. Orange boxes show the content of the data store. Gray boxes show transaction execution. Dashed line represents the waiting time for $T2$. The red crossed entries of Q(y) represent their elimination upon Remove. 41

4.2 Handling read-only transactions along with non-conflicting update transactions. We omitted snapshot-queue entries elimination upon Remove to improve readability. 42

4.3 Throughput of SSS against 2PC-baseline and Walter, varying % of read-only transactions. Number of nodes in X-axes. 48

4.4 Performance of SSS against 2PC-baseline using 5k objects and 50% read-only transactions. 50

4.5 Breakdown of SSS transaction latency. 50

4.6 SSS, 2PC-baseline, ROCOCO varying % of read-only transactions. Legend in (a) applies to (b). 51

4.7 Throughput of SSS against 2PC-baseline and Walter with 80% read-only transactions and 50% locality. 52

4.8 Speedup of SSS over ROCOCO and 2PC-baseline increasing the size of read-only transactions. 52

5.1 Long-fork anomaly accepted by PSI consistency level. Dashed arrows represent the asynchronous propagation messages. The reading snapshot of T_1 reflects the timestamp of T_2 in the second entry of T_1's vector clock ($T_1.VC$) but it does not reflect the timestamp of T_3 in the third entry of T_1's vector clock. The reading snapshot of T_4 reflects the timestamp of T_3 in the third entry of T_4's vector clock ($T_4.VC$) but it does not reflect the timestamp of T_2 in the third entry of T_4's vector clock. 57

5.2 Example of execution where a read-only transaction advances its reading snapshot and still reads consistently. VAS is the version-access-set. Bold vector clock entries show where *hasRead* is `true`. The red crossed entries of VAS represent their elimination upon Remove. 63

5.3 Example showing how an update transaction establishes its reading snapshot. 64

5.4 Throughput of FPSI, Walter and 2PC-baseline using YCSB and by varying % of read-only transactions, the total number of keys, and the number of nodes. 72

5.5 Average size of anti-dependency collected by update transactions during prepare phase of FPSI for different % of read-only transactions and keys. 73

5.6 Abort rate using 20 nodes and varying number of keys while delaying propagate messages in both FPSI and Walter. 74

5.7 Throughput of FPSI, Walter and 2PC-baseline using TPC-C and by varying % of read-only transactions, the number of warehouses per node (W/n), and the number of nodes. 75

5.8 Performance of FPSI and Walter varying the number of warehouses per node (W/n). 76

6.1 Designing an Externally Serializable Concurrency Control. The Serializable Concurrency Control determines provided rules by the serializable concurrency control. Externally Dependency Checker determines the information regarding the external dependency provided by programmer. 81

7.1 Transaction metadata and node metadata in Walter. 87

7.2 Example execution of a write transaction in EPSI. Dashed blue lines separate phases of transactional execution. Dashed arrows represent asynchronous messages. Overlapped arrows show parallel messages. 94

7.3 Throughput of YCSB varying the size of object values and 50% read-only transactions. 102

7.4 Throughput of YCSB varying the size of object values and 80% read-only transactions. 103

7.5 Throughput and Abort rate of YCSB varying the size of object values and 50% read-only transactions under the high contention (i.e., 10K keys). 105

7.6 Throughput of YCSB varying the size of object values and 50% read-only transactions using both a dedicated and a shared network in EPSI and Cocytus. 106

7.7 Throughput of YCSB varying the size of object values and 80% read-only transactions using both a dedicated and a shared network in EPSI and Cocytus. 107

7.8 Throughput of Retwis varying the size of data using EPSI and Walter. 108

List of Algorithms

1 Internal Commit of SSS of Transaction T in node N_i by SSS 34

2 Handling Internal Commit by Transaction T in node N_i using SSS 35

3 Start Pre-commit by Transaction T in node N_i using SSS 36

4 End Pre-commit of Transaction T in node N_i using SSS 37

5 Read Operation by Transaction T in node N_i using SSS 38

6 Version Selection Logic in node N_i using SSS 39

7 Begin procedure of transaction T in node N_i using FPSI 60

8 Read Operation in FPSI 61

9 Version Selection Logic in node N_i using FPSI 62

10 Commit of transaction T in node N_i using FPSI 66

11 FPSI's Commit message handlers received by node N_i for transaction T issued by node N_j 67

12 Remove and Propagate messages from transaction T issued by N_j to node N_i using FPSI 68

13 Walter API description in the node N_i 88

14 The procedure of $EPSIBegin(T)$ in node N_{shard_i} 96

15 Handling the procedure of $EPSIBegin(T)$ for the forwarded transaction T in $leader(shard_i)$ 97

16 The procedure of write operation in EPSI 97

17 Read Operation in the node N_i using EPSI 98

18 Commit of transaction T in $leader(shard_i)$ (i.e., N_i) using EPSI 98

Abstract

Distributed transactional systems are the standard medium to let client (or application) requests interact with a shared state, in the presence of concurrency. These systems require clients to wrap the application logic that acts on the shared state around a well-defined block of code, called transaction. Transactions are guaranteed to execute atomically and in isolation by the transactional system. Practical workloads in distributed transactional systems often involve a high number of interacting users whose requests need to be processed safely while maintaining high performance, scalability, and fault tolerance of the system.

The concurrency control is the component responsible for ensuring that operations from concurrent transactions are performed while preserving safety and ensuring a desired consistency level. A consistency level provided by a concurrency control directly affects the simplicity in which programmers develop distributed applications. The stronger the consistency level, the closer the behavior of the system to a serial one. For a programmer, a desirable consistency level is the one that does not violate application semantics and prevents the exposure of behaviors (or anomalies) that cause clients interacting with the system to observe "unrealistic" results (e.g., where read operations return obsolete values). In addition to providing a desirable consistency level, a transactional system should also provide fault tolerance, which refers to its capability of surviving failures without compromising the shared state.

The traditional approach to ensure fault tolerance is introducing redundant resources (e.g., storage) to preserve multiple copies. However, two problems arise from adopting redundant resources. First, these copies should be kept consistent, which adds additional design challenges to the synchronization protocol. As a result, system performance might

be negatively affected, especially if the assumption is that no centralized source of synchronization is deployed. Second, storage cost increases due to redundancy. This aspect can be neglected if the data repository is not very large. However, modern workloads, such as those produced by social networking applications, replicating the data repository would introduce an unfeasible increase in storage cost.

The research included in this dissertation has a twofold aim applied to distributed transactional systems, *i)* improving the data freshness of transactional read operations while adopting a fully decentralized design where no centralized source of synchronization is assumed; and *ii)* improving the cost of fault tolerance, namely the space overhead needed to ensure normal operation despite the presence of failures. These two properties are both appealing and orthogonal, therefore can be independently adopted by transactional systems.

In this dissertation, we are interested in the external consistency and Parallel Snapshot Isolation (PSI) consistency levels as they are widely accepted by a variety of real-world applications, spanning from On-Line Transaction Processing (OLTP) to social networking applications. Ensuring high level of data freshness in these consistency levels is the first key driving factor of the presented research. The level of data freshness is a property of the concurrency control formalizing the guarantees of a read operation in terms of obsoleteness of the returned value. Given the result of a read operation, the gap between the returned version of a shared object and its latest version quantifies the level of freshness of that read. The second key driving factor of the research included in this dissertation is to address the aforementioned problem of increased storage space due to redundant resources.

In order to improve the level of data freshness, we present two transactional systems named SSS and FPSI.

SSS is a transactional key-value store that guarantees external consistency. SSS supports abort-free read-only transactions and its novelty is in the deployment of a new technique, named snapshot-queuing, to propagate established serialization orders among concurrent transactions. Such a propagation enables update transactions to be serialized along with read-only transactions in a unique order where reads always return values written by the last update transaction returned to its client, hence providing the highest level of data freshness.

FPSI is a distributed transactional in-memory key-value store whose primary goal is to enable read operations to read fresher than existing implementations of the well-known Parallel Snapshot Isolation (PSI) consistency level, in the absence of a synchronized clock service among nodes. At the core of FPSI, a novel concurrency control allows abort-free read-only transactions to access the latest version of objects upon their first access to a node. FPSI does that by implementing a visible read technique, which lets read-only transactions to advance their logical clock upon the first access to a node, while retaining safety.

The third contribution in this dissertation overcomes the lack of theoretical framework to reasoning about correctness of distributed concurrency control implementations based on their data freshness.

The first two contributions of this research aim at providing fault tolerance through replication, which might cause a significantly large amount of data to be replicated. To address this challenge, we present EPSI, a transactional key-value store that integrates a layered concurrency control connected to an existing PSI to handle storage access. The general idea of EPSI is to deploy the erasure coding technique in its read and write operations and provide a desired system resiliency level through a lower storage cost, in comparison with the traditional replication techniques deployed by state-of-the-art transactional systems.

Chapter 1

Introduction

Web-accessible system software and applications are popular technological components exploited by (commercial or not) entities that aim at providing users (or clients) with innovative and appealing services. Recent advancements in hardware infrastructures for computing (e.g., multicore processors [1]) and network communication (e.g., Remote Direct Memory Access, or RDMA [2, 3]) enable such system software and applications to sustain an increasing number of requests issued by the connected users [4, 5]. Being able to improve the performance robustness of these systems while increasing the user workload (a property also known as scalability) and retaining desirable higher-level system properties, such as high programmability is an open problem that currently drives the development effort of leading companies, startups, and computer scientists in academia [6–8].

Real-world applications' workloads experience a massive amount of interacting clients [7, 9,10]. The nature of these interactions spans from simply querying a shared state, to manipulating it. In the presence of manipulations, advanced coordination management is needed to handle these concurrent accesses to the shared state and ensure all clients always observe correct results. In addition, to preserving safety in the presence of simultaneous requests accessing the shared state, theses systems should also provide reliability [11] in order to guarantee the survival of the shared state in case of unexpected failures. The common high-level approach to address this problem is to introduce redundant resources (e.g., processors, devices, storage) to preserve multiple copies [12–14]. However, these copies should be kept consistent, which adds additional design challenges since further synchronization might significantly decrease system performance [3].

Database transactions [15, 16] are the state-of-the-art programming abstraction used by applications to manipulate shared data in a correct manner. Transactions allow programmers to escape dealing with the pitfalls of concurrency by just enclosing shared data accesses (e.g., read, write, etc.) within a well-defined block of code (i.e., the transaction) [17]. Those transactions are then submitted to a transactional system, which has dedicated components responsible for producing a correct and concurrent execution of those transactions. At the core of a transactional system, there is the concurrency control, a component implementing a synchronization protocol allowing multiple simultaneous transactional executions to coexist and to perform read and write accesses in isolation and atomically (i.e., all changes to data are performed as if they were a single operation).

Concurrency controls have been studied intensively for several decades in different types of transactional repositories (e.g., distributed database management systems [18], distributed NoSQL/key-value stores [19–21], embedded databases [22, 23]). Many of these concurrency controls perform well under low applications' workload, however, their performance does not scale along with an increased number of client requests. Extensive research activities have been conducted to make transactional systems able to maintain high performance, scalability and fault tolerance while retaining a well-suited level of consistency (or correctness) for the clients [4, 5, 21, 21, 24–28, 28–34].

The general objective of a concurrency control is to improve the level of parallelism in which transactions are processed while still synchronizing their operations to satisfy a user-requested safety condition (or consistency level). Generally, strong (or strict) levels of consistency require transactions to undergo many synchronization steps, which often leads to decisions that force these transactions to behave conservatively, possibly resulting in low performance. On the other hand, allowing less synchronization among transactions usually increases the chance that most of them successfully finalize their executions without waiting for concurrent operations to be concluded first. This is the case of concurrency controls implementing relaxed (or less strict) consistency levels. The downside of concurrency controls providing relaxed consistency levels is that programmers might lose programmability [30], namely the property that expresses the simplicity for a programmer to use certain abstraction. Relaxing consistency means allowing transactions to order each other in a way that might contradict the traditional sequential reasoning. This entails anomaly [35, 36] due to concurrency are not anymore handled inside the transactional systems but they can be exposed to the application itself.

In Sections 1.1 and 1.2, we discuss the various consistency and fault tolerance levels in distributed transactional systems, and their effect on the system performance.

1.1 The Importance of Consistency in Concurrency Controls

In general, application programmers prefer strong levels of consistency to simplify the development of applications that deal with concurrency [24, 30, 37]. Relying on weak levels of consistency for an application means allowing more concurrent transactions to be committed, even in the presence of conflict (i.e., when two transactions access a common shared data and at least one of them is an update). Although performance is expected to increase in this case, transactions can be ordered by the concurrency control in a way that does not match any sequential execution of the same transactions. When that happens, the programmer needs to ensure that the application semantics is preserved, even when those reordered executions occur.

For a programmer, a desirable consistency level is the one that satisfies application semantics [38,39] and eliminates behaviors (or anomalies) [35,36] that cause clients interacting with the system observe “unrealistic" results (e.g., where read operations return obsolete values), and at the same time does not lead to low performance. By relying on an underlying system software that provides such a desirable consistency level, application development process is shortened and its complexity is decreased.

In this dissertation, we focus on three widely used consistency levels, namely *external consistency* [28, 40], *Serializability* [36, 37] and *Snapshot Isolation* (SI) [35, 36, 41]. The relation between these three isolation levels, in terms of allowed executions, is represented in Figure 1.1.

A consistency level α is stricter than a consistency level β if β accepts more transactional executions than α. The expectation is that β applies more relaxed synchronization rules

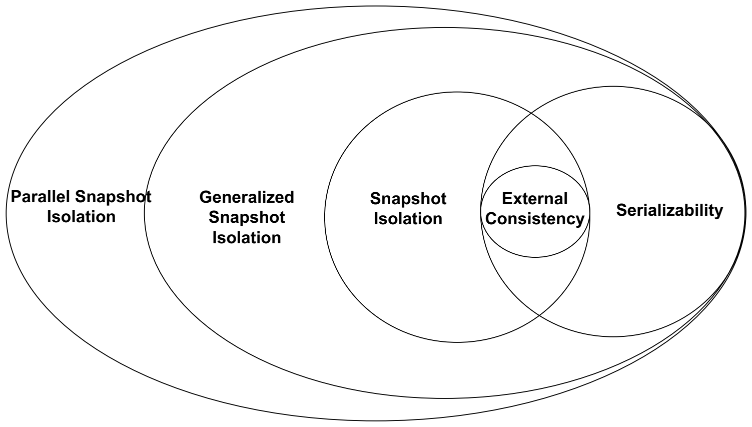


Figure 1.1: The relation between external consistency, Serializability and SI variants. Each circle represents the set of accepted executions by each consistency level. External consistency is the strictest of the three. Serializability, and Snapshot Isolation both are strictly weaker than external consistency.

across both concurrent and non-concurrent transactions than α, therefore increasing the number of possible committed executions. In this figure we can see how external consistency is strictly stronger than both Snapshot Isolation and Serializability; while Snapshot Isolation and Serializability are incomparable since each of them accepts executions that are rejected by the other, and vice versa.

In the remainder of this chapter, we describe each of our targeted consistency levels using examples borrowed by a simple monetary application. In order to do so, we consider two clients C_1 and C_2, each with two accounts, one for saving (Sav_1 and Sav_2) and one for checking (Chk_1 and Chk_2). Every client can issue his/her transaction (T_1 and T_2) and each issued transaction by clients contains read operations, for reading the value of some account, or write operations, for updating (i.e, depositing or withdrawing) the amount of some account.

$Read(Chk_i = amount)$ (or $Read(Sav_i = amount)$) represents a read operation returning the value *amount* for checking account Chk_i (or saving account Sav_i). $Write(Chk_i, +amount)$ (or $Write(Sav_i, +amount)$) determines a deposit operation of the value *amount* that should be done on Chk_i (or Sav_i). $Write(Chk_i, -amount)$ (or $Write(Sav_i, -amount)$) also stands for withdrawing the value *amount* from Chk_i (or Sav_i). For simplicity, let us assume that there exists \$10 in each client's saving account and checking account before running each example.

To better highlight the differences among the consistency levels discussed below we define the *transaction reading snapshot* as the set of versions returned by all the read operations of a transaction. As a metric of comparison, we define the *level of freshness* for a transaction reading snapshot as the metric that evaluates the gap between the read version of a shared object and its latest version. The higher is this gap, the older is the read version.

1.1.1 External Consistency

Suppose that transaction T_1, issued by C_1 and transaction T_2, issued by C_2 are as the following. T_1: $Read(Sav_1 = \$10)$ $Read(Chk_1 = \$10)$ $Write(Sav_1, +10)$. When T_1 completes its execution there will be \$20 in Sav_1 and \$10 in Chk_1. After completion of T_1, if T_2 starts its execution and completes as the following: T_2: $Read(Chk_1 = \$10)$ $Read(Sav_1 = \$20)$, then the updated value for Sav_1 by T_1 is visible by T_2. The latter property which is enforcing every transaction T_2, started after completion of transaction T_1, to observe the outcome of T_1 is guaranteed by a consistency level named external consistency.

External consistency is a strong level of consistency that clients often desire in an interactive transactional system, similar to the case of monetary application. Roughly, under external consistency a distributed system behaves as if all transactions were executed sequentially, all clients observed the same unique order of transactions completion (also named external schedule in [24]), in which every read operation returns the value written by the previous write operation. Therefore, in external consistency every read operation returns the most recent (we also name it as the *freshest*) version of an object, installed in the data repository before the transaction issuing the read operation started. Generally, a concurrency control provides the freshest reading snapshot if each read operation issued by every transaction returns the most recent version of a shared object.

By relying on the definition of external consistency, a transaction terminates when its execution is returned to its client; therefore the order defined by transaction client returns matches the order in which transactions read from other transactions (also known as serialization order). The latter property carries one great advantage: if clients communicate with each other outside the system, they cannot be confused about the possible mismatch between transaction order they observe and the transaction serialization order provided by the concurrency control inside the system. Simply, if a transaction is returned to its client, the serialization order of that transaction will be after any other transaction returned earlier, and before any transaction that will return subsequently. Therefore, external consistency goal is to make sure the value of shared objects is always up-to-date when they are returned back to clients.

1.1.2 Serializability

Serializability is one of the most targeted consistency criterion by transactional systems. Serializability (called One-Copy Serializability (1SR) [24, 35] in a distributed system where each object is replicated by multiple nodes), ensures that all completed transactions appear as executed serially in an environment where each shared object is not replicated. In this equivalent serial execution, where transactions do not overlap their operations, read operations of conflicting transactions should return the same values observed in the original execution. For non-conflicting transactions, any order is considered valid, as long as no two transactions observe different non-conflicting transactions in any opposite order.

In addition to that, while Serializability has been successfully used in a lot of applications (e.g., banking, airline reservations), applications might need guarantees from their transactions that demand different rules in deciding whether an execution can be committed depending upon the return value of its operations. These new requirements have motivated the introduction of different consistency levels that go beyond the Serializability. In the following section, we discuss Snapshot Isolation and its different variants, as one category of most used consistency level other than Serializability.

To connect the impact of a serializable concurrency control with the examples used in Section 1.1.1, if the transactions executed under Serializability, C_2 will still not be guaranteed to observe the deposited value by C_1. This is because using a serializable concurrency control, C_2 might decide to observe \$10 in each Sav_1 and Chk_1 which are applied to these accounts before C_1's deposit on Sav_1 is applied.

1.1.3 Snapshot Isolation, Generalized Snapshot Isolation, Parallel Snapshot Isolation

Snapshot Isolation (SI) [35] is a widely adopted consistency level often used as an alternative to Serializability for developing concurrency controls. It significantly improves the level of concurrency and performance over other known strong isolation levels by overcoming most (but not all) of the common anomalies due to concurrent data manipulation. It is supported by leading products, such as Oracle Database [42], Microsoft SQL Server [43], as well as by many other transactional repositories (e.g., [44]).

Consider the same monetary application this time executes two concurrent transactions T_1 and T_2 as the following. T_1: $Read(Sav_1 = 10)\ Write(Chk_1, +\$10)$ and T_2: $Read(Chk_1 = 10)\ Write(Sav_1, +\$10)$. In this case each transaction T_1 and T_2 updates a different account which are Chk_1 and Sav_1. Also they do not see the update of each other when they read. A serializable concurrency control considers T_1 and T_2 as conflicting transactions. And it never allows both T_1 and T_2 to commit, if they are conflicting. However, this specific conflict between T_1 and T_2 might be accepted by the monetary application due to its assumed semantics. The consistency level that allows both T_1 and T_2 completes while they are concurrent is named Snapshot Isolation (SI). In other words, two conflicting update transactions can complete in SI, as long as there is no intersection between the set of objects written by them.

One of the most important properties of SI, which enables high performance, is that read and write operations do not impede each other. A multi-versioned data repository, where multiple versions are recorded per shared object rather than just the one produced by the latest write, allows a read transaction to identify its *reading snapshot*. Generally, this reading snapshot is at least the set of versions available prior to the transaction's starting point (versions committed after a transaction begins can also be included in the reading snapshot). Because this set does not change while the transaction executes, concurrent writes cannot affect reads. The main difference of SI with respect to other strong consistency levels, such Serializability, is that it allows a transaction to have a serialization point for its read operations that is different from its serialization point for its write operations, unless they intersect.

Since a transaction's reading snapshot remains immutable throughout the execution, if a transaction only issues read operations without any write operation, then such a transaction will always be committed at its first trial. This property is very appealing because many real-world applications produce significant read-only workload [45]. Hence SI's level of concurrency, even among concurrent conflicting transactions, is high [46].

Generalized Snapshot Isolation (GSI) [41] is a weaker version of SI, mostly deployed in a way in which the starting point of a transaction is considered as a physical [47] or a logical timestamp [21, 48] acquired by each transaction when it begins its execution. This starting point might include outcomes of all transactions, or a subset of them, committed before a transaction T starts. If the starting point of T does not require T to observe the outcome of all transactions committed before T starts, then the system does not provide the freshest reading snapshot. Such a reading snapshot, which is prevented by SI, is accepted by GSI.

Parallel Snapshot Isolation (PSI) [21] is a weaker variant of GSI, therefore of SI, in which users might experience additional delays for their actions to be seen by other users, as guaranteed by SI instead. PSI introduces a new anomaly, called long-fork, which is caused by having non-conflicting transactions ordered in a different way by other transactions.

1.2 Fault Tolerance and Performance in Transactional Systems

The traditional approach to ensure fault tolerance and high availability of the shared state is replication [12, 14, 21, 30, 48–51]. In general, availability refers to the readiness of the system in providing a response of a client request. On the other hand, fault tolerance is the capability of the system to keep functioning in the presence of faults. In the rest of this section, we overview different replication schemes. In all of them, the state of the shared data is replicated $M + 1$ times to tolerate M failures.

Replicated schemes can be categorized as active or passive, depending on the role of the nodes holding object copies in the transaction processing.

The most deployed active replication approach is State Machine Replication (SMR) [14].

In SMR, clients' requests are processed by all replica nodes in the same order, and this order is traditionally established using a consensus layer that ensures total order [52, 53]. One known challenge of this scheme is achieving scalability since increasing nodes entails significantly increase the number of network messages exchanged to define a total order.

As apposed to the active replication, a passive replication scheme uses nodes as backups, ready to replace the primary node in case a failure occurs. A common deployment of passive replication is the Primary Backup Replication (PBR) [49, 54] technique. In PBR, one replica is the primary and executes clients' requests for changing shared data. The role of the Primary is also to send the modified state of the shared data to the backup replicas. If the primary replica fails, one of the backup replicas takes over its role of serving clients' request.

Partial Replication (PR) is an alternative approach proposed in the literature to minimize the number of contacted replicas every time a modification is made to the shared state. In fact, in partial replication, a transaction that commits a new modification to the shared state should only contact a subset of the system's nodes, not all as in SMR. This allows different replicas to handle independent parts of application workload in parallel [12, 21, 48, 50]. This approach improves SMR's scalability issue and also overcomes the high latency of PBR in case the primary node is overloaded. The disadvantage of PR when compared with SMR is the reduced resiliency, namely the capability of tolerating less failures than SMR.

When we compare the above techniques in terms of performance, moving from SMR to PR means increasing the correlation between the transaction latency and the performance of the slowest replica node for an object accessed by that transaction. In fact, in SMR an operation can be served by any node in the system, which inherently means that if a node is slow, other nodes can serve that operation quicker than it. The other extreme is PBR, where a transaction accessing an object can only contact one node, the primary, in order to retrieve its content [30]. If that node is slow, the transaction has no other choice but waiting for the primary serves the operation. On the other hand, PBR represents a middle ground where more than one node per object can be contacted to perform operations [21, 32, 48, 50].

Another technique to provide fault tolerance retaining the performance advantage of PR is erasure coding. In the erasure coding, one object can be divided into N coded data units and P equal-size coded parity units. Therefore, $M = N + P$ equal-size coding elements needed to tolerate P failures. Using an erasure coded data, any N of the M coded units can decode the original data due to a property called *maximum distance separable (MDS)* preserved by the erasure coded data [6, 55–61]. The latter property allows transactions to read from a quorum of nodes, as opposed to all the nodes replicating an object, and therefore avoiding high latency due to a slow replica node, as in PR.

An orthogonal, although appealing, property of erasure coding is that given a repository with a set of objects and a level of resiliency, erasure coding allows for reducing the amount of storage needed to store all the objects, compared to the other replication techniques [6, 7].

1.3 Dissertation Motivation

Seeking for the most appropriate level of consistency for a distributed transactional system given the characterization of a workload and/or application invariant has been the focus on much research in academia and industry [4, 5, 18, 21, 21, 24–28, 28–34, 48, 50, 62–66]. This dissertation specifically focuses on external consistency and snapshot isolation (SI) consistency level. This is because, looking at real-world application requirements, these two correctness criteria have been widely confirmed as appropriate for OLTP [46, 67], the former, and social network applications [10, 68], the latter.

Looking at the practicality of the above correctness levels, existing systems achieve them by means of deploying especial purpose hardware [4, 5, 18, 28, 62] to efficiently order distributed events. Through this special hardware, these systems can timestamp transactions and capture the total order of their operations. Although powerful, these systems cannot be easily adopted and extended because of the impossibility of accessing the technology by the majority of designers and developers.

The first set of results included in this dissertation aims at filling this gap, namely allowing developers to deploy systems with strong ordering requirements, without the need of relying on especial hardware. We generalize this requirement by designing systems that do not rely on a global source of synchronization to order transactions.

In literature, the most deployed technique for achieving global distributed synchronization is to use logical clocks (i.e., vector clocks [21, 50, 65, 69, 70]) for capturing causal dependency relations among distributed transactions. Logical clocks enable transactions to read and write data, consistently. However, the current literature of systems based on logical clocks lacks of solutions that allow transactions to enforce external consistency and retain important properties, such as the capability of always committing read-only workload at the first attempt (i.e., abort-freedom). The first contribution in this dissertation is SSS, a transactional system that updates vector clocks in a way transactions are always allowed to access the freshest reading snapshot and read-only transactions do not abort due to concurrency. SSS is overviewed in Section 1.4.

Snapshot Isolation has the same stringent ordering constrains in terms of consistency for its read operations as external consistency. Because of that, no solution is currently available in the literature of high-performance distributed systems that provide Snapshot Isolation without using a centralized or distributed clock service. On the other hand, a relaxation of such a correctness level, called Parallel Snapshot Isolation, has recently prevailed, motivated by its applicability to the popular social networking applications [21].

Despite its appeal, Parallel Snapshot Isolation imposes no restriction on the freshness of the returned value of read operations. The concept of data freshness is not formalized in literature and no practical existing high-performance system attempts to increase data freshness in Parallel Snapshot Isolation. The second contribution of this dissertation addresses the problem of data freshness in Parallel Snapshot Isolation. The goal of improving data freshness is achieved while maintaining the primary design goal of SSS, namely relying solely on off-the-shelf hardware [21, 48, 50, 65] to favor widespread adoption. We present FPSI, a distributed concurrency control that tracks causality and provides innovations to advance vector clocks during transaction execution. FPSI is overviewed in Section 1.5.

The third contribution in this dissertation overcomes the lack of theoretical foundation on reasoning about correctness of distributed concurrency control implementations based on their data freshness. Data freshness can be formalized as ordering constraints among transactions. We challenge the fact that proving the safety of distributed transactional systems by hand is error prone. To address this problem, we present a unified model to characterize the behavior of the concurrency control in the presence of ordering constrains and application invariant. The model is summarized in Section 1.6.

All the aforementioned systems provide fault tolerance through replication. While looking at the innovations motivated by the characteristics of social networking applications, we discovered that in order for the presented systems to be practical, traditional replication techniques should be rethought to account for the significantly high amount of stored data. The literature lacks of transactional systems that are optimized for storage cost and, at the same time, they achieve high performance by trading the level of freshness of data accesses. The intuition behind our forth contribution in this dissertation is to use erasure coding techniques for optimizing the storage cost of transactional systems whose distributed concurrency control is based on Parallel Snapshot Isolation. The key features of EPSI are summarized in Section 1.7.

1.4 SSS: Improving Serializability with External Consistency using Off-the-Shelf Hardware

We propose design and development of a system named *SSS*, which is a key-value store that implements a novel distributed concurrency control providing external consistency and assuming off-the-shelf hardware. The major innovations allowing SSS to enable high performance and scalability in SSS as the following:

- SSS supports read-only transactions that never abort due to concurrency, therefore the return value of all their read operations is always consistent at the time the operation is issued. We name them as *abort-free* hereafter. This property is very appealing because many real-world applications produce significant read-only workload [45].

- SSS goal is to provide availability and fault tolerance by deploying a general partial replication scheme where each key (or object) is replicated on multiple nodes without predefined partitioning schemes (e.g., sharding [29, 31]). To favor scalability, SSS does not rely on expensive ordering primitives, such as Total Order Broadcast or Multicast [71], but captures ordering relations using vector clocks.

In order to provide the above properties, we propose two techniques to be deployed in SSS which are explained below:

- SSS uses a vector clock-based technique to track dependent events originated on different nodes. This technique allows SSS to track events without a global source of synchronization.

- SSS uses the *snapshot-queuing* technique, where each key is associated with a *snapshot-queue*. Since read-only transactions surely commit in SSS, they are inserted into the snapshot-queues of their accessed keys in order to leave a trace of their existence to other concurrent transactions.

  SSS's proposed policy for update transactions is to insert these transactions into their modified keys' snapshot-queues after their commit decision is reached. Only update transactions can wait for read-only transactions if they belong to the same *snapshot-queue*. Read-only transactions leverage their membership into *snapshot-queue* to inform update transactions about their performed read operations.

  A transaction in a snapshot-queue is inserted along with a scalar value, called *insertion-snapshot*. This value represents the latest snapshot visible by the transaction on the node storing the accessed key, at the time the transaction is added to the snapshot queue. SSS concurrency control orders transactions with lesser insertion-snapshot before conflicting transactions with higher insertion-snapshot in the external schedule.

If a transaction T_R reads a shared object x subsequently modified by a concurrent committed transaction T_W, x's snapshot-queue is the medium to record the existence of an established serialization order between T_R and T_W, even if T_R and T_W operate on different nodes. With that, any other concurrent transaction accessing x can see this established order and define its serialization accordingly. Snapshot-queuing is designed to also work in replicated deployments, where an object is stored on multiple nodes, therefore each of these copies has an associated snapshot-queue. When T_W initiates its commit phase, it must contact all the copies of each written object, therefore being able to detect any concurrent T_R regardless of which node T_R reached for reading.

Snapshot-queuing enables the achievement of external consistency by delaying T_W's client response until T_R completes its execution. This delay is needed so that update and read-only transactions can be serialized in a unique order such that reads return values written by the last update transaction returned to its client. Failing in delaying T_W's response would result in a discrepancy between the external order and the transaction serialization order. In fact, the external order would show T_W returning earlier than T_R but T_R is serialized before T_W.

1.5 FPSI: Improving Parallel Snapshot Isolation by Refreshing Reading Snapshot

We present a distributed concurrency control implementation, named FPSI, for providing additional guarantees to the Parallel Snapshot Isolation (PSI) [21] consistency model. FPSI offers a solution to the absence of data freshness guarantees of PSI by making every transaction to select the freshest reading snapshot upon the first contacted node for distributed transactions, in the absence of a single source of synchronization. The key features allowing FPSI to achieve high performance and data freshness are listed below.

- FPSI preserves two properties of SI for read-only transactions. Read-only transactions are abort-free and they do not block update transactions. In some of the existing solutions such as Walter [21], all read-only transactions are serialized at their startup time. This design choice might impact freshness of data visible by transactions, a problem which is tackled in FPSI by attempting to advance the reading snapshot during the transaction execution when it is safe to do that. By advancing the reading snapshot for all transactions, FPSI allows read-only transactions to read fresher data than Walter, and update transactions to not experience false positive aborts due to outdated vector clocks, as opposed to Walter.

- FPSI improves transaction data freshness, latency, and throughput by avoiding unavailability of a viable snapshot to serve operations and a single point of failure, which characterizes solutions whose synchrony assumption is based on physical clocks [47].

The core components that make the above properties possible in FPSI are the following.

- FPSI uses a vector clock-based technique [69] to track dependent events that originated on different nodes. This technique is similar to the one used by existing distributed transactional systems, such as Walter [21] and GMU [50], and allows FPSI to track events without a centralized timestamp authority center [28, 62].

- FPSI uses the *version-access-set*, a metadata associated with each version containing identifiers of read-only transactions that read that specific version. During the commit phase of an update transaction, the set of identifiers of concurrent conflicting read-only transactions is collected. This set is then propagated to the version-access-sets of the newly created versions of those update transactions upon commit. If a read-only transaction contacts a node for the first time, it can advance its reading snapshot unless it finds that its own identifier exists in the version-access-set of the version to be read. In that case, the read-only transaction should select a previous version whose version-access-set does not contain its identifier.

1.6 Correctness of Transaction Processing with External Dependency

We propose to classify relations among committed transactions into data-related and application semantic-related. Our model delivers a condition that can be used to verify the safety of transactional executions in the presence of application invariant.

When the concurrency control implementation of a transactional system is required to enforce an application-level invariant on shared data accesses (i.e., an expression that should be preserved upon every atomic update [38]), ad-hoc reasoning about its correctness is a tedious and error-prone process. Traditional (data-related) constraints (e.g., transaction conflicts) are well-formalized with established correctness levels, such as Serializability and Snapshot Isolation [36]. However, a unified model encompassing the various *external* (semantic-related) constraints that enforce application invariant has not been formalized yet.

We make a step towards defining such a model. We introduce a theoretical framework that formalizes correctness levels stronger than (or equal to) Serializability by defining their transaction ordering relations as a union of two sets of data and external dependency. This approach is opposed to the traditional way of defining these relations through an ad-hoc analysis. This framework can be used to define an offline checker that verifies the safety of transactional executions.

1.7 EPSI: Improving the performance and storage space in Replicated PSI Transactional Systems

We present EPSI as a sharded key-value store that processes transactions and is designed to optimize the data repository's storage space. EPSI accomplishes the former goal by leveraging a distributed concurrency control that provides the Parallel Snapshot Isolation (PSI) correctness level. On the other hand, the goal of optimizing storage cost is accomplished by relying on the erasure coding technique. EPSI is motivated by the workload characteristics of social networking applications, in which storage cost, fault tolerance, and high availability are vital factors, as opposed to providing transactions with a strong consistency level. Specifically reducing storage cost is a requirement that derives from the fact that users of social applications post content of heterogeneous nature and various size, often much bigger than traditional OLTP application workloads.

At the core of EPSI there is a concurrency control middleware that exposes transactional APIs to the application and leverages the APIs of an existing distributed concurrency control to read and write data stored in the data repository, in parallel. Since EPSI finds its optimal deployment with social applications, we decide to adopt Walter [21] as underlying concurrency control. This is because Walter's consistency level is PSI, which is designed to fit the requirements of social applications. Walter's APIs to read and write on the shared data repository are used to store coding elements, namely chunks of the original application objects, encoded to satisfy the used erasure coding technique.

To the best of our knowledge, there is no transactional repository in the literature that optimizes storage cost and retains high performance for social applications.

The key features of EPSI are summarized below.

- EPSI provides an efficient sharded transactional key-value store ensuring the same level of fault tolerant and performance as the state-of-the-art PSI protocol, such as Walter. In EPSI, each shard implements the Reed Solomon erasure coding scheme where every node stores either coded data units or parity coded units. Depending on the desired system resiliency level, EPSI can be configured to produce the right coding elements to explore the trade off between fault tolerance and cost space utilization.

- EPSI provides scalable performance with respect to the size of the stored objects, due to a more effective network utilization in the presence of write workload. Existing solutions, such as Cocytus [57], saturate the network bandwidth when the object size increases. The main reason this improved performance is the fact that EPSI exploits the characteristics of erasure coding to improve performance during normal operations. The direct consequence of this decision is that read operations can be served reading from a quorum of nodes. This feature is particularly useful when the object size is high. As opposed to EPSI, Cocytus [57] uses erasure coding only during recovery.

1.8 Dissertation Organization

This dissertation is organized as follows. Chapter 2 provides the background for the presented research contributions in this dissertation. Chapter 3 discusses the related previous works in the space. Chapter 4 describes and evaluates SSS. Chapter 5 describes and evaluates FPSI. In Chapter 6, a description of our unified model to characterize consistency levels stronger (or equal to) Serializability in the presence of application invariant is presented. In Chapter 7, EPSI's architecture is discussed and its performance is extensively evaluated. Finally, Chapter 8 concludes the provided research contributions and discusses future work.

Chapter 2

Background, Terminologies and System Model

2.1 Distributed Processes and Communication Primitives

We model a distributed system as a set of nodes such that $\Pi = \{N_1, ..., N_n\}$. Nodes communicate through message passing and do not have access to either a shared memory or a global clock. There is not any timing assumptions on sending or receiving messages and they might experience arbitrarily long (but finite) delays. There is no assumption on the speed and on the level of synchrony among nodes. We consider the classic crash-stop failure model: nodes may fail by crashing, but do not behave maliciously. A node that never crashes is correct; otherwise it is faulty.

Nodes communicate through message passing and reliable asynchronous channels, meaning the system provides nodes with primitives to send and receive point-to-point messages on reliable channels via the primitive *send(m)* and *receive(m)* such that for any two correct nodes N_i and N_j, if N_i sends message m to N_j then N_j eventually receives m.

Clients are assumed to be colocated with nodes in the system; this way a client is immediately notified of a transaction's commit or abort outcome, without additional delay. Clients are allowed to interact with each other through channels that are not provided by the system's APIs.

2.2 Data Organization

Every node N_i maintains shared objects (or keys) adhering to the key-value model [50] (either partial or full) copy of data). Multiple versions are kept for each key k. Each version ver, is equal to a pair such that $ver =< val, vc >$, all associated with a key k. The field val represents the value of k and vc is the logical vector timestamp (or vector clock) associated with the commit of ver. In this dissertation it is assumed that the size of vector clocks is equal to the number of nodes in the system, or the number of shards if the data repository is sharded. While acknowledging that the size of vector clocks grows linearly with the system size, there are existing orthogonal solutions to increase the granularity of such a synchronization to retain efficiency [72, 73].

Given a set of versions associated with each key k and stored on node n_i, the value of vc have a monotonically increasing entry going from the oldest version to the most recent committed version. Throughout the dissertation the binary relation $\leq$ defines an order for vc such that for each two vector clocks v_1 and v_2, the pair $< v_1, v_2 >$ is in $\leq$ written $v_1 \leq v_2$ if $\forall i$, $v_1[i] \leq v_2[i]$. If $<$ is the standard less relation defined for natural numbers and if there exists also an index j such that $v_1[j] < v_2[j]$, then $v_1[i] < v_2[i]$.

Every shared key can be stored in one or more nodes, depending upon the chosen replication degree. Formally we say the replication degree is r if the following conditions are satisfied in the system. Objects are subdivided across m partitions, and each partition is replicated across r nodes. Each group of node which replicate object o is composed of exactly r nodes (to ensure the target replication degree), of which at least a majority is assumed to be correct. $replicas(S)$ denotes the set of nodes that replicate the data partitions containing all the keys $k \in S$, called also owners of S. For object reachability, we assume the existence of a local look-up function that matches keys with nodes.

2.3 Transaction Model

We model transactions as programs executing a sequence of *read* and *write* operations on shared keys, preceded by a *begin*, and followed by a *commit* or *abort* operation. Local computation in between operations on shared keys performed by transactions is also permitted.

Transactions ensure the ACID properties.

Every transaction starts with a client submitting it to the system, and finishes its execution informing the client about its final outcome: *commit* or *abort*. Transactions that do not execute any write operation are called read-only, otherwise they are update transactions. We assume programmer identify whether a transaction is update or read-only. Every operation of a transaction is mapped to only a unique version in the system and a read operation can only return one committed value.

2.4 History and Direct Serialization Graph

A history (as is also defined in [35, 36]) models the interleaved execution of a set of transactions T_i as a linear ordering of their operations (such as Reads and Writes). A direct serialization graph $\mathcal{DSG}(\mathcal{H})$ [36] on a history $\mathcal{H}$ is a graph with a vertex T_i for each transaction T_i in $\mathcal{H}$ and an edge $T_i \neq T_j$ for each pair of transactions T_i and T_j in $\mathcal{H}$ such that T_i and T_j are conflicting. The $\mathcal{DSG}(\mathcal{H})$ contains three types of edges depending on the three types of conflicts, also called dependencies, that two transactions T_i, T_j can have in $\mathcal{H}$:

- *read dependency:* ($T_i \xrightarrow{\text{WR}} T_j$) A transaction T_j read-depends on T_i if a read of T_j returns a value written by T_i.
- *write dependency:* ($T_i \xrightarrow{\text{WW}} T_j$) A transaction T_j write-depends on T_i if a write of T_j overwrites a value written by T_i.
- *anti-dependency:* ($T_i \xrightarrow{\text{RW}} T_j$) A transaction T_j anti-depends on T_i if a write of T_j overwrites a value previously read by T_i.

Chapter 3

Related Work

In this chapter, we overview the literature that relates to our solutions. In Sections 3.1-3.5, we discussed the related work based on the consistency level guaranteed by each of the related solution. Section 3.6 discusses how the existing solutions achieve fault tolerance using different data replication techniques to improve the performance or the storage space.

3.1 Externally Consistent Protocols

Google Spanner [28] is a high performance solution that leverages a global source of synchronization to timestamp transactions so that a total order among them can always be determined, including when nodes are in different geographic locations. This form of synchronization is materialized by the *TrueTime* API. This API uses a combination of a very fast dedicated network, GPS, and atomic clocks to provide the accuracy of the assigned timestamps. Spanner provides properties similar to SSS, but its architecture needs special-purpose hardware and therefore it cannot be easily adopted and extended. On the other hand, SSS relies on off-the-shelf hardware.

Scatter provides external consistency on top of a Paxos-replicated log. The major difference with SSS is that Scatter only supports single key transactions while SSS provides a more general semantics. ROCOCO [30] uses a two-round protocol to establish an external schedule in the system, however, unlike SSS, it does not support abort-free read-only transactions. In ROCOCO read-only transactions wait for conflicting transactions to complete.

Calvin [15] uses a deterministic locking protocol supported by a sequencer layer that orders transactions. In order to do that, Calvin requires a priori knowledge on accessed read and written objects. Although the sequencer can potentially be able to assign transaction timestamp to meet external consistency requirements, SSS does that without assuming knowledge of read-set and write-set prior transaction execution and without the need of such a global source of synchronization.

FaRM [5] is a distributed externally consistent computing platform which uses RDMA to directly access data in a shared address space, and for fast messaging between the nodes. SSS is designed to not leverage special purpose hardware.

3.2 Serializable Protocols

Replicated Commit [74] provides Serializability by replicating the commit operation using 2PC in every data center and Paxos [52] to establish consensus across data centers. In Replicated Commit, read operations require contacting all data centers and collect replies from a majority of them in order to proceed while SSS's read operations are handled by the fastest replying server.

Granola [75] ensures Serializability using a timestamp-based approach with a loosely synchronized clock per node. Granola provides its best performance when transactions can be defined as independent, meaning they can entirely execute on a single server. SSS has no restriction on transaction accesses.

CockroachDB [18] uses a serializable optimistic concurrency control, which processes transactions by relying on multi-versioning and timestamp-ordering. The main difference with SSS is the way consistent reads are implemented. CockroachDB relies on consensus while SSS needs only to contact the fastest replica of an object.

SCORe [48], guarantees similar properties as SSS, but it fails to ensure external consistency since it relies on a single non-synchronized scalar timestamp per node to order transactions, and therefore its abort-free read-only transactions might be forced to read old version of shared objects.

FaSST [4] is an RDMA-based system that provides distributed in-memory transactions with Serializability. SSS is designed to not leverage special purpose hardware.

3.3 SI Protocols

Clock-SI [47] provides SI using a loosely synchronized clock scheme. Google Percolator [62] guarantees SI using a centralized source of synchronization to timestamp distributed transactions for Bigtable [76]. With respect to FPSI and SSS, Clock-SI might lead to the unavailability of the reading snapshot because of skews across distributed clocks, with a consequence low performance. In terms of comparing the consistency levels, FPSI supports PSI by allowing read-only transactions to access the latest version of objects upon their first contact to a node and SSS guarantees a stronger level of consistency than Clock-SI using additional metadata.

The solution in [77] proposed the Incremental Snapshot method as an efficient solution to implement Distributed Snapshot Isolation. In this method, a local transaction only interacts with the local clock to establish the reading snapshot. A non-local transaction interacts with the remote node to obtain an appropriate reading snapshot. For validating the remote accesses, a global clock is still required. The validation requires maintaining the mapping between each local clock and the global clock.

3.4 Protocols with other variants of SI

Many distributed transactional repositories provide weaker variants of SI for the sake of achieving high performance. Examples of these systems include [21, 63, 78].

Jessy [63], provides transactions with reading snapshots that can include versions committed after the transaction starting time and that are produced by causally dependent transactions. The latter property improves progress for read operations and reduces abort rate when compared to SI solutions. Jessy uses per-version dependence vectors. Each vector reflects all the versions read or written by the transaction that created that specific version. FPSI and Jessy both aim at improving data freshness; however, unlike FPSI, the amount of metadata required to support execution grows significantly. In fact, if Jessy transactions access random objects, the size of each dependence vector is comparable with the total number of objects in the system.

Walter provides the same level of consistency that FPSI provides (PSI). However, FPSI improves data freshness by eliminating the effect of asynchronous propagations on read-only transactions. FPSI traces (anti-)dependencies and allows all transactions to advance their reading snapshots at no significant performance degradation.

Elnikety et. al in [78] extend SI to replicated databases. It allows transactions to use local snapshots of the database on each replica, which relaxes the level of data freshness as opposed to FPSI.

3.5 Other Vector Clock/Physical Clock Protocols

GMU [50] provides transactions with the possibility to read the latest version of an object by using vector clocks; however it cannot guarantee serializable transactions. SSS [32] improves upon GMU by relaxing the G-read anomaly, which affects GMU. This is done by delaying concurrent conflicting transactions, which also allows for achieving external consistency. With respect to FPSI, GMU provides transactions with the possibility to read the latest version of an object by using vector clocks; however, it needs more communication steps for validating read objects by update transactions.

GentleRain [66], Orbe [64] and Cure [65] all provide causal consistency, which is a consistency level weaker than external consistency (provided by SSS) and Parallel Snapshot Isolation (provided by FPSI). GentleRain uses loosely synchronized physical clocks to ensure that clients, after a read on an object version, will be able to serve subsequent reads without the need of explicitly checking dependencies. This happens by waiting for a causally consistent version to be available. Cure is a vector clock based technique similar to SSS and FPSI with a weaker consistency level. Orbe is a key-value store protocol relies on loosely synchronized physical clocks to provide causally consistent read-only transaction and use a similar technique to GentleRain for reading and catching up in the presence of clock skew.

3.6 Erasure-coded Protocols

Large scale in memory key-value stores like Memcached [79] and Redis [80] have been widely used in Facebook [7] and Twitter [10]. On the other hand, a large amount of existing works in recent years has been focused on building high performance key-value stores using erasure coding [5, 57–59, 81–84]. In the following, we move our focus on such key-value stores.

Some in memory key-value stores, such as [57, 83, 84], combine erasure coding and replication. They store keys and metadata using replication while storing values of objects relying on erasure coding (i.e., hybrid encoding). Cocytus integrates the Primary Backup Replication (PBR) [49] with erasure coding, and to provide a transactional key-value store. Transactional accesses in Cocytus access the non-erasure coding data by contacting the node where the data is stored. During recovery, the system recalculates lost data coding elements or parity coding elements using the Reed Solomon technique [55]. Cocytus uses Two-Phase Commit [85] in its underlying transactional layer and provides Serializability [37]. LH^*_{RS} [83] uses a variant of Two-Phase Commit to retaining data delta backups in a temporary buffer for possible rollbacks. Both Cocytus and LH^*_{RS} needs more storage space in comparison with EPSI.

MemEC [58] is an erasure coding-based in-memory key-value store which encodes objects entirety (e.g., keys, values, and metadata) without replication. MemEC's fully erasure coded technique is effective when the value size of the key is small (e.g., less than 1KB). EPSI's erasure coded design allows for high performance for a larger size of values.

Giza [81] deploys erasure coding by storing coding elements of keys in the storage layer and replicating metadata across data centers. Giza supports single operations such as put, get, and delete. It relies on the FastPaxos [86] protocol, in case of not conflicting operations and it deploys classic Paxos [52] in case of having concurrent conflicting updates. Unlike EPSI, Giza does not support the general transactional scheme.

BCStore [82] designs a batch coding mechanism to achieve high bandwidth efficiency for write workload using erasure coding for a single operation and write-intensive workloads. EPSI in comparison with BCStore is able to provide high throughput for read-only workload and supports the general transactional scheme.

Memc3 [59] is an in-memory key-value store that targets improving Memcached performance. Memc3 presents a set of workload inspired algorithms such as optimistic cuckoo hashing and optimistic locking to achieves high memory efficiency and allows multiple readers and a single writer to concurrently access the hash table. In comparison with EPSI, EPSI's transactional architecture allows concurrent read-only to proceed with concurrent update workload.

Konwar et. al. in [60], proposed the Layered Distributed Storage (LDS) algorithm by relying on regenerating codes [87] instead of Reed Solomon codes. In LDS a write operation completes after writing the object value to the first layer and can be exposed to the read operations which are concurrent with write. The latter improved the performance of reads by making them be served directly from the first layer. Otherwise, reads are served by regenerating coded data from the second layer.

Chan et. al. in [88] proposed a solution to improve the performance of update transactions. The solution provided in [88] is called parity logging with reserved space, which uses a hybrid of in place data updates and log-based parity updates. For mitigating the disk seeks for reading parity chunks in their proposed solution, deltas of parity chunks in a reserved space that is allocated next to their parity chunks.

Chapter 4

SSS: Scalable Key-Value Store with External Consistent and Abort-free Read-only Transactions

4.1 Overview

We present a scalable transactional key-value store, named SSS, deploying a novel distributed concurrency control that provides external consistency for all transactions, never aborts read-only transactions due to concurrency, all without specialized hardware. SSS's concurrency control uses a combination of vector clocks and a new technique, called snapshot-queuing, to establish a single serialization order where transactions are guaranteed to read from the latest non-concurrent transaction externally visible to clients.

With the delivery of SSS, we make the following contributions:

- SSS implements the first distributed transactional protocol for general purpose replicated systems where read-only transactions read consistently the latest committed version of objects without relying on a single synchronization service and without aborting.

- SSS's synchronization technique to serialize read-only and update transactions is the first to merge the semantics of vector clocks [69] with visible read operations [89] to produce the snapshot-queuing technique.

- SSS solves the problem of serializing two non-conflicting update transactions in different orders when multiple conflicting read-only transactions execute on different nodes [21, 36, 50], without relying on a single synchronization service and without aborting the read-only transactions.

In Section 4.2, we describe the SSS concurrency control, followed by two execution examples. In Section 4.3 we describe additional considerations of SSS. Section 4.4 and Section 4.5 provide the correctness arguments and evaluation study respectively.

4.2 SSS Concurrency Control

In the following we show the details regarding the metadata and different steps of the concurrency control of SSS.

Transaction vector clocks. In SSS a transaction T holds two vector clocks, whose size is equal to the number of nodes in the system. One represents its actual dependencies with transactions on other nodes, called `T.VC`; the other records the nodes where the transaction read from, called `T.hasRead`.

`T.VC` represents a version visibility bound for T. Once a transaction begins in node N_i, it assigns the vector clock of the latest committed transaction in N_i to its own `T.VC`. Every time T reads from a node N_j for the first time during its execution, `T.VC` is modified based on the latest committed vector clock visible by T on N_j. After that, `T.hasRead[j]` is set to true.

Transaction read-set and write-set. Every transaction holds two private buffers. One is rs (or *read-set*), which stores the keys read by the transaction during its execution, along with their value. The other buffer is ws (or *write-set*), which contains the keys the transaction wrote, along with their value.

Snapshot-queue. A fundamental component allowing SSS to establish a unique external schedule is the snapshot-queuing technique. With that, each key is associated with an ordered queue (`SQueue`) containing: read-only transactions that read that key; and update transactions that wrote that key while a read-only transaction was reading it.

Entries in a snapshot-queue (`SQueue`) are in the form of tuples. Each tuple contains: transaction identifier $T.id$, the *insertion-snapshot*, and transaction type (read-only or update). In general, the *insertion-snapshot* for a transaction T enqueued on some node N_i's snapshot-queue is the value of T's vector clock in position i^{th} at the time T is inserted in the snapshot-queue (see Section 4.2.1 and Section 4.2.2 for the actual value of the insertion snapshot, which varies depending upon the transaction type). Transactions in a snapshot-queue are ordered according to their insertion-snapshot.

A snapshot-queue contains only transactions that will commit; in fact, besides read-only transactions that are abort-free, update transactions are inserted in the snapshot-queue only after their commit decision has been reached.

Transaction transitive anti-dependencies set. An update transaction maintains a list of snapshot-queue entries, named `T.PropagatedSet`, which is populated during the transaction's read operations. This set serves the purpose of propagating anti-dependencies previously observed by conflicting update transactions.

Node's vector clock. Each node N_i is associated with a vector clock, called `NodeVC`. The i^{th} entry of `NodeVC` is incremented when N_i is involved in the commit phase of a transaction that writes some key replicated by N_i. The value of j^{th} entry of `NodeVC` in N_i is the value of the j^{th} entry of `NodeVC` in N_j at the latest time N_i and N_j cooperated in the commit phase of a transaction.

Commit repositories. `CommitQ` is an ordered queue, one per node, which is used by SSS to ensure that non-conflicting update transactions are ordered in the same way on the nodes where they commit. `CommitQ` stores tuples $< T,\ vc,\ s >$ with the following semantics. When an update transaction T, with commit vector clock vc, enters its commit phase, it is firstly added to the `CommitQ` of the nodes participating in its commit phase with its status s set as *pending*.

When the transaction commit phase concludes successfully, the status of the transaction is changed to *ready*. A ready transaction inside the `CommitQ` is assigned with a final vector clock produced during the commit phase. In each node N_i, transactions are ordered in the `CommitQ` according to the i^{th} entry of the vector clock ($vc[i]$). This allows them to be committed in N_i with the order given by $vc[i]$. Although the commitment of non-conflicting transactions in a sequential way on a node might reduce performance, it is indeed needed to guarantee a single serialization order with respect to the nodes replicating the same keys [50].

When T commits, it is deleted from `CommitQ` and its vc is added to a per node repository, named `NLog`. We identify the most recent vc in the `NLog` as `NLog.mostRecentVC`.

Overall, the presence of additional metadata to be transferred over the network might appear as a barrier to achieve high performance. To alleviate these costs we adopt metadata compression. In addition, while acknowledging that the size of vector clocks grows linearly with the system size, there are existing orthogonal solutions to increase the granularity of such a synchronization to retain efficiency [72, 73].

4.2.1 Execution of Update Transactions

Update transactions in SSS implements lazy update [90], meaning their written keys are not immediately visible and accessible at the time of the write operation, but they are logged into the transactions write-set and become visible only at commit time. In addition, transactions record the information associated with each read key into their read-set.

Read operations of update transactions in SSS simply return the most recent version of their requested keys (Lines 21-23 of Algorithm 6). At commit time, validation is used to verify that all the read versions have not been overwritten.

An update transaction that completes all its operations and commits cannot inform its client if it observes anti-dependency with one or more read-only transactions. In order to capture this waiting stage, we introduce the following phases to finalize an update transaction (Figure 4.1 pictures them in a running example).

Internal Commit. When an update transaction successfully completes its commit phase, we say that it commits internally. In this stage, the keys written by the transactions are visible to other transactions, but its client has not been informed yet about the transaction completion. Algorithms 1 and 2 show the steps taken by SSS to commit a transaction internally.

Algorithm 1 Internal Commit of SSS of Transaction T in node N_i by SSS

```
1:  function COMMIT($TransactionT$)
        // Check if T is a read-only transaction
2:      if ($T.ws = \phi$) then
3:          for ($k \in T.rs$) do
4:              send $Remove[T]$ to all $replicas(k)$
5:          $T.outcome \leftarrow true$
6:          return $T.outcome$
        // Start 2PC if T is an update transaction
7:      $commitVC \leftarrow T.VC$
8:      $T.outcome \leftarrow true$
9:      send $Prepare[T]$ to all $N_j \in replicas(T.rs \cup T.ws) \cup N_i$
10:     for all ($N_j \in replicas(T.rs \cup T.ws) \cup N_i$) do
11:         wait receive $Vote[T.id, VC_j, res]$ from $N_j$ or timeout
        // Check if T's 2PC commit decision was successful
12:         if ($\neg res \vee timeout$) then
13:             $T.outcome \leftarrow false$
14:             break;
15:         else
16:             $commitVC \leftarrow \max(commitVC, VC_j)$
17:     $xactVN \leftarrow \max\{commitVC[w] : N_w \in replicas(T.ws)\}$
        // Finalize T's commit vector clock
18:     for all ($N_j \in replicas(T.ws)$) do
19:         $commitVC[j] \leftarrow xactVN$
20:     send $Decide[T, commitVC, outcome]$ to all $N_j \in replicas(T.rs \cup T.ws) \cup N_i$
21:     return $T.outcome$
22: function VALIDATE($Set\ rs, VC\ T.VC$)
        // Check if T's read keys are not overwritten
23:     for all ($k \in rs$) do
24:         if ($k.last.vc[i] > T.VC[i]$) then
25:             return $false$
26:     return $true$
```

SSS relies on the Two-Phase Commit protocol (2PC) to internally commit update transactions. The node that carries the execution of a transaction T, known as its coordinator, initiates 2PC issuing the prepare phase, in which it contacts all nodes storing keys in the read-set and write-set. When a participant node N_i receives a prepare message for T, all keys read/written by T and stored by N_i are locked. If the locking acquisition succeeds, all keys read by T and stored by N_i are validated by checking if the latest version of a key matches the read one (Lines 22-26 Algorithm 1). If successful, N_i replies to T's coordinator with a `Vote` message, along with a proposed commit vector clock. This vector clock is equal to N_i's $NodeVC$ where $NodeVC[i]$ has been incremented. Finally, T is inserted into N_i's `CommitQ` with its $T.VC$.

Algorithm 2 Handling Internal Commit by Transaction T in node N_i using SSS

```
1: upon receive $Prepare[Transaction\ T]$ from $N_j$ do
2:     $boolean\ outcome \leftarrow getExclusiveLocks(T.id, T.ws) \wedge getSharedLocks(T.id, T.rs)$
        $\wedge validate(T.rs, T.VC)$
3:     if ($\neg outcome$) then
4:         $releaseLocks(T.id, T.rs, T, ws)$
5:         send $Vote[T.id, T.VC, outcome]$ to $N_j$
6:     else
7:         $prepVC \leftarrow NLog.mostRecentVC$
8:         if ($N_i \in replicas(T.ws)$) then
9:             $NodeVC[i]++$
10:            $prepVC \leftarrow NodeVC$
11:            $CommitQ.put(< T, prepVC, pending >)$
12:        send $Vote[T.id, prepVC, outcome]$ to $N_j$
13: end
14: upon receive $Decide[T, commitVC, outcome]$ from $N_j$ atomically do
15:     if ($outcome$) then
16:         $NodeVC \leftarrow \max(NodeVC, commitVC)$
17:         if ($N_i \in replicas(T.ws)$) then
18:             $CommitQ.update(< T, commitVC, ready >)$
19:     else
20:         $CommitQ.remove(T)$
21:         $releaseLocks(T.id, T.ws, T.rs)$
22: end
23: upon $\exists < T, vc, s >$:  $< T, vc, s >= commitQ.head \wedge s = ready$ do
    // Finalize internal commit of T
24:     for all ($k \in T.ws : N_i \in replicas(k)$) do
25:         $apply(k,\ val,\ vc)$
26:     $NLog.add(< vc >)$
27:     $CommitQ.remove(T)$
28:     $releaseLocks(T.id, T.ws, T.rs)$
29: end
```

After receiving each successful `Vote`, T's coordinator computes the commit vector clock ($commitVC$) by calculating the maximum per entry (Line 16 of Algorithm 1). This update makes T able to include the causal dependencies of the latest committed transactions in all

2PC participants. After receiving all `Vote` messages, the coordinator determines the final commit vector clock for T as in (Lines 16-19 of Algorithm 1), and sends it along with the 2PC `Decide` message.

Lines 14-22 of Algorithm 2 shows how 2PC participants handle the `Decide` message. When N_i receives `Decide` for transaction T, N_i's $NodeVC$ is updated by computing the maximum with $commitVC$. Importantly, at this stage the order of T in the $CommitQ$ of N_i might change because the final commit vector clock of T has been just defined, and it might be different from the one used during the 2PC prepare phase when T has been added to $CommitQ$.

In Algorithm 2 Lines 23-29, when transaction T becomes the top standing of N_i's $CommitQ$, the internal commit of T is completed by inserting its commit vector clock into the $NLog$ and removing T from $CommitQ$. When transaction's vector clock is inserted into the node's `NLog`, its written keys become accessible by other transactions. At this stage, T's client has not been informed yet about T's internal commit.

Pre-Commit. An internally committed transaction spontaneously enters the Pre-Commit phase after that. Algorithm 3 shows detail of Pre-commit phase. At this stage, T evaluates if it should hold the reply to its client depending upon the content of the snapshot-queues of its written keys. If so, T will be inserted into the snapshot-queue of its written keys in N_i with $commitVC[i]$ as *insertion-snapshot*.

Algorithm 3 Start Pre-commit by Transaction T in node N_i using SSS

```
1: for all ($k \in T.ws$) do
2:     if ($N_i \in replicas(k)$) then
3:         $k.SQueue.insert(< T.id, T.commitVC[i], “W” >)$
4:         for all ($T^{'} \in T.PropagatedSet$) do
5:             $k.SQueue.insert(< T^{'}.id, T^{'}.snapshot, “R” >)$
```

If at least one read-only transaction (T_{ro}) with a lesser *insertion-snapshot* is found in any snapshot-queue $SQueue$ of T's written keys, it means that T_{ro} read that key before T_w internally committed, therefore a write-after-read dependency between T_{ro} and T_w is established. In this case, T is inserted into $SQueue$ until T_{ro} returns to its client. With the anti-dependency, the transaction serialization order has been established with T_{ro} preceding

T_w. Informing immediately T_w's client about T_w's completion would expose an external order where T_w is before T_{ro}, which might violates external consistency if another non-conflicting update transaction T'_w is observed by a conflicting read-only transaction T'_{ro} in a different serialization order (e.g., the case in Figure 4.2).

Tracking only non-transitive anti-dependencies is not enough to preserve correctness. If T reads the update done by $T_{w'}$ and $T_{w'}$ is still in its Pre-commit phase, then T has a transitive anti-dependency with $T_{ro'}$ (i.e., $T_{ro'} \xrightarrow{\text{rw}} T_{w'} \xrightarrow{\text{wr}} T$). SSS records the existence of transactions like $T_{ro'}$ during T's execution by looking into the snapshot-queues of T's read keys and logging them into a private buffer of T, called $T.PropagatedSet$. The propagation of anti-dependency happens during T's Pre-commit phase by inserting transactions in $T.PropagatedSet$ into the snapshot-queues of all T's written keys (Lines 4-5 of Algorithm 3).

External Commit. Transaction T remains in its Pre-commit phase until there is no read-only transaction with lesser insertion-snapshot in the snapshot-queues of T's written keys. After that, T is removed from these snapshot-queues and an `Ack` message to the transaction 2PC coordinator is sent (Lines 1-5 of Algorithm 4).

Algorithm 4 End Pre-commit of Transaction T in node N_i using SSS

```
1: for all ($k \in T.ws$) do
2:     if $N_i \in replicas(k)$ then
   // T waits for anti-dependent transactions to be removed from snapshot-queues of T.ws.
3:         wait until ($\exists < T'.id, T'.snapshot, - >$:
           $k.SQueue.contains(< T'.id, T'.snapshot, - >) \wedge$
           $T'.snapshot < T.commitVC[i]$)
4:         $k.SQueue.remove(< T.id, T.commitVC[i], \text{“W”} >)$
5:         send $Ack$ $[T, vc[i]]$ to $T.coordinator$
```

The coordinator can inform its client after receiving `Ack` from all 2PC participants. At this stage, update transaction's external schedule is established, therefore we say that SSS commits the update transaction *externally*.

4.2.2 Execution of Read-Only Transactions

In its first read operation (Algorithm 5 Lines 4-5), a read-only transaction T on N_i assigns `NLog.mostRecentVC` to its vector clock ($T.VC$). This way, T will be able to see the latest updated versions committed on N_i. Read operations are implemented by contacting all nodes that replicate the requested key and waiting for the fastest to answer.

When a read request of T returns from node N_j, T sets $T.hasRead[j]$ to `true`. With that, we set the visibility upper bound for T from N_j (i.e., $T.VC[j]$). Hence, subsequent read operations by T contacting a node N_k should only consider versions with a vector clock vc_k such that $vc_k[j] < T.VC[j]$. After a read operation returns, the transaction vector clock is updated by applying an entry-wise maximum operation between the current $T.VC$ and the vector clock associated with the read version (i.e., VC^*) from N_j. Finally, the read value is added to $T.rs$ and returned.

Algorithm 5 Read Operation by Transaction T in node N_i using SSS

```
1:  upon Value Read (Transaction T, Key k) do
2:      if ($\exists < k, val > \in T.ws$) then
3:          return val
    // T's vector clock is initialized with the latest committed vector clock in $N_i$
4:      if ($is\ first\ read\ of\ T$) then
5:          $T.VC \leftarrow NLog.mostRecentVC$
6:      target ← {replicas(k)}
    // isUpdate is a boolean showing whether T is read-only or update
7:      send $READREQUEST[k, T.VC, T.hasRead, T.isUpdate]$ to all $N_j \in target$
8:      wait receive $READRETURN\ [val, VC^*, PropagatedSet]$ from $N_h \in target$
9:      $T.hasRead[h] \leftarrow true$
10:     $T.VC \leftarrow \max(T.VC, VC^*)$
11:     $T.rs \leftarrow T.rs \cup \{< k, val >\}$
12:     $T.PropagatedSet \leftarrow T.PropagatedSet \cup PropagatedSet$
13:     return val
14: end
```

Algorithm 6 shows SSS rules to select the version to be returned upon a read operation that contacts node N_i. The first time N_i receives a read from T, this request should wait until the value of N_i's `NLog.mostRecentVC[i]` is equal to $T.VC[i]$ (Line 5 Algorithm 6). This means that all transactions that are already included in the current visibility bound of $T.VC[i]$ must perform their internal commit before T's read request can be handled. After that, a correct version of the requested key should be selected for reading. This process starts by identifying the set of versions that are within the visibility bound of T, called $VisibleSet$. This means that, given a version v with commit vector clock vc, v is visible by T if, for each entry k such that $T.hasRead[k] = true$, we have that $vc[k] \leq T.VC[k]$

(Algorithm 6 Line 6).

It is possible that transactions associated with some of these vector clocks are still in their Pre-Commit phase, meaning they exist in the snapshot-queues of T's requested key. If so, they should be excluded from $VisibleSet$ in case their insertion-snapshot is higher than $T.VC[i]$. The last step is needed to serialize read-only transactions with anti-dependency relations before conflicting update transactions.

Algorithm 6 Version Selection Logic in node N_i using SSS

```
1:  upon receive $READREQUEST[T, k, T.VC, hasRead, isUpdate]$ from $N_j$ do
2:      $PropagatedSet \leftarrow \phi$
3:      if $(\neg isUpdate)$ then
4:          if $(\neg hasRead[i])$ then
5:              wait until $NLog.mostRecentVC[i] \geq T.VC[i]$
6:              $VisibleSet \leftarrow \{vc : vc \in NLog \wedge \forall w(hasRead[w] \Rightarrow vc[w] \leq T.VC[w])\}$
7:              $ExcludedSet \leftarrow \{T^{'} :< T^{'}.id, T^{'}.snapshot,$ “W” $>\in$
            $k.SQueue \Rightarrow T^{'}.snapshot > T.VC[i])\}$
8:              $VisibleSet \leftarrow VisibleSet \backslash ExcludedSet$
9:              $maxVC \leftarrow vc : \forall w, vc[w] = \max\{v[w] : v \in VisibleSet\}$
10:             $k.SQueue.insert(< T.id, maxVC[i],$ “R” $>)$
11:             $ver \leftarrow k.last$
12:             while $(\exists w : hasRead[w] \wedge ver.vc[w] > maxVC[w] \vee \exists vc \in ExcludedSet :$
                 $ver.vc = vc \wedge vc[i] > maxVC[i])$ do
13:                 $ver \leftarrow ver.prev$
14:         else
15:             $maxVC \leftarrow T.VC$
16:             $k.SQueue.insert(< T.id, maxVC[i],$ “R” $>)$
17:             $ver \leftarrow k.last$
18:             while $(\exists w :$ (hasRead[w] $\wedge$ $ver.vc$[w] $> maxVC[w]))$ do
19:                 $ver \leftarrow ver.prev$
20:     else
21:         $maxVC \leftarrow NLog.mostRecentVC$
22:         $PropagatedSet$=$\{T^{'}$:<$T^{'}.id$,$T^{'}.snapshot$,“R”>$\in k.SQueue\}$
23:         $ver \leftarrow k.last$
24:     send $READRETURN$[ver.val, maxVC,PropagatedSet ] to $N_j$
25: end
```

This condition is particularly important to prevent a well-known anomaly, firstly observed by Adya in [36], in which read-only transactions executing on different nodes can observe two non-conflicting update transactions in different serialization order [50]. Consider a distributed system where nodes do not have access to a single point of synchronization (or an ordering component), concurrent non-conflicting transactions executing on different nodes cannot be aware of each other's execution. Because of that, different read-only transactions might order these non-conflicting transactions in a different way, therefore breaking the client's perceived order. SSS prevents that by serializing both these read-only transactions before those update transactions.

At this stage, if multiple versions are still included in $VisibleSet$, the version with the maximum $VC[i]$ should be selected to ensure external consistency. Once the version to be returned is selected, T is inserted in the snapshot-queue of the read key using $maxVC[i]$ as insertion-snapshot (Line 10 of Algorithm 6). Finally, when the read response is received, the maximum per entry between $maxVC$ (i.e., VC^* in Algorithm 5) and the $T.VC$ is computed along with the result of the read operation.

When a read-only transaction T commits, it immediately replies to its client. After that, it sends a message to the nodes storing only the read keys in order to notify its completion. We name this message `Remove`. Upon receiving `Remove`, the read-only transaction is deleted from all the snapshot-queues associated with the read keys. Deleting a read-only transaction from a snapshot-queue enables conflicting update transactions to be externally committed and their responses to be released to their clients.

Because of transitive anti-dependency relations, a node might need to forward the `Remove` message to other nodes as follows. Let us assume T has an anti-dependency with a transaction T_w, and another transaction $T_{w'}$ reads from T_w. Because anti-dependency relations are propagated along the chain of conflicting transactions, T exists in the snapshot-queues of T_w's and $T_{w'}$'s written keys. Therefore, upon `Remove` of T, the node executing T_w is responsible to forward the `Remove` message to the node where $T_{w'}$ executes for updating the affected snapshot-queues.

When a read operation is handled by a node that already responded to a previous read operation from the same transaction, the latest version according to $maxVC$ is returned, and T can be inserted into the snapshot-queue with its corresponding identifier and $maxVC[i]$ as insertion-snapshot.

4.2.3 Examples

External Consistency and Anti-dependency

Figure 4.1 shows an example of how SSS serializes an update transaction T_1 in the presence of a concurrent read-only transaction T_2. Two nodes are deployed, N_1 and N_2, and no replication is used for simplicity. T_1 executes on N_1 and T_2 on N_2. Key y is stored in N_2's repository. The `NLog.mostRecentVC` for Node 1 is {5,4} and for Node 2 is {3,7}.

T_1 performs a read operation on key y by sending a remote read request to N_2. At this point, T_1 is inserted in the snapshot-queue of y ($Q(y)$) with 7 as insertion-snapshot. This value is the second entry of N_2's `NLog.mostRecentVC`. Then the update transaction T_2 begins with vector clock {3,7}, buffers its write on key y in its write-set, and performs its internal commit by making the new version of y available, and by inserting the produced commit vector clock (i.e., $T2.commitVC$={3,8}) in N_2's $NLog$. As a consequence of that, $NLog.mostRecentVC$ is equal to $T2.commitVC$.

Now T_2 is evaluated to decide whether it should be inserted into $Q(y)$. The insertion-snapshot of T_2 is equal to 8, which is higher than T_1's insertion-snapshot in $Q(y)$. For this reason, T_2 is inserted in $Q(y)$ and its Pre-commit phase starts.

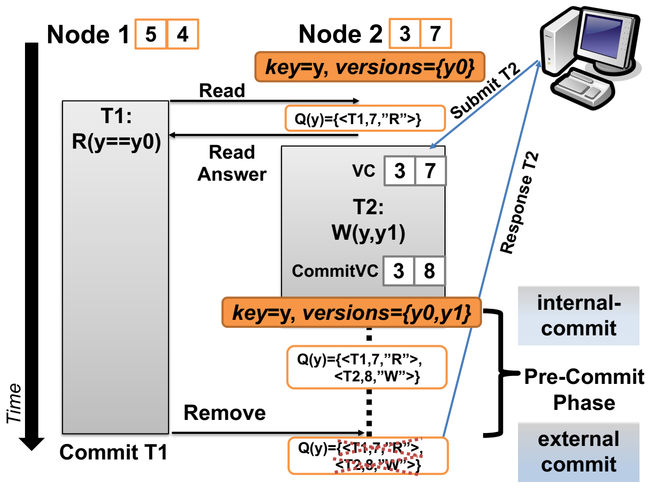


Figure 4.1: SSS execution in the presence of an anti-dependency. Orange boxes show the content of the data store. Gray boxes show transaction execution. Dashed line represents the waiting time for $T2$. The red crossed entries of Q(y) represent their elimination upon Remove.

At this stage, T_2 is still not externally visible. Hence T_2 remains in its Pre-Commit phase until T_1 is removed from $Q(y)$, which happens when T_1 commits and sends the `Remove` message to N_2. After that, T_2's client is informed about T_2's completion. Delaying the external commit of T_2 prevents clients from observing the internal completion of T_2, until T_1 returns to its client.

External Consistency and Non-conflicting transactions

Figure 4.2 shows how SSS builds the external schedule in the presence of read-only transactions and non-conflicting update transactions. There are four nodes, N_1, N_2, N_3, N_4, and four concurrent transactions, T_1, T_2, T_3, T_4, each executes on the respective node. By assumption, T_2 and T_3 are non-conflicting update transactions, while T_1 and T_4 are read-only.

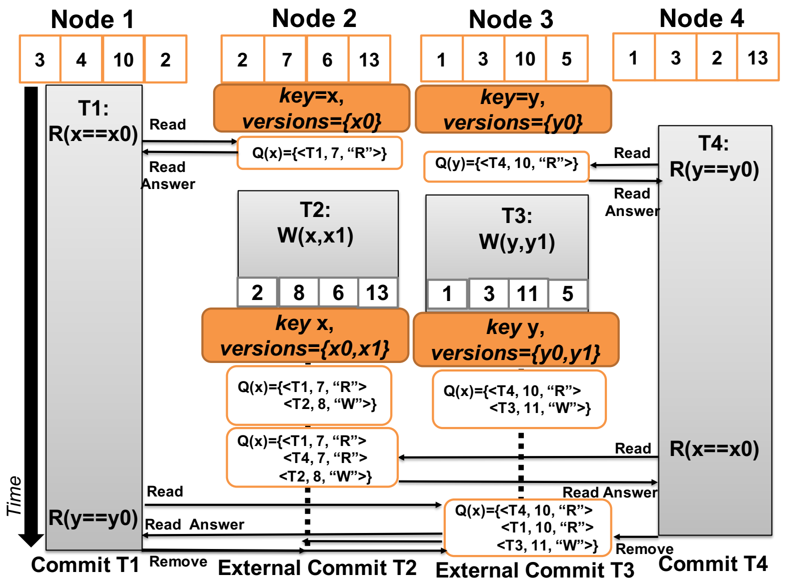


Figure 4.2: Handling read-only transactions along with non-conflicting update transactions. We omitted snapshot-queue entries elimination upon Remove to improve readability.

SSS ensures that T_1 and T_4 do not serialize T_2 and T_3 in different orders and they return to their clients in the same way they are serialized by relying on snapshot-queuing. T_1 is inserted into $Q(x)$ with insertion-snapshot equals to 7. Concurrently, T_4 is added to the snapshot-queue of y with insertion-snapshot equals to 10. The next read operation by T_1 on y has two versions evaluated to be returned: $y0$ and $y1$. Although $y1$ is the most recent, since T_4 returned $y0$ previously (in fact T_4 is in $Q(y)$), $y1$ is excluded and $y0$ is returned. Similar arguments apply to T_4's read operation on x. The established external schedule serializes T_1 and T_4 before both T_2 and T_3.

4.3 Additional Considerations of SSS

4.3.1 Garbage Collection

A positive side effect of the `Remove` message is the implicit garbage collection of entries in the snapshot-queues. In fact, SSS removes any entry representing transactions waiting for a read-only transaction to finish upon receiving `Remove`, which cleans up the snapshot-queues.

4.3.2 Starvation

Another important aspect of SSS is the chance to slow down update transactions, possibly forever, due to an infinite chain of conflicting read-only transactions issued concurrently. We handle this corner case by applying admission control to read operations of read-only transactions in case they access a key written by a transaction that is in a snapshot-queue for a pre-determined time. In practice, if such a case happens, we apply an artificial delay to the read operation (exponential back-off) to give additional time to update transaction to be removed from the snapshot-queue. In the experiments we never experienced starvation scenarios, even with long read-only transactions.

4.3.3 Deadlock-Freedom

SSS uses timeout to prevent deadlock during the commit phase's lock acquisition. Also, the waiting condition applied to update transactions cannot generate deadlock. This is because read-only transactions never wait for each other, and there is no condition in the protocol where an update transaction blocks a read-only transaction. The only wait condition occurs when read-only transactions force update transactions to hold their client response due to snapshot-queuing. As a result, no circular dependency can be formed, thus SSS cannot encounter deadlock.

4.3.4 Fault Tolerance

SSS deploys a protocol that tolerates failures in the system using replication. In the presented version of the SSS protocol, we did not include either logging of messages to recover update transactions' 2PC upon faults, or a consensus-based approach (e.g., Paxos-Commit [85]) to distribute and order 2PC messages. Solutions to make 2PC recoverable are well-studied. To focus on the performance implications of the distributed concurrency control of SSS and all its competitors, operations to recover upon a crash of a node involved in a 2PC have been disabled. This decision has no correctness implication.

4.4 Correctness Arguments

Our target is proving that every history H executed by SSS, which includes committed update transactions and read-only transactions (committed or not), is external consistent.

We adopt the classical definition of history [36]. For understanding correctness, it is sufficient to know that a history is external consistent if the transactions in the history return the same values and leave the data store in the same state as they were executed in a sequential order (one after the other), and that order does not contradict the order perceived by clients, namely the precedence relations between non-concurrent transactions as observed by clients (similar to the real-time order relations [89] in strict serializability).

We decompose SSS's correctness in three statements, each highlighting a property guaranteed by SSS. Each statement claims that a specific history H', which is derived from H, is external consistent. In order to prove that, we rely on the characteristics of the Direct Serialization Graph (DSG) [36] which is derived from H'. Note that DSG also includes order relations between transactions' external commit. Every transaction in H' is a node of the DSG graph, and every dependency of a transaction T_j on a transaction T_i in H' is an edge from T_i to T_j in the graph. The concept of dependency is the one that is widely adopted in the literature: *i)* T_j read-depends on T_i if a read of T_j returns a value written by T_i, *ii)* T_j write-depends on T_i if a write of T_j overwrites a value written by T_i; *iii)* T_j anti-depends on T_i if a write of T_j overwrites a value previously read by T_i. We also map transactions relations as observed by clients to edges in the graph: if T_i commits externally before T_j

starts, then the graph has an edge from T_i to T_j. A history H' is external consistent iff the DSG does not have any cycle [36, 37].

In our proofs we use the binary relation $\leq$ to define an ordering on pair of vector clocks v_1 and v_2 as follows: $v_1 \leq v_2$ if $\forall i$, $v_1[i] \leq v_2[i]$. Furthermore, if there also exists at least one index j such that $v_1[j] < v_2[j]$, then $v_1 < v_2$ holds.

Statement 1. For each history H executed by SSS, the history H', which is derived from H by only including committed update transactions in H, is external consistent.

Proof. In the proof we show that if there is an edge from transaction T_i to transaction T_j in DSG, then $T_i.commitVC < T_j.commitVC$. This statement implies that transactions modify the state of the data store as they were executed in a specific sequential order (provided by *CommitQ*), which does not contradict the transaction external commit order. Because no read-only transaction is included in H', the internal commit is equivalent to the external commit (i.e., no transaction is delayed). The formal proof is included in the technical report [91].

Statement 2. For each history H executed by SSS, the history H', which is derived from H by only including committed update transactions and one read-only transaction in H, is external consistent.

Proof. The proof shows that a read-only transaction always observes a consistent state by showing that in both the case of a direct dependency or anti-dependency, the vector clock of the read-only transaction is comparable with the vector clocks of conflicting update transactions. This statement implies that read operations of a read-only transaction always return values from a state of the data store as the transaction was executed atomically in a point in time that is not concurrent with any conflicting update transaction. The formal proof is included in the technical report [91].

Statement 3. For each history H executed by SSS, the history H', which is derived from H by including committed update transactions and two or more read-only transactions in

H, is external consistent.

Proof. Since Statement 2 holds, SSS guarantees that each read-only transaction appears as it were executed atomically in a point in time that is not concurrent with any conflicting update. Furthermore, since Statement 1 holds, the read operations of that transaction return values of a state that is the result of a sequence of committed update transactions. Therefore, Statement 3 implies that, given such a sequence $S1$ for a read-only transaction T_{r1}, and $S2$ for a read-only transaction T_{r2}, either $S1$ is a prefix of $S2$, or $S2$ is a prefix of $S1$. In practice, this means that all read-only transactions have a coherent view of all transactions executed on the system. The formal proof is included in the technical report [91].

4.5 Evaluation Study

We implemented SSS in Java from the ground up and performed a comprehensive evaluation study. In the software architecture of SSS there is an optimized network component where multiple network queues, each for a different message type, are deployed. This way, we can assign priorities to different messages and avoid protocol slow down in some critical steps due to network congestion caused by lower priority messages (e.g., the `Remove` message has a very high priority because it enables external commits). Another important implementation aspect is related to snapshot-queues. Each snapshot-queue is divided into two: one for read-only transactions and one for update transactions. This way, when the percentage of read-only transactions is higher than update transactions, a read operation should traverse few entries in order to establish its visible-set.

We compare SSS against the following competitors: 2PC-baseline (2PC in the plots), ROCOCO [30], and Walter [21]. All these competitors offer transactional semantics over key-value APIs. With 2PC-baseline we mean the following implementation: all transactions execute as SSS's update transactions; read-only transactions validate their execution, therefore they can abort; and no multi-version data repository is deployed. As SSS, 2PC-baseline guarantees external consistency.

ROCOCO is an external consistent two-round protocol where transactions are divided into pieces and dependencies are collected to establish the execution order. ROCOCO classifies pieces of update transactions into immediate and deferrable. The latter are more efficient because they can be reordered. Read-only transactions can be aborted, and they are implemented by waiting for conflicting transactions to complete. Our benchmark is configured in a way all pieces are deferrable. ROCOCO uses preferred nodes to process transactions and consensus to implement replication. Such a scheme is different from SSS where multiple nodes are involved in the transaction commit process. To address this discrepancy, in the experiments where we compare SSS and ROCOCO, we disable replication for a fair comparison. The third competitor is Walter, which provides PSI a weaker isolation level than SSS. Walter has been included because it synchronizes nodes using vector clocks, as done by SSS.

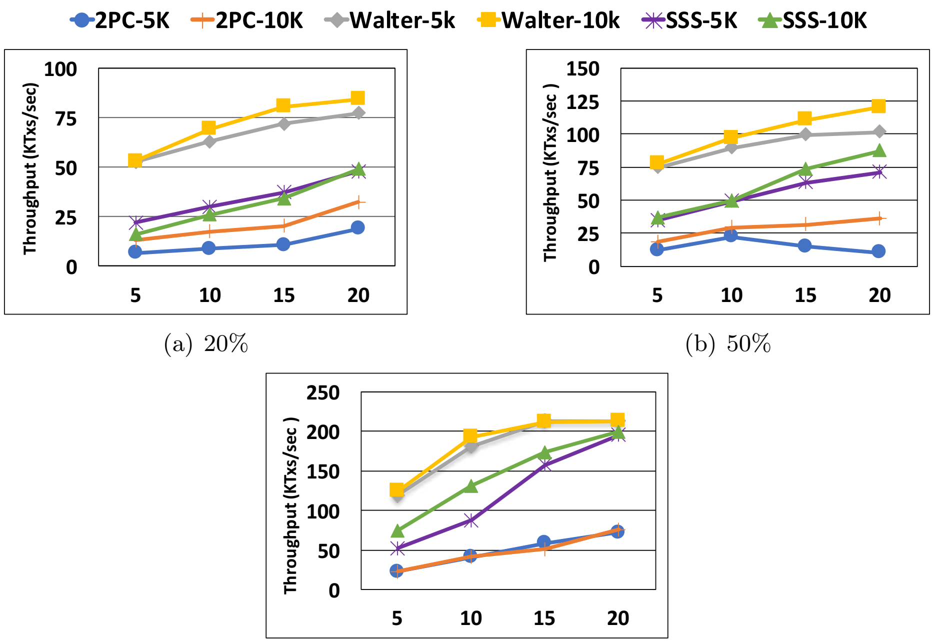
All competitors have been re-implemented using the same software infrastructure of SSS because we want to provide all competitors with the same underlying code structure and optimization (e.g., optimized network). For fairness, we made sure that the performance obtained by our re-implementation of competitors matches the trends reported in [21] and [30], when similar configurations were used.

In our evaluation we use YCSB [92] benchmark ported to key-value store. We configure the benchmark to explore multiple scenarios. We have two transaction profiles: update, where two keys are read and written, and read-only transactions, where two or more keys are accessed. In all the experiments we co-locate application clients with processing nodes, therefore increasing the number of nodes in the system also increases the amount of issued requests. There are 10 application threads (i.e., clients) per node injecting transactions in the system in a closed-loop (i.e., a client issues a new request only when the previous one has returned). All the showed results are the average of 5 trials.

We selected two configurations for the total number of shared keys: 5k and 10k. We selected these ranges since they give us the appropriate level of contention on snapshot-queues in the case of 20% read-only transactions (write-dominated work load) and 80% read-only transactions (read-dominated work load). With the former, the observed average transaction abort rate is in the range of 6% to 28% moving from 5 nodes to 20 nodes when 20% read-only transactions are deployed. In the latter, the abort rate was from 4% to

14%. Unless otherwise stated, transactions select accessed objects randomly with uniform distribution.

As test-bed, we used CloudLab [93], a cloud infrastructure available to researchers. We selected 20 nodes of type c6320 available in the Clemson cluster [94]. This type is a physical machine with 28 Intel Haswell CPU-cores and 256GB of RAM. Nodes are interconnected using 40Gb/s Infiniband HPC cards. In such a cluster, a network message is delivered in around 20 microseconds (without network saturation), therefore we set timeout on lock acquisition to 1ms.



(a) 20%

(b) 50%

(c) 80%

Figure 4.3: Throughput of SSS against 2PC-baseline and Walter, varying % of read-only transactions. Number of nodes in X-axes.

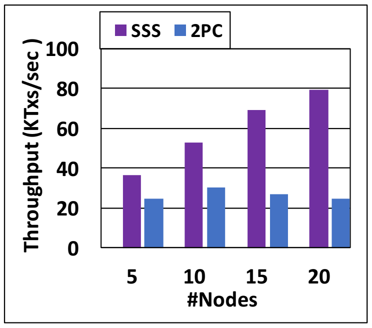
In Figure 4.3 we compare the throughput of SSS against 2PC-baseline and Walter in the case where each object is replicated in two nodes of the system. We also varied the percentage of read-only transactions in the range of 20%, 50%, and 80%. As expected, Walter is the leading competitor in all the scenarios because its consistency guarantee is much weaker than external consistency; however, the gap between SSS and Walter reduces from 2× to 1.1× when read-only transactions become predominant (moving from Figure 4.3(a) to 4.3(c)). This is reasonable because in Walter, update transactions do not have the same impact in read-only transactions' performance as in SSS due to the presence of the snapshot-queues. Therefore, when the percentage of update transactions reduces, SSS reduces the gap. Considering the significant correctness level between PSI (in Walter) and external consistency, we consider the results of the comparison between SSS and Walter remarkable.

Performance of 2PC-baseline is competitive when compared with SSS only at the case of 20% read-only. In the other cases, although SSS requires a more complex logic to execute its read operations, the capability of being abort-free allows SSS to outperform 2PC-baseline by as much as 7× with 50% read-only and 20 nodes. 2PC-baseline's performance in both the tested contention levels become similar at the 80% read-only case because, although lock-based, read-only transaction's validation will likely succeed since few update transactions execute in the system.

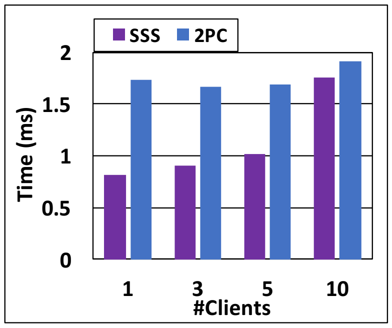
Figure 4.3 also shows the scalability of all competitors. 2PC-baseline suffers from higher abort rate than others, which hampers its scalability. This is because its read-only transactions are not abort-free. The scalability trend of SSS and Walter is similar, although Walter stops scaling at 15 nodes using 80% of read-only transactions while SSS proceeds. This is mostly related with network congestion, which is reached by Walter earlier than SSS since Walter's transaction processing time is lower than SSS, thus messages are sent with a higher rate.

In Figure 4.4 we compare 2PC-baseline and SSS in terms of maximum attainable throughput and transaction latency. Figure 4.4(a) shows 2PC-baseline and SSS configured in a way they can reach their maximum throughput with 50% read-only workload and 5k objects, meaning the number of clients per nodes differs per reported datapoint. Performance trends are similar to those in Figure 4.3(b), but 2PC-baseline here is faster than before. This is related with the CPU utilization of the nodes' test-bed. In fact, 2PC-baseline requires less threads to execute, meaning it leaves more unused CPU-cores than SSS, and those CPU-cores can be leveraged to host more clients.

The second plot (Figure 4.4(b)) shows transaction latency from its begin to its external commit when 20 nodes, 50% read-only transactions, and 5k objects are deployed. In the experiments we varied the number of clients per node from 1 to 10. When the system is far from reaching saturation (i.e., from 1 to 5 clients), SSS's latency does not vary, and it is on



(a) Max attainable throughput.



(b) External Commit latency.

Figure 4.4: Performance of SSS against 2PC-baseline using 5k objects and 50% read-only transactions.

average 2× lower than 2PC-baseline's latency. At 10 clients, SSS's latency is still lower than 2PC-baseline but by a lesser percentage. This confirms one of our claim about SSS capability of retaining high-throughput even when update transactions are held in snapshot-queues. In fact, Figure 4.3(b) shows the throughput measurement in the same configuration: SSS is almost 7× faster than 2PC-baseline.

Figure 4.5 shows the relation between the internal commit latency and the external commit latency of SSS update transactions. The configuration is the one in Figure 4.4(b). Each bar represents the latency between a transaction begin and its external commit. The internal gray bar shows the time interval between the transaction's insertion in a snapshot-queue and its removal (i.e., from internal to external commit). This latter time is on average 30% of the total transaction latency.

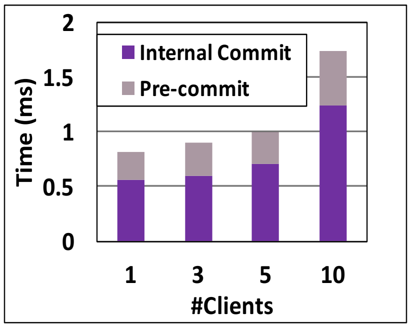
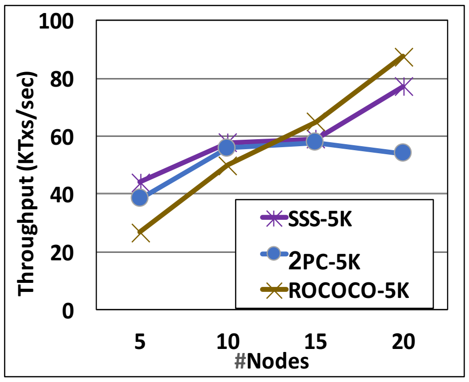


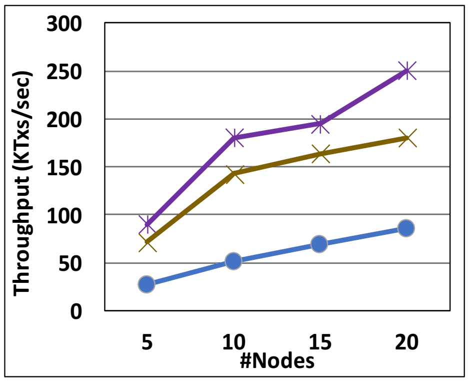
Figure 4.5: Breakdown of SSS transaction latency.

In Figure 4.6 we compare SSS against ROCOCO and 2PC-baseline. To be compliant with ROCOCO, we disable replication for all competitors and we select 5k as total number of shared keys because ROCOCO finds its sweet spot in the presence of contention. Accesses are not local.

Figures 4.6(a) and 4.6(b) show the results with 20% and 80% read-only transactions respectively. In write intensive workload, ROCOCO slightly outperforms SSS due to its lock-free executions and its capability of re-ordering deferrable transaction pieces. However, even in this configuration, which matches a favorable scenario for ROCOCO, SSS is only 13% slower than ROCOCO and 70% faster than 2PC-baseline. In read-intensive workload, SSS outperforms ROCOCO by 40% and by almost 3× 2PC-baseline at 20 nodes. This gain is because ROCOCO is not optimized for read-only transactions; in fact, its read-only are not abort-free and they need to wait for all conflicting update transactions in order to execute. Also, since in YCBS transaction size is small, the overhead of ROCOCO's two-round commitment protocol is dominant.



(a) 20%.



(b) 80%.

Figure 4.6: SSS, 2PC-baseline, ROCOCO varying % of read-only transactions. Legend in (a) applies to (b).

We also configured the benchmark to produce 50% of keys access locality, meaning the probability that a key is stored by the node where the transaction is executing (local node), and 50% of uniform access. Increasing local accesses has a direct impact on the application contention level. In fact, since each key is replicated on two nodes, remote communication is still needed by update transactions, while the number of objects accessible by a client reduces when the number of nodes increases (e.g., with 20 nodes and 5k keys, a client on a node can select its accessed keys among 250 keys rather than 5k). Read-only transactions are the ones that benefit the most from local accesses.

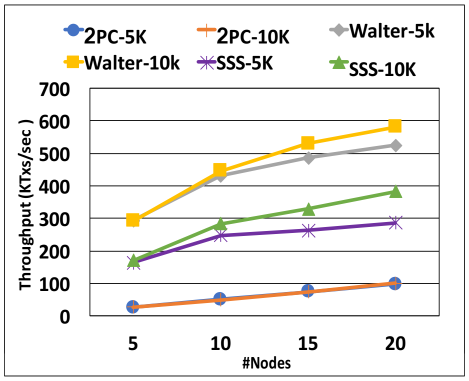


Figure 4.7: Throughput of SSS against 2PC-baseline and Walter with 80% read-only transactions and 50% locality.

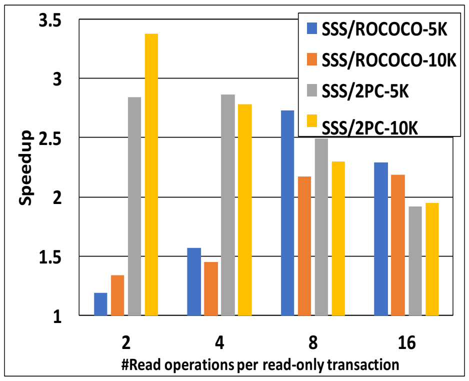


Figure 4.8: Speedup of SSS over ROCOCO and 2PC-baseline increasing the size of read-only transactions.

We report the results (in Figure 4.7) using the same configuration in Figure 4.3(c) because that is the most relevant to SSS and Walter. Results confirm similar trend. SSS is more than 3.5× faster than 2PC-baseline but, as opposed to the non-local case, here it cannot close the gap with Walter due to the high contention around snapshot-queues.

In Figure 4.8 we show the impact of increasing the number of read operations inside read-only transactions from 2 to 16. For this experiment we used 15 nodes and 80% of read-only workload. Results report the ratio between the throughput of SSS and both ROCOCO and 2PC-baseline. When compared to ROCOCO, SSS shows a growing speedup, moving from 1.2× with 2 read operations to 2.2× with 16 read operations. This is because, as stated previously, ROCOCO encounters a growing number of aborts for read-only transactions while increasing accessed objects. 2PC-baseline degrades less than ROCOCO when operations increases because it needs less network communications for read-only transactions.

Chapter 5

FPSI: Improving Read Guarantees in Parallel Snapshot Isolation

5.1 Overview

In a centralized deployment of SI [43, 95], the reading snapshot is often determined by assuming that time is measured by a shared atomic counter that advances whenever any transaction starts or commits [96]. On the other hand, in distributed systems where nodes do not share a synchronized clock and the communication among them is asynchronous, SI transactions cannot simply define an up-to-dated reading snapshot at the time they start because of the absence of a shared notion of time among nodes.

Walter [21] is a state-of-the-art distributed transactional system whose concurrency control implements a relaxed variant of SI called Parallel Snapshot Isolation (or PSI). In PSI, the transaction reading snapshot can be arbitrarily outdated in order to deal with the aforementioned absence of shared clocks among nodes (other relaxations are overviewed in Chapter 3).

Walter logically assigns objects to so called *preferred nodes*. A preferred node always stores the latest version of an object. The object might also be replicated on other *non-preferred* nodes, which might not always have the latest version of objects. If a transaction begins on a node N and reads an object whose preferred node is N (we name such a transaction *local*), then its reading snapshot is guaranteed to be up-to-date. Otherwise, when a transaction begins on a non-preferred node or any other node (for brevity, in both these cases we refer to this transaction as *non-local*), the read operations can result in an *outdated* object version.

Walter attempts to patch the above issue by using asynchronous messages, sent outside the transaction critical path, aimed at periodically updating the logical clock of other nodes, including the non-preferred ones. However, until asynchronous messages are received, non-local read-only transactions can still return arbitrarily old versions. Another side effect of this solution is that non-local update transactions will be repeatedly aborted until the above asynchronous messages are delivered.

In this chapter, we present *Fresher Parallel Snapshot Isolation* (or FPSI), a distributed concurrency control that uses logical (vector) clocks [69] to implement an enhanced version of Walter's concurrency control with the goal of improving data freshness for read-only transaction. FPSI exploits the fact that a common behavior for transactions is accessing mostly local objects [21, 77]. For the remaining accesses, Walter must adhere to a possibly old reading snapshot. FPSI improves this scenario for read-only transactions. Every access to a new node made by a read-only transaction is guaranteed to observe the most recent and correct reading snapshot. The *only* case in which a read-only transaction is prevented from accessing the latest reading snapshot is when multiple accesses target objects stored on the same node.

A practical example where FPSI always returns the most recent reading snapshot is when we consider the two transaction profiles `Order-Status` and `Payment` in the TPC-C benchmark [97]. The former queries the status of a customer's last order from a warehouse to retrieve information about related order lines. The latter processes the payment for the customer and modifies the balance of the warehouse where the order took place. The read-only transaction `Order-Status` can see the latest version of the accessed objects modified by `Payment` since the first access is to retrieve the warehouse, and the subsequent read operations are on objects that have been committed along with that warehouse, regardless of the preferred node of the warehouse.

The major algorithmic challenge in achieving FPSI's goals is to deal with the (fast) technique used by Walter to update nodes' logical clocks upon transaction commits. In fact, since Walter's transaction reading snapshot can be arbitrarily old, vector clocks are updated without synchronously propagating causal dependency with other transactions.

Unlike Walter, the reading snapshot of a read-only transaction in FPSI is established during its execution by means of attempting to include the newest versions of an object stored by a node that has not been contacted so far by this transaction. FPSI ensures that by efficiently tracking some (but not all) transaction dependency relations. Update transactions execute with similar guarantees as in Walter, although FPSI still attempts to improve data freshness by deploying a technique, similar to the one used in SCORe [48], where the reading snapshot is defined upon the first read operation.

This design explores a trade-off between high performance and complexity of the distributed concurrency control. For many applications, transactions access objects mostly from preferred sites and a limited number of objects from non-preferred sites. If the latter objects are stored in disjoint nodes and transactions are read-only, FPSI provides the highest level of freshness in its accesses and retains comparable performance to Walter. FPSI also guarantees that transactions access the latest version of multiple objects if these objects are stored in the same node and updated by the same update transaction. This access pattern is typical of OLTP applications, as encapsulated by standard benchmarks such as TPC-C [97].

In Section 5.2, we study the existing challenges in the concurrency control of Walter. In Section 5.3, the FPSI's concurrency control is described. Finally, Sections 5.4 and 5.5 provide the correctness arguments and evaluation study of FPSI respectively.

5.2 Background and Motivation

5.2.1 Walter and PSI

Walter [21], is a multi-version transactional key-value store that provides a relaxed version of SI called Parallel Snapshot Isolation (PSI). Walter uses a technique named *preferred site* where each object is logically assigned to a specific site (or node) in the system. The concept of preferred site is meant to favor transactions accessing objects maintained by the local nodes. With that, Walter can quickly commit these transactions without checking other nodes for write conflicts.

In other words, if a local transaction issues an operation on object x, then it can access the latest version of x. However, non-local transactions are still allowed to modify x on N_i but their updates can be repeatedly aborted in case the accessed version of x is not the latest one.

After a local transaction commits, the acknowledgment of its successful commit should be propagated to other nodes in the system. This propagation is done asynchronously and its goal is to eventually allow non-local transactions to advance their reading snapshot. As an example of the above propagation mechanism, suppose the preferred site of object x is N_1. Local transaction T_1 starts at N_1 and creates a new version x_1 of x. A non-local transaction T_2, started at node N_2, cannot create another version of x (i.e., x_2) until N_2 is being acknowledged about the commit of T_1 in N_1. After N_2 receives the propagation message of T_1's commit, T_2 is able to proceed with its execution and successfully create x_2.

5.2.2 The Challenge of Updating Reading Snapshot in Walter

Walter does not update the reading snapshot of a transaction during its execution. This is because, by doing that without leveraging additional metadata, a well-known anomaly called Read Skew [35] might occur. Read Skew happens if a transaction T_1 reads a version x_1 for object x and concurrently another transaction T_2 commits an update on objects x and y, which creates a new version x_2 of x and y_2 of y. If T_1 reads y after committing T_2, by advancing its reading snapshot without any consideration it might return y_2, which is incorrect.

Solutions in literature, such as SSS [32] and GMU [50], overcome the Read Skew anomaly by updating vector clocks in a way that takes into account causal dependencies among nodes that have been previously contacted by a transaction. Walter prefers a simpler approach in which only the vector clock entry associated with the node where the transaction executes is updated upon commit. Such a decision is supported by the fact that read operations in Walter can read arbitrarily old values, therefore there is no need to account for causal dependency relations developed after the chosen reading snapshot. FPSI's goal is to preserve the advantage of Walter's simpler concurrency control while adding additional metadata to improve data freshness of read-only transactions.

5.2.3 The Impact of Data Freshness in the Long Fork Anomaly

Figure 5.1 shows an example of an execution accepted by Walter in which two read-only transactions are allowed to see the results of two update transactions in different orders. Although this execution is admitted by PSI (anomaly is known as long-fork [21]), it introduces an undesirable behavior at the application level, as described below.

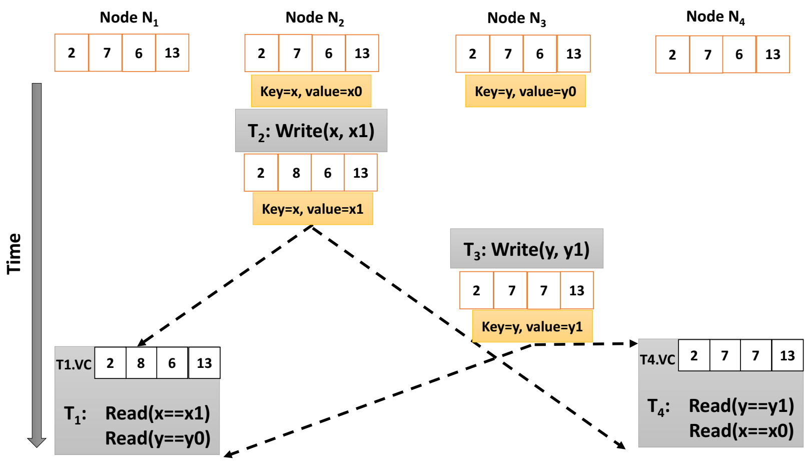


Figure 5.1: Long-fork anomaly accepted by PSI consistency level. Dashed arrows represent the asynchronous propagation messages. The reading snapshot of T_1 reflects the timestamp of T_2 in the second entry of T_1's vector clock ($T_1.VC$) but it does not reflect the timestamp of T_3 in the third entry of T_1's vector clock. The reading snapshot of T_4 reflects the timestamp of T_3 in the third entry of T_4's vector clock ($T_4.VC$) but it does not reflect the timestamp of T_2 in the third entry of T_4's vector clock.

In the example we assume four nodes, N_1, N_2, N_3, N_4, and four transactions, T_1, T_2, T_3, T_4, each begins and executes on the respective node. By assumption, T_2 and T_3 are non-conflicting local update transactions; while T_1 and T_4 are non-local read-only transactions both accessing objects from N_2 and N_3. As of Walter's rule, each read-only transaction starts its execution by acquiring the latest vector clock of the node where it executes.

Both T_2 and T_3 after their commit on their preferred sites N_2 and N_3 send a propagation message to all other nodes. Let us assume T_1 starts its execution after receiving the propagation of T_2 and before receiving the propagation of T_3. On the other hand, T_4 starts its execution after receiving the propagation of T_3 and before receiving the propagation of T_2. Receiving propagate from different nodes in different orders is a likely scenario in an asynchronous distributed system.

Since T_1 and T_4 start after the commit of T_2 and T_3, their respective clients might have had the chance to interact with each other outside the system (e.g., in a social media platform when a user publishes a new post and alerts her/his friends about the new content so that they can read it). The consequence of this interaction is that T_1's and T_4's clients will not expect to observe a snapshot in which only some of the updates that they expected to be committed are returned by their read-only transactions.

FPSI overcomes the above issue by allowing T_1 and T_4 to read the modifications made by T_2 and T_3, as long as *i)* no other reads accessing objects on N_2 and N_3 are issued by T_1 and T_4, and *ii)* T_2 and T_3 commit before T_1 and T_4 start. Note that, in the case T_1 and T_4 are concurrent with T_2 and T_3, both FPSI and PSI allow T_1 and T_4 to observe update transactions in different order, therefore long fork is still possible for FPSI as well. However, the latter case of long fork cannot trigger the behavior illustrated above at the client side, and this is again thanks to FPSI's improve data freshness.

5.3 FPSI: Protocol Description

5.3.1 Metadata

Since FPSI is built on top of Walter, we first list Walter's metadata for completeness and then we show the additional metadata required by FPSI.

Transaction vector clock. A transaction T holds a vector clock `T.VC` whose size is equal to the number of nodes in the system. `T.VC` encapsulates the knowledge of T with respect to the logical timestamps of other nodes. In practice, `T.VC` is used as visibility bound for all versions accessible by T.

Transaction write-set. Every transaction T holds a private buffer called $T.writeset$, which contains the objects the transaction wrote, along with their values.

Current sequence number. Every node N_i is assigned with a number $CurrSeqNo_i$, which represents the sequence number of the latest transaction issued and committed at node N_i.

The following metadata are exclusive for FPSI.

Transaction node access vector clock. A transaction T records the sites where it reads from in a vector clock, called `T.hasRead`. Every time T reads from a node N_j for the first time during its execution, `T.hasRead[j]` is set to `true`. When `T.hasRead[j]` is set to `true`, T's visible timsestamp with respect to N_j is fixed and cannot be advanced for T's future accesses to N_j.

Node vector clock. Each node N_i is associated with a vector clock, called $siteVC_i$. The j^{th} entry of this vector clock represents the last transaction from node N_j that was committed at site N_i.

Transaction commit vector clock. When the commit decision for transaction T issued by N_i is made, the $CurrSeqNo_i$ is incremented and $siteVC$ of N_i is updated at the i^{th} position and the updated value of $siteVC$ is assigned to transaction commit vector clock (i.e., $T.commitVC$). In addition to that, the $CurrSeqNo_i$ is also sent to the other nodes involved in the commit procedure to update their $siteVC$ at the i^{th} position.

Version's vector clock. As it is mentioned in Section 2.2, each object o is associated with a set of versions where each version v is created by an update transaction. The commit vector clock of each update transaction is assigned to its created versions and is called version vector clock ($v.VC$).

Version identifier. Each version v of object o is associated with a monotonically increasing scalar number, called $v.id$.

Version access set. As shown in Section 5.2, by relying on the way Walter establishes transactions' commit vector clocks (i.e., without tracing causal dependencies among involved nodes), advancing transaction vector clock during execution without additional metadata violates PSI.

In order to advance the reading snapshot, given a transaction T_i the concurrency control needs to be able to trace concurrent transactions T_j that overwrite versions read by T_i. In this case, we say that T_i has an anti-dependency relation (i.e., a read-after-write conflict) with T_j. FPSI does that by implementing a technique called *visible reads* [89].

The visible reads technique is implemented by FPSI in the following way. Each version is associated with a set containing identifiers of read-only transactions that read that specific version. During the commit phase of an update transaction, the set of identifiers of concurrent conflicting read-only transactions is collected. This set is propagated to the version-access-sets of the newly created versions of this update transaction since with its commit, it establishes transitive anti-dependency relations with those read-only transactions.

If a read-only transaction T contacts a node for the first time, it can advance its reading snapshot unless it finds that its own identifier exists in the version-access-set of the version to be read. In that case, T should select a previous version whose version-access-set does not contain T's identifier.

5.3.2 Transactional Begin Operation

Algorithm 7 represents the way that transaction T vector clocks ($T.hasRead$ and $T.VC$) are initialized once T begins. When T begins in node N_i, it assigns the $siteVC$ of N_i, which shows the vector clock of the latest committed/propagated transactions from all the sites in/to N_i, to its own $T.VC$. At this point, since no read is issued yet, all elements of $T.hasRead[j]$ are set to `false`.

Algorithm 7 Begin procedure of transaction T in node N_i using FPSI

```
1: function BeginTx($Transaction\ T$)
2:     $T.VC \leftarrow siteVC_i$
3:     for all ($T.hasRead[i]$) do
4:         $T.hasRead[i] \leftarrow false$
```

5.3.3 Transactional Write Operation

In FPSI update transactions implement lazy update, meaning their written keys are not immediately visible and accessible at the time of the write operation, but they are buffered in the transaction *writeset*.

5.3.4 Transactional Read Operation

Algorithm 8 describes the steps of a read operation for key k by transaction T. If k has been already written by transaction T, then the written value of k is returned (Lines 2-3 of Algorithm 8). Otherwise, a read request (`ReadRequest`) is forwarded to the node that stores k, which might be the same node where T executes (local read) (Line 5 of Algorithm 8).

The read is handled differently depending on the type of the issuing transaction. Importantly, for avoiding concurrent modifications while the read logic is processed, the read handler should be executed in mutual exclusion with respect to message handlers from other concurrent conflicting update transactions. However, read-only transactions are still allowed to operate simultaneously on read handlers.

Algorithm 8 Read Operation in FPSI

```
1:  function READ($Transaction\ T,\ key\ k$)
2:      if $< k, val > \in T.writeset$ then
3:          return $val$
4:      $target \leftarrow site(k)$
5:      send $ReadRequest[T,\ k]$ to $target$
6:      wait Receive $ReadReturn[val,\ maxVC]$ from $target$
7:      $T.hasRead[target] \leftarrow true$
8:      $T.VC \leftarrow max(T.VC, maxVC)$
9:      if (T is a read-only) then
10:         $T.readKeys \leftarrow T.readKeys \cup \{k\}$
11:     return $val$
```

Read operations by Read-only Transactions

Lines 2-9 of Algorithm 9 describes the read policy for a read-only transaction T. The first step is to identify the set of versions for k that are visible according to $T.VC$. We say a version v is *visible* for a transaction T if all the entries of $T.VC$, for which $T.hasRead$ is `true`, have values greater or equal to the values of the respective entries in $v.VC$ (Algorithm 9 Lines 4).

Algorithm 9 Version Selection Logic in node N_i using FPSI

```
1: upon receive ReadRequest[T, k] from $N_j$ do
2:     if (T is a read-only) then
3:         get lock($key = k$, $owner = T.id$)
4:         $VisibleSet \leftarrow \{v \in k.versionSet : \forall site \in sites : T.hasRead[site] = true \Rightarrow$
            $v.VC[site] \leq T.VC[site]\}$
5:         $ExcludedSet \leftarrow \{v \in VisibleSet : T.id \in v.accessSet\}$
6:         $VisibleSet \leftarrow VisibleSet \backslash ExcludedSet$
7:         $version \leftarrow ver \in VisibleSet : \forall v \in VisibleSet \Rightarrow ver.id \geq v.id$
8:         $version.accessSet \leftarrow version.accessSet \cup \{T.id\}$
9:         release lock($key = k$, $owner = T.id$)
10:    if (T is an update) then
11:        get lock($key = k$, $owner = T.id$)
12:        $VisibleSet \leftarrow \{v \in k.versionSet : \forall site \in sites : T.hasRead[site] = true \Rightarrow$
            $v.VC[site] \leq T.VC[site]\}$
13:        $ExcludedSet \leftarrow \{v \in VisibleSet : \forall site \in sites : T.hasRead[site] = true \Rightarrow$
           $v.VC[site] = T.VC[site] \wedge \exists s \in sites : T.hasRead[s] = false. \wedge v.VC[s] >$
           $T.VC[s]\}$
14:        $VisibleSet \leftarrow VisibleSet \backslash ExcludedSet$
15:        $version \leftarrow ver \in VisibleSet : \forall v \in VisibleSet \Rightarrow ver.id \geq v.id$
16:        release lock($key = k$, $owner = T.id$)
17:    send ReadReturn[version, version.VC] to $N_j$
18: end
```

From the latter set ($VisibleSet$ in the Algorithm 9), those versions whose version-access-set include T's identifier, should be excluded because that means T has already established an anti-dependency (directly or transitively) with the transactions that committed those versions. Among the remaining versions, the one with the highest identifier (meaning the freshest among them) is selected as the result of the read operation.

Figure 5.2 illustrates an example of how read-only transactions establishes their reading snapshots. Transaction T_1 starts its execution at node N_1 and reads x_0, the latest version of object x, when it accesses node N_2. Upon reading x_0, the identifier of T_1 is inserted into the corresponding version-access-set of x_0. T_1 also updates $T_1.VC[2]$ to the latest timestamp of N_2 which is "7". After that, a concurrent update transaction T_3 commits an update on x and y on N_2 and increments $siteVC_2[3]$ to timestamp "7". Later, after T_3 commits at N_2, T_1 issues another read on y. At this point, since y_1's version-access-set includes T_1's identifier, because it has been inserted by the commit procedure of T_3 (see Section 5.3.5), y_1 cannot be returned by T_1's read operation due to the anti-dependency relation already established between T_1 and T_3. After committing, a `Remove` message to N_2 is sent for notifying the completion of T_1 (see Section 5.3.6).

Note that a read-only transaction should record the accessed keys in a set, called `readkeys`, that is used only to dispatch `Remove` messages.

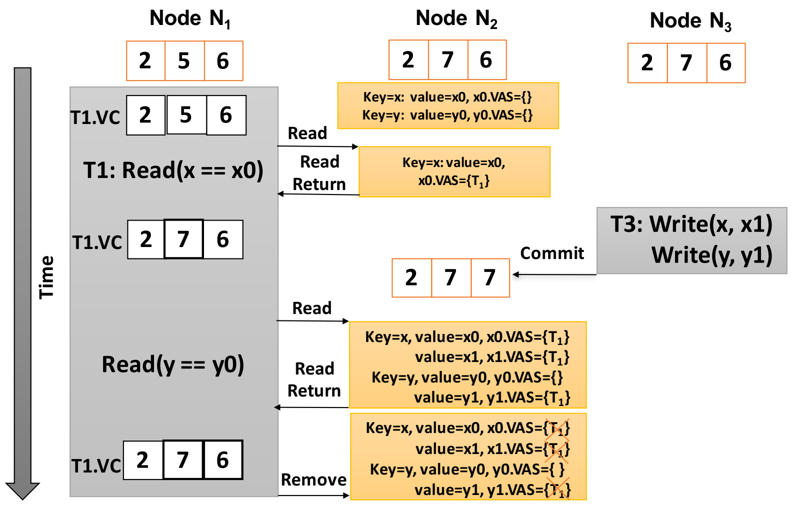


Figure 5.2: Example of execution where a read-only transaction advances its reading snapshot and still reads consistently. VAS is the version-access-set. Bold vector clock entries show where *hasRead* is `true`. The red crossed entries of VAS represent their elimination upon Remove.

Read operations by Update Transactions

Update transactions do not insert their identifier in the version-access-set of their read keys. However, upon their first read operation, they still advance their reading snapshot to be able to observe the accessed object. Subsequent read operations will use the same established reading snapshot without updating it.

Lines 10-16 of Algorithm 9 show the pseudo code for handling read operations by an update transaction T. The $VisibleSet$ is determined as follows. First, the versions that are visible according to $T.VC$ are selected. From them, the versions produced by concurrent transactions with anti-dependency with T should be excluded. However, since the version-access-set cannot be leveraged to precisely identify anti-dependency relations, as the case of read-only transactions, we adopt a more conservative condition for version exclusion, inspired by [48], which over-approximates the existence of an anti-dependency by just comparing T's vector clock against the candidate version's commit vector clock.

A version should be excluded if it has a vector clock in which, in all the positions where $T.hasRead$ is `true`, the value is equal to the value of the same entry in $T.VC$ (Lines 12-14 of Algorithm 9) and there exist at least one position in $T.VC$ whose corresponding entry in $T.hasRead$ is `false` and in the same position the version vector clock has a greater value than $T.VC$. The latter clause of the above condition allows an update transaction to also exclude a version committed by a concurrent transaction, or a transaction whose acknowledgment has not been received yet, without an anti-dependency with T, which is a false positive case since that version could be read without compromising PSI.

After that, the version with the highest identifier in the resulting $VisibleSet$ is returned as the result of the read operation (Line 15 of Algorithm 9), along with its vector clock.

When the response for T's read operation is returned to N_i, an entry-wise maximum between $T.VC$ and the version vector clock is performed to advance the reading snapshot of T (Line 8 of Algorithm 8).

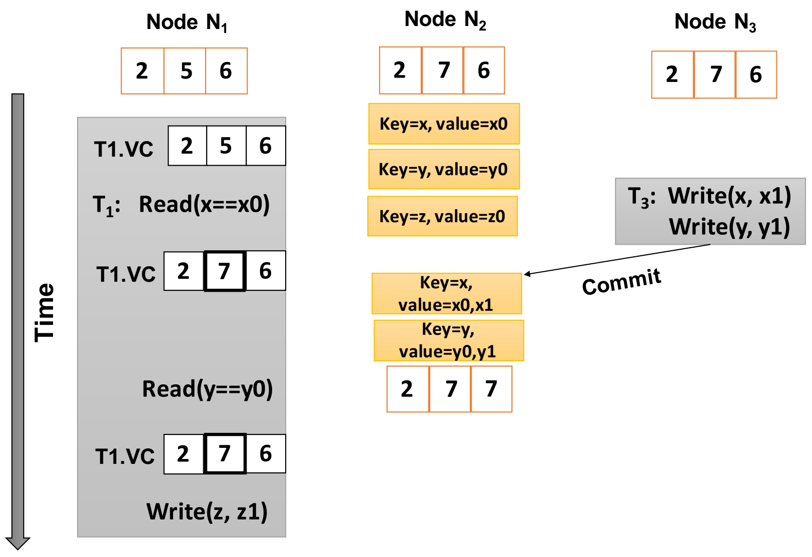


Figure 5.3: Example showing how an update transaction establishes its reading snapshot.

Figure 5.3 shows an example of how update transactions establish their reading snapshot. We have two update transactions T_1 and T_3. T_1 reads x_0, the latest version of object x at its first access to node N_2 and advances its reading snapshot by updating the second entry of $T_1.VC$ to "7". Concurrently T_3 updates both objects x and y, stored on N_2, and commits by advancing N_2's vector clock at its third entry to "7".

After that T_1 performs its second read operation on y. Here, T_1 cannot read version y_1. This is because $T_1.VC[2]$ is equal to $y1.VC[2]$ and $T_1.hasRead[2]$ is `true`. In this case, since $T_1.VC[3]$ is less than $y1.VC[3]$, it might mean that y_1 has been committed by a concurrent conflicting transaction. However, due to the way vector clocks are incremented upon commit, T_1 does not have enough knowledge to verify if y_1's committer was a conflicting transaction. Therefore the read operation returns a safe snapshot for T_1, which in this case is y_0 because y_0's vector clock (i.e., $y_0.VC$) is visible by T_1.

5.3.5 Commit protocol

The commit phase of transaction T is performed through the `COMMIT` function in Algorithm 10. If T is a read-only transaction, the commit phase only consists of a clean up step to remove traces of its execution on the version-access-set of its read versions. To do that, `Remove` messages are sent to the nodes where T read from (Lines 2-6 of Algorithm 10).

If T is an update transaction, similar to Walter the Two-Phase Commit (2PC) protocol is used to accomplish the commit phase and install new versions into the data repository. The node in which T executes (i.e., T's coordinator) starts the 2PC by sending a `Prepare` message to the (preferred) nodes that store the objects written by T (Line 10 of Algorithm 10). When a 2PC participant node N_i receives a `Prepare` message for T, all the written objects by T and stored by N_i are locked. If the locking acquisition succeeds, then versions are validated to certify that they have not being overwritten meanwhile.

At this point, the existing read-only transactions' identifiers in the versions-access-set of T's written objects are retrieved by the 2PC participants and sent back to the 2PC coordinator with the `Vote` message (Lines 3-10 of Algorithm 11). Once the coordinator receives all the `Vote` messages from participants, it merges all the received transactions' identifiers and include them into $T.collectedSet$ (Line 17 of Algorithm 10).

In the case all participants vote for committing T, meaning they were able to acquire locks on the written objects and validate their version, then N_i's sequence number ($CurrSeqNo_i$)

Algorithm 10 Commit of transaction T in node N_i using FPSI

```
1: function COMMIT(Transaction T)
   // Check if T is a read-only transaction
2:     if (T.writeset=$\phi$) then
3:         for ($k \in T.readKeys$) do
4:             send $Remove[T.id, k]$ site(k)
5:         $T.outcome \leftarrow true$
6:         return $T.outcome$
   // Start 2PC if T is an update transaction
7:     $commitVC \leftarrow T.VC$
8:     $T.collectedSet \leftarrow \phi$
9:     $T.outcome \leftarrow true$
10:    send $Prepare[T]$ to all $N_j \in sites(T.writeset)$
11:    for all ($N_j \in sites(T.writeset)$) do
12:        wait receive $Vote[collectedSet_j, result_j]$ from $N_j$ or timeout
   // Check if T's 2PC commit decision is successful
13:        if ($\neg result_j \vee timeout$) then
14:            $T.outcome \leftarrow false$
15:            break;
16:        else
   // Collect all existing anti-dependencies in T.collectedSet
17:            $T.collectedSet \leftarrow T.collectedSet \cup collectedSet_j$
18:    $currSeqNo_i \leftarrow currSeqNo_i + 1$
19:    $T.seqNo \leftarrow currSeqNo_i$
   // Finalize T's commit vector clock
20:    $T.commitVC \leftarrow siteVC_i$
21:    $T.commitVC[i] \leftarrow T.seqNo$
22:    send $Decide[T,\ T.outcome]$ to all $N_j \in sites(T.writeset \cup N_i)$
23:    send $Propagate[T,\ T.seqNo]$ asynchronously to all $N_j \in sites \backslash sites(T.write)$
24:    return $T.outcome$
```

is incremented and the commit vector clock of T is established. This vector clock is then sent along with the `Decide` message to the 2PC participants (Line 18-22 of Algorithm 10).

Lines 12-22 of Algorithm 11 show the steps taken by a 2PC participant N_i when it receives the `Decide` message from the coordinator executing on node N_j. In order for N_i to commit T, N_i must wait for all already decided/propagated transactions by N_j. N_i can easily detect if this wait condition should occur by looking at the gap between the node vector clock in position j and the commit vector clock of T at position j (e.g., $T.seqNo$). When T finally commits, the $siteVC$ of each 2PC participant is updated in the j^{th} position.

Similar to Walter, after sending the `Decide` of T FPSI sends the asynchronous `Propagate` message to all other nodes in the system in order to allow them to advance their reading

Algorithm 11 FPSI's Commit message handlers received by node N_i for transaction T issued by node N_j

```
upon receive $Prepare[Transaction\ T]$ from $N_j$ do
    $collectedSet \leftarrow \phi$
    boolean $result \leftarrow getLocks(T.writeset, owner = T.id) \wedge validate(T)$
    if $(\neg result)$ then
        $releaseLocks(T.writeset, owner = T.id)$
        send $Vote[collectedSet, result]$ to $N_j$
    else
        for all $k \in T.writese$ do
            $collectedSet \leftarrow collectedSet \cup k.version.accessSet$
        send $Vote[collectedSet, result]$ to $N_j$
end
upon receive $Decide[T, outcome]$ from $N_j$ do
    if $(outcome)$ then
        wait until $siteVC_i[j] = T.seqNo - 1$
        $update(T.writeset,\ T.seqNo,\ j)$
        for all $(k \in T.writeset)$ do
            $k.lastVersion.accessSet \leftarrow k.lastVersion.accessSet \cup T.collectedSet$
        $siteVC_i[j] \leftarrow T.seqNo$
        $releaseLocks(T.writeset, owner = T.id)$
    else
        $releaseLocks(T.writeset, owner = T.id)$
end
function validate($Transaction\ T$)
    for all $(k \in T.writeSet)$ do
        if $(k.lastVersion.VC[lastUpdaterSite] > T.VC[lastUpdaterSite])$ then
            return $false$
    return $true$
```

snapshot with respect to N_i. Note that, although FPSI requires `Propagate` messages to commit non-local update transactions, it does not abort these transactions as Walter does due to late delivery of `Propagate` messages. In fact, in Walter if a `Propagate` message from a node N_j is not delivered by a node N_i, a non-local update transaction from N_i will repeatedly fail its validation step causing an abort that will be solved only after receiving the `Propagate` message.

FPSI does not abort the update transaction in such a case. However, although it still needs the `Propagate` message to be delivered in order to finalize the commit, *i)* it is able to overlap the transaction execution with the delivery of the `Propagate` message, which is likely to arrive meanwhile; and *ii)* it reduces network traffic due to saving multiple transaction retries.

5.3.6 Handling Asynchronous Messages

Algorithm 12 shows how node N_i handles asynchronous messages, namely `Propagate` and `Remove`. When N_i receives a `Remove` message because a read-only transaction T committed at node N_j, T's identifier is removed from the version-access-sets of T's read versions whose preferred site is N_j, and from all other version-access-sets in N_j in which T's identifier has been propagated by concurrent update transactions that committed meanwhile (Lines 5-9 of Algorithm 12).

Algorithm 12 Remove and Propagate messages from transaction T issued by N_j to node N_i using FPSI

```
1: upon receive $Propagate[T, T.seqNo]$ from $N_j$ do
2:     wait until $siteVC_i[j] = T.seqNo - 1$
3:     $siteVC_i[j] \leftarrow T.seqNo$
4: end
5: upon receive $Remove[T.id,\ k]$ from $N_j$ do
6:     $k.version.accessSet \leftarrow k.version.accessSet \setminus \{T.id\}$
7:     for all $(k' : v \in k'.versionSet \wedge T.id \in v.accessSet)$ do
8:         $v.accessSet \leftarrow v.accessSet \setminus \{T.id\}$
9: end
```

Upon receiving a `Propagate` message by node N_i for the commit of an update transaction T from node N_j, N_i can advance its reading snapshot with respect to N_j to $T.seqNo$.

In PSI the outcome of all committed transactions that update some objects whose preferred site is N_j should be observed in the same order by N_i. For this reason, T should wait for all previously committed transactions in N_j with a lesser sequence number than $T.seq$ to be received by N_i (Line 2 of Algorithm 12). After that, $siteVC$ of N_i can be updated with $T.seqNo$ at the j^{th} position of $siteVC$ (Line 3 of Algorithm 12).

5.4 Correctness Arguments

We assess the correctness of FPSI by discussing how our modifications on top of the Walter distributed concurrency control still preserve PSI. The major difference between FPSI and Walter lies on the fact that FPSI can read the latest version of an accessed object upon the first access to a node, even if an asynchronous propagate message has not been being delivered yet. Our approach is to focus on the necessary and sufficient condition to assess if an execution satisfies the Generalized Snapshot Isolation (GSI) correctness level [41]. GSI generalizes SI by allowing reading snapshots to be arbitrarily old, but still disallows PSI's long fork anomaly. Showing the equivalence to GSI is enough since we have already shown that FPSI does not eliminate the long fork anomaly of Walter, as discussed in Section 5.2.3. Without considering this anomaly, PSI is equivalent to GSI [21, 41].

For a schedule to be accepted by GSI, if a transaction history has a cycle, then this cycle includes at least two adjacent anti-dependency edges in the Directed Serialization Graph [41].

Our correctness discussion shows that as soon as a transaction detects an anti-dependency with respect to a concurrent update transaction, a direct dependency, including a transitive one, cannot occur. This can be achieved by relying on either the content of the version-access-set (populated through the visible reads technique) for read-only transactions, or the selection of a safe snapshot for update transactions. As a consequence of this observation, only transactions executions with two adjacent anti-dependency edges can be committed by FPSI, which is needed to satisfy GSI (and PSI by including the long fork anomaly).

Regarding the reading policy of read-only transactions, since a transaction T_{RO} that reads a version o_v of object o is included in the version-access-set (Line 8 of Algorithm 9) of o_v, when an update transaction creates a new version o_{v+1}, the write-after-read (anti-) dependency is established and can be detected by any other reading transaction after that. That means, if a conflicting transaction, directly or transitively, produces a new version, that version cannot be returned by any subsequent read operation from T_{RO} because of the way the version-access-set is propagated to conflicting transactions, including those transitive (see Lines 16-17 of Algorithm 11. By doing that, there cannot be a read-only transaction involved in a loop with an outgoing anti-dependency edge preceded by an incoming direct dependency edge. In the presence of an established anti-dependency, our concurrency control reads previous versions, which transforms the aforementioned direct dependency into an anti-dependency, as demanded by PSI.

The argument for an update transaction T is simpler since it cannot always attempt to access the latest version of an object. In fact, after the first read operation, a safe reading snapshot is established for T. Such a reading snapshot guarantees that if a concurrent transaction T' overwrites a read version by T, since T's vector clock will be strictly lesser than T''s vector clock, T cannot include that newer version in its reading snapshot. (Recall that this conservative rule might produce false conflicts that can unnecessarily order T before T' as mentioned is Section 5.3.4).

5.5 Evaluation Study

FPSI's distributed concurrency control has been embedded into an in-memory distributed transactional key-value store. We use the code base of Walter available at [32] and we modify it to integrate FPSI's metadata and reading/writing policy. We recall that our performance assessment for FPSI aims at showing how its algorithmic modifications, which ensure higher level of freshness than Walter, can still provide comparable performance with respect to Walter, and retain significant performance improvement over a serializable distributed concurrency control [85].

We conduct the performance evaluation using two well-known OLTP benchmarks, YCSB [92] and TPC-C [97], both ported to the key-value data model. For YCSB, we have two transaction profiles: update, where two keys are read and written, and read-only transactions, where two keys are accessed. YCSB is configured to use keys of 4 bytes and values of 12 bytes. TPC-C is a more complex benchmark that simulates an order-entry environment with several warehouses. It includes five transaction profiles, three of them are update transactions and the remaining are read-only transactions.

We configure the benchmarks to explore different runtime scenarios. First, YCSB transactions are shorter than TPC-C's transactions; also, since update transactions in YCSB write the same keys they read, the final execution is equivalent to an execution in which the concurrency control ensures Serializability. This is done to particularly stress the importance of reading a fresh snapshot for update transactions. In fact, FPSI will be able to reduce the number of aborts of update transactions due to outdated reading snapshot in Walter. On the other hand, TPC-C transactions' logic allows for reading and writing different shared objects, showing a favorable case for Walter since an outdated reading snapshot still suffices to commit while preserving PSI.

We compare the performance of FPSI against Walter, which guarantees PSI, and 2PC-baseline (2PC in the plots), a serializable key-value store where all transactions execute optimistically and rely on the Two-Phase Commit protocol to commit both update and read-only transactions, thus without needing multiversioning. We also included a version of Walter and FPSI in which the asynchronous propagate messages are intentionally delayed to show the effect of such an event on the abort rate of update transactions.

In all the experiments there are five application threads (i.e., clients) per node injecting transactions in a closed-loop (i.e., a client issues a new request only when the previous one has returned). In terms of transaction mix, we evaluate our competitors using 20% and 50% read-only transactions. We do not include the test with 80% read-only transactions because performance of both Walter and FPSI are almost identical using this configuration, especially when the contention is low. This is expected since most of the algorithmic differences between the two competitors are related to the propagation of anti-dependency developed with update transactions. If version-access-sets are almost empty, the performance of read-only transactions in both competitors will be similar.

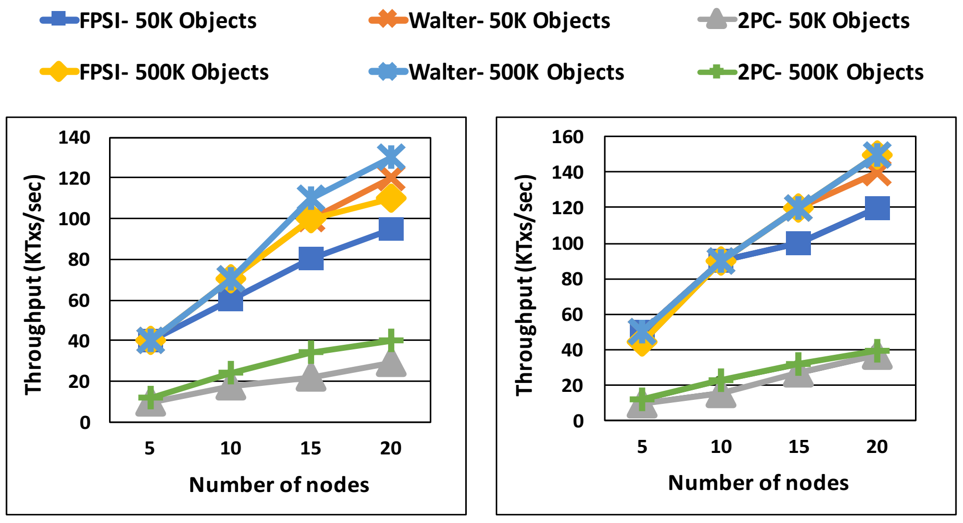
In both benchmarks, transactions select keys to be accessed using a uniform distribution, which entails accesses might or might not be to the local data repository. We do not test the case of a skewed access distribution to highlight the performance impact of FPSI design. In fact, if accesses target local nodes, data freshness is already guaranteed to be the highest level. In this scenario, FPSI performs equally to Walter since no protocol modification has been made to Walter to improve the freshness of local accesses. In terms of data distribution, keys are evenly distributed across nodes.

As test-bed, we use CloudLab [93], a cloud infrastructure available to researchers. We selected 20 nodes of type c6320 available in the Clemson cluster. This type is a physical machine with 28 Intel Haswell CPU-cores and 256GB of RAM. Nodes are interconnected using a 10Gb/s network, which delivers a message in about 20 microseconds without saturation. Considering that, we set the timeout on lock acquisition to 1 ms. All the results are the average of 5 trials.

5.5.1 YCSB

Figure 5.4 shows the throughput of all competitors using YCSB and a total of 50k and 500k shared keys while increasing the total number of nodes. Recall that more nodes means more clients injecting transactions in the system, therefore an increasing level of contention. In all these configurations, the measured abort rate is below 10% at the highest contention level (i.e., 50k keys and 20 nodes).

The performance and scalability of FPSI match Walter's in the cases where contention is low, namely up to 10 nodes in all tested cases and in the 500k configuration. When contention increases (e.g., due to the higher number of clients), the gap between FPSI and Walter becomes more visible. This is because of two factors: the additional synchronization steps needed by FPSI's read operations, and the increasing size of version-access-sets (see Figure 5.5). Quantifying, for 20% read-only workload the highest gap measured between FPSI and Walter is 20% and 16% with 50k and 500k keys, respectively. At 50% read-only workload, the gap is 15% at 50k keys, and such a gap is annulled at 500k keys.



(a) 20%.

(b) 50%.

Figure 5.4: Throughput of FPSI, Walter and 2PC-baseline using YCSB and by varying % of read-only transactions, the total number of keys, and the number of nodes.

Both PSI competitors substantially improve performance over 2PC-baseline because its read-only transactions undergo an expensive commit phase using the 2PC protocol, which is skipped by FPSI and Walter since their read-only transactions are abort-free. Achieved speedup of PSI competitors against 2PC-baseline is constantly more than 3x.

As observed earlier, the size of version-access-set impacts the gap in performance between FPSI and Walter when the contention increases. Figure 5.5 confirms that. In this figure, we report the average number of collected anti-dependency while an update transaction in FPSI undergoes the prepare step of its commit phase. We explored the configurations with 20%, 50%, and 80% of read-only transactions, with 50k, 100k, and 500k shared objects.

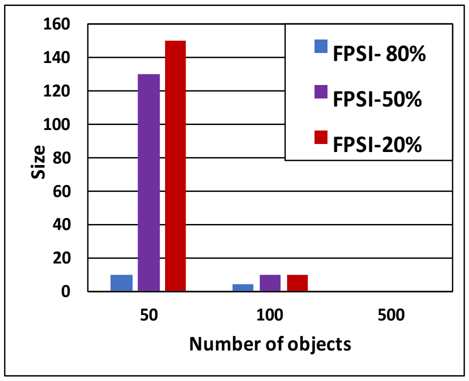


Figure 5.5: Average size of anti-dependency collected by update transactions during prepare phase of FPSI for different % of read-only transactions and keys.

Increasing the percentage of update transactions increases the number of anti-dependencies. The sharp jump from 80% to 50% read-only at 50k keys is due to the transitive propagation of those anti-dependency. In fact, if an update transaction reads a key whose version-access-set includes a number of read-only transaction identifiers, this set of real-only transactions will be propagated to the version-access-set of the new written versions of the update transaction upon its commit.

In Figure 5.5, we also test the cases of 100k and 500k keys to show how the size of collected anti-dependencies gradually decreases to zero, as with 500k. Note that, YCSB transactions are short, therefore the chance for an anti-dependency to occur at the low contention case, such as using 500k keys, is low.

To show the effectiveness of a fresher reading snapshot for read operations of update transactions, in Figure 5.6 we measure the abort rate (of update transactions since read-only transactions cannot abort) using 20 nodes in case we intentionally delay the asynchronous propagate messages (by 1 ms) in both FPSI and Walter. We select 1 ms because in our testbed it mimics around 5x slowdown of network delay, which might be due to congestion

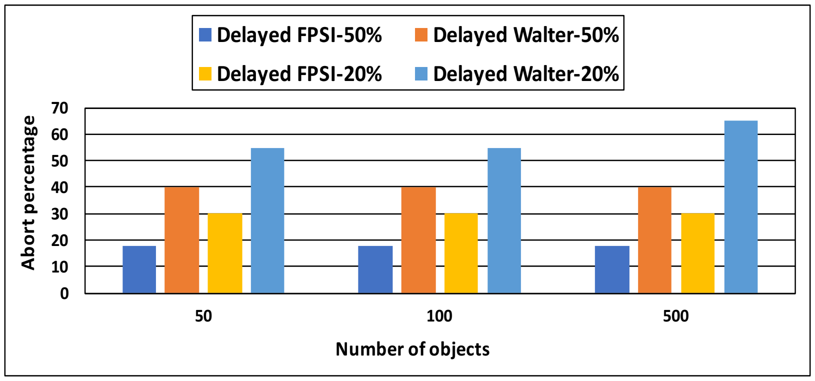


Figure 5.6: Abort rate using 20 nodes and varying number of keys while delaying propagate messages in both FPSI and Walter.

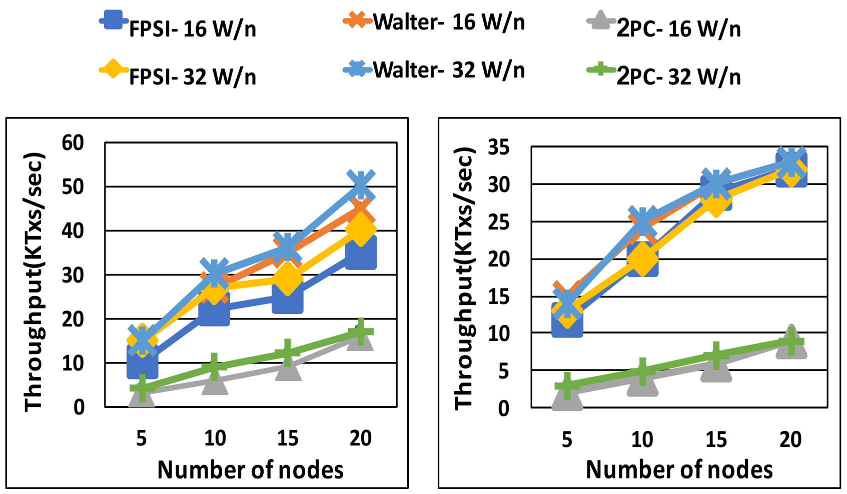
at high utilization.

Walter's abort rate is on average twice the one of FPSI. The reason for such a significant increase for Walter is because update transactions' reading snapshot in our configuration of YCSB should be the freshest since the same read keys are also written, therefore they need to be validated. Slowing down the propagate messages forces update transactions in Walter to repeatedly abort before being able to commit when finally the node's vector clock is updated. Another interesting aspect to be observed is that in general, the abort rate does not decrease while the contention decreases at 500k keys. This is due to the fact that, even if contention is absent, a transaction in Walter may not be able to read the latest version of a key because of an outdated node vector clock.

Abort rate increases in both Walter and FPSI with respect to the case where asynchronous messages are not delayed because update transactions still need to receive the propagate messages in order to finally commit. While they wait for such a message, they hold the locks on their written keys. Therefore, the lock holding time increases, as the abort rate. The abort rate without delaying propagate messages is below 10% for both PSI and FPSI, and decreases in low-contention scenario.

5.5.2 TPC-C

TPC-C transactions are much longer than YCSB's, especially the read-only ones. Generally, the performance at 50% read-only workload is slower than the one at 20%. Because of the hierarchical object access pattern of TPC-C, the contention in the system is modified by varying the number of warehouses (the warehouse object sits at the top of this access hierarchy).

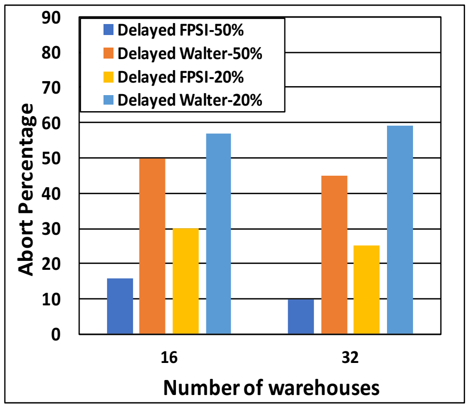


(a) 20%. (b) 50%.

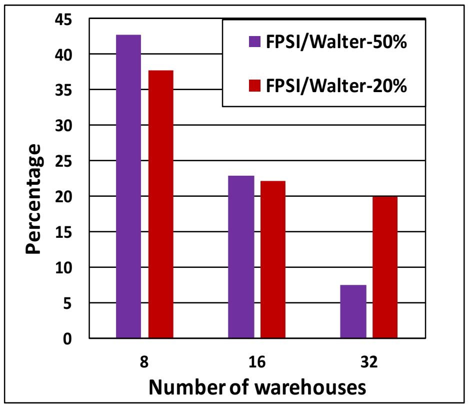
Figure 5.7: Throughput of FPSI, Walter and 2PC-baseline using TPC-C and by varying % of read-only transactions, the number of warehouses per node (W/n), and the number of nodes.

Figure 5.7 shows the results for all competitors varying the number of nodes and the number of warehouses per node. As opposed to YCSB benchmark, in TPC-C transactions do not necessarily read the same keys that they write. This allows an update transaction to commit even if the reading snapshot is not the freshest. The consequence of this characteristic is that PSI competitors are much faster than 2PC-baseline, and both Walter and FPSI has a similar growing trend. In fact, with 50% read-only transactions, the performance of the two PSI competitors are withing 5% of each other. At 20% read-only workload, the maximum observed gap is 28%.

Figure 5.8(a) includes the abort rate measured at 20 nodes deploying 16 and 32 warehouses per node in the case where the propagate messages have been intentionally delayed. Without delaying them, abort rate of Walter and FPSI is comparable. Walter shows an average of almost 4x higher abort rate than FPSI. This is because of the way the safe snapshot is selected by update transactions in FPSI. In fact, according to TPC-C logic, the warehouse is often the first accessed key, which is guaranteed to be the latest version by FPSI's concurrency control, subsequent accesses to objects will be likely related to that warehouse. This



(a) Abort rate.



(b) Slowdown of FPSI over Walter varying # of warehouses.

Figure 5.8: Performance of FPSI and Walter varying the number of warehouses per node (W/n).

pattern ensures that all the objects updated along with that warehouse will be accessed by reading the latest version. Because of that, FPSI's degradation in abort rate is less than Walter's.

Finally, in Figure 5.8(b) we show the slowdown in throughput between FPSI and Walter when we vary the number of warehouses, using 20 nodes. When 8 warehouses per node are deployed, contention is higher, therefore the slowdown of FPSI with respect to Walter increases. This is expected because the cardinality of the version-access-set in high contention increases. On the other hand, reducing contention also reduces the performance gap between the two PSI competitors.

Chapter 6

On the Correctness of Transaction Processing with External Dependency

6.1 Overview

When the concurrency control implementation of a transactional system is required to enforce an application-level invariant on shared data accesses (i.e., an expression that should be preserved upon every atomic update [38]), ad-hoc reasoning about its correctness is a tedious and error-prone process. Traditional (data-related) constraints (e.g., transaction conflicts) are well-formalized with established correctness levels, such as Serializability [35–37] and Snapshot Isolation [36, 41]. However, a unified model encompassing the various *external* (semantic-related) constraints that enforce application invariant has not been formalized yet.

In this chapter we make a step towards defining such a model. We introduce a theoretical framework that formalizes correctness levels stronger than (or equal to) Serializability by defining their transaction ordering relations as a union of two sets of data and external dependency. This approach is opposed to the traditional way of defining these relations through an ad-hoc analysis. This framework can be used to define an offline checker that verifies the safety of transactional executions. Assuming a serializable concurrency control [36], relations between transactions in an execution can be characterized as data dependency, if they are generated by data conflicts, or external dependency, if they affect the satisfaction of application invariant. This decomposition allows us to define a methodology to enrich the traditional transaction Direct Serialization Graph (DSG) [36] with such external ordering relations. We use the formalization to introduce a safety condition that verifies correctness of transactional executions (Theorem 6.2.3).

We motivate our model by showing an example of application with associated invariant. The example mimics a simple monetary application that imposes different requirements to clients interacting from different branch locations of the bank. The application mandates the following invariant: when a transaction is issued by a client in one branch, this transaction accesses the modifications performed by the latest transactions completed on the same branch prior its starting. At the same time, the application does not require special constraints on the order of monetary transactions issued from other branches. That is, transactions from a remote branch should execute atomically and in isolation, but they might access stale data.

Suppose clients C_1 and C_2 from branch α issue two subsequent non-concurrent transactions T_1 and T_2 accessing the same bank account Ac. The first deposits \$10 and the second checks the total amount of Ac and then withdraws the latest deposited amount (\$10). According to the application semantics, T_2 must observe the deposit by T_1. Consider another transaction T_3, issued by a client from branch β doing auditing on accounts, including Ac. Application semantics for T_3 does not enforce any requirement on the set of transactions whose outcome should be observed, including T_1 and T_2. A serializable concurrency control would “only" guarantee a transactions order of T_1, T_2 and T_3 equivalent to some serial order. This serial order does not consider the application invariant and might order T_2 before T_1. Such a mismatch is due to the lack of application invariant representation in the concurrency control.

One solution to overcome this problem in a serializable concurrency control is to provide session guarantee [98], meaning transactions from one branch belong to the same session. This guarantee imposes an additional constraint between T_1 and T_2 where T_2 must observe the output of T_1. Clearly, T_3 would belong to a different session. The other solution would be adopting a stronger correctness level (e.g., strict serializability [36]) among all transactions, irrespective of their originating branch. An even more conservative solution is to apply external consistency [28], which brings the clients perceived order among transactions into the concurrency control so that mismatches are prevented.

With our unified model, these three correctness levels can be modeled in the same way as a combination of data-related transaction dependency, to satisfy Serializability constraints, and external transaction dependency, to satisfy application invariant. This way, despite the differences among these correctness levels, our model can assess the correctness of concurrency controls that satisfy each of them by relying on a single framework.

6.2 Formalization

A history [36] models the interleaved execution of a set of transactions T_1, T_2, ..., T_n, as an ordered sequence of their operations (such as *read*, *write*, *abort*, *commit*). First, we recall the formalization of the transaction dependency graph, already introduced in Chapter 2. The $DSG(\mathcal{H})$ for a history $\mathcal{H}$ represents the data-related dependency among transactions in $\mathcal{H}$. Roughly, in this graph each node is a committed transaction in $\mathcal{H}$, and each directed edge between two nodes can be of the following categorise:

- *read dependency:* ($T_i \xrightarrow{\text{WR}} T_j$) A transaction T_j read-depends on T_i if a read of T_j returns a value written by T_i.
- *write dependency:* ($T_i \xrightarrow{\text{WW}} T_j$) A transaction T_j write-depends on T_i if a write of T_j overwrites a value written by T_i.
- *anti-dependency:* ($T_i \xrightarrow{\text{RW}} T_j$) A transaction T_j anti-depends on T_i if a write of T_j overwrites a value previously read by T_i.

Definition 6.2.1. *$DSG(\mathcal{H})$ contains a set of tuples and each tuple has the following form: $(T_i, T_j, type)$. This representation shows that a directed data-related (read/write/anti-) dependency edge exists from transaction T_i to transaction T_j. $DSG(\mathcal{H}) = \{(T_i, T_j, type) : i, j \in \{1, .., n\} \wedge type \in \{RW, WW, WR\}\}$.*

Since our model focuses on correctness levels stronger than, or equal to, Serializability, we recall that a history $\mathcal{H}$ is serializable if its corresponding DSG does not contain any cycle [36].

Performing an offline analysis of the DSG graph is a convenient tool for reasoning about the correctness of data-related dependencies produced by a concurrency control. However, it does not help verifying correctness of application when invariant should be preserved in addition to Serializability. Our model aims at filling this gap, as follows.

Definition 6.2.2. *An External Dependency Graph (EDG) for a given history $\mathcal{H}$, denoted as $EDG(\mathcal{H})$, determines application-level constraints. In this graph, an edge from transaction T_i to transaction T_j means an application-level requirement forces an external dependency between T_i and T_j. We say T_j externally-depends on T_i ($T_i \xrightarrow{EXT} T_j$).*

Intuitively, application invariant expressed by EDG should neither violate data-related dependency produced by the concurrency control nor include any two contradicting constraints. This observation leads to the following theorem where, informally, we consider both DSG and EDG as a single graph made by the union of them. We can check if a history is serializable and does not violate application invariant by verifying that the aforementioned single graph does not contain any cycle.

First, given a history $\mathcal{H}$ of n transactions, we define DSG, EDG, and their union as follows:

- $DSG(\mathcal{H}) = \{(V, E1) : V = \{T_i : i \in \{1,..,n\}\} \wedge E1 = \{(T_i, T_j, type) : i, j \in \{1,..,n\} \wedge type \in \{WR, WW, RW\}\}$.
- $EDG(\mathcal{H}) = \{(V, E2) : V = \{T_i : i \in \{1,..,n\}\} \wedge E2 = \{(T_i, T_j, type) : i, j \in \{1,..,n\} \wedge type \in \{EXT\}\}$.
- $DSG(\mathcal{H}) \cup EDG(\mathcal{H}) = (V, E1 \cup E2)$.

We now define our new *External Serializability* consistency level. We call a history $\mathcal{H}$ Externally Serializable (or EC-SR) if: *1)* it is serializable, and *2)* external dependency defined by the edges of its EDG are not violated. To prove that, it is necessary and sufficient to show that the union of its DSG, built from the concurrency control implementation, with its EDG, built from application invariant, does not have any cycle. We formalize that in the following theorem (the proof is intuitive and omitted due to space limitations):

Theorem 6.2.3. *A history $\mathcal{H}$ satisfies EC-SR iff $DSG(\mathcal{H}) \cup EDG(\mathcal{H})$ does not have any cycle. A concurrency control CC satisfies EC-SR iff all the histories produced by CC are EC-SR.*

6.3 Designing a System using our Unified Model

According to the formalization provided in Section 6.2, a static analysis on the DSG of a given history detects dependencies between transactions accessing the same memory locations. This static analysis helps programmers to assess the correctness of concurrency control with respect to transactional accesses (e.g., read, write, etc.) to the shared objects and provides a necessary and sufficient condition to verify if an execution is safe with respect to the targeted isolation level.

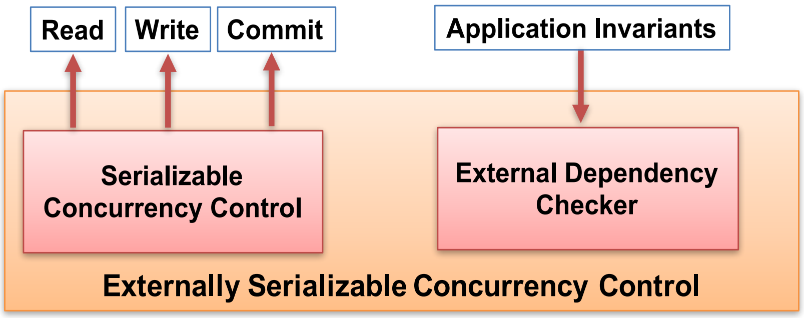

Read
Write
Commit
Application Invariants
Serializable Concurrency Control
External Dependency Checker
Externally Serializable Concurrency Control

Figure 6.1: Designing an Externally Serializable Concurrency Control. The Serializable Concurrency Control determines provided rules by the serializable concurrency control. Externally Dependency Checker determines the information regarding the external dependency provided by programmer.

However, DSG analysis on a produced schedule by a concurrency control does not disclose applications' requirements. In order to capture the application requirements, an additional step is necessary to be taken by modeling application requirements inside *EDG*. The *EDG* analysis models the information which is application specific such as programmer-provided external dependency (or application invariants). These requirements are defined by programmers inside applications and are applicable on top of a given concurrency control. The combination of these two steps allow programmers to reason about the behavior of concurrency control and make sure that external dependencies are compatible with data-related dependencies. Figure 6.1 represents the design components of an externally consistent serializable concurrency control.

The figure envisions a concurrency control that exposes the traditional APIs for reading/writing shared objects, and to commit or abort transactions. The concurrency control should also expose an API so that applications can provide the concurrency control with additional ordering constraints so that internally the system can enforce them transparently.

Chapter 7

EPSI: Efficient Erasure-coded Parallel Snapshot Isolation for Key-Value Stores

7.1 Overview

Key-value stores [7,15,18,21,57,58,74,80,99–102] are among the most deployed data repository systems for storing unstructured data because they do not require a rigid data model, as opposed to the more traditional database relational model [16]. Key-value stores offer interfaces to atomically read and write a single key [82,103], as well as to initiate and commit transactions [15,21,32,57,74]. Because of their appealing interfaces and flexible data model, key-value stores are often the preferred data repository for large scale social network applications [21,57,104]. These types of applications produce a large amount of data [7,105] and they are particularly interested in guaranteeing high performance operations and high availability of the stored data.

The major drawback of existing solutions that provide the latter two properties is the high *storage cost*. In fact, high availability and fault tolerance are traditionally implemented using full-replication techniques [14,53,106], such as State Machine Replication [14], where redundant copies of the entire data repository are kept consistent and ready to be utilized so that user will not experience any noticeable issue upon failures.

In this chapter, we present EPSI, a key-value store design that provides high performance, high availability, and reduces the storage cost. With this composition of properties, EPSI finds its ideal deployment with social networking applications. The above two properties are achieved by EPSI as follows.

- High performance is ensured by integrating Walter [21], a distributed concurrency control providing Parallel Snapshot Isolation (PSI) as the correctness level. PSI is weaker than Serializability (see Section 5.2 for a comprehensive description of Walter's guarantees), and it is specifically designed to match the needs of social network applications.

- High availability is ensured by partitioning the data repository into shards [29, 31, 74], and each shard uses the erasure coding technique [55, 56] to encode each shared object in a set of *coding elements*, each of which is stored on a separate replica in that shard. The process of encoding an object in coding elements enables a significantly lower space utilization (see Section 3.6 for a summary of existing erasure coding techniques) as opposed to the traditional full-replication approach. On the other hand, in this structure accessed shared objects need to be decoded (rebuilt) from their corresponding coding elements.

Traditionally erasure coding is used by data repositories to improve the performance of the recovery steps upon failures [57, 59, 81, 82]. The peculiarity of EPSI is that while providing high availability and fault tolerance using erasure coding, performance is also improved since erasure coding allows for read operations to be served by a quorum of nodes within a shard of objects.

For generality, EPSI has been designed for easy integration with existing distributed concurrency controls. In fact, EPSI APIs internally rely on an existing distributed key-value store engine to transactionally read/write coding elements from/to the data repository. This feature enables an easy deployment of EPSI in different erasure coding schemes, concurrency controls, and application workloads.

We summarize our contributions in this chapter as follows:

- EPSI, a sharded architecture for a high performance, highly available transactional key-value store. EPSI reduces storage cost by relying on erasure coding techniques for storing objects on nodes within a shard. EPSI is optimized for read-dominated workload and supports parallel writes by leveraging the PSI correctness level.

- EPSI's improved performance robustness, which scales with the size of the stored objects. In fact, the way EPSI integrates erasure coding allows for a more effective network utilization especially in the presence of write workload, as opposed to existing solutions, such as Cocytus [57], that saturate the network bandwidth at high object sizes or that use erasure coding only during recovery upon failures.

- The integration of EPSI with Walter's concurrency control and an extensive experimental study to evaluate its performance and scalability against state-of-the-art competitors, including those that do not deploy erasure coding, and those that utilize stronger correctness levels, such as Serializability.

7.2 Background

7.2.1 Walter Protocol

Social networking applications such as Facebook and Twitter, require a highly available storage system to keep a huge amount of user data, such as status updates and photos [7,10]. For example, the median of the object value sizes of Facebook is 4.34 KB for Region and 10.7 KB for Cluster [7]. An attractive storage choice for this setting is a key-value store [100] because of its efficiency, flexibility, and low latency in accessing values. One important key factor in these types of applications is dealing with high availability when users issue large number of concurrent requests.

Walter [21] is a state-of-the-art distributed key-value store transactional system whose concurrency control implements Parallel Snapshot Isolation (PSI), as a suitable consistency model for social networking applications (PSI consistency level is studied in Section 5.2.1). By relying on the lazy replication technique, Walter asynchronously propagates the outcome of update transactions to all nodes in the system.

Lazy replication in Walter is designed by logically assigning objects to *preferred nodes*. A preferred node always stores the latest version of an object. The object might also be replicated on other *non-preferred* nodes, which might not always have the latest version of objects. If a transaction begins on a node N and reads an object whose preferred node is N, this transaction is considered a `local` transaction and it is able to access the latest existing version of the object. Otherwise, when a transaction begins on a non-preferred node or any other node (for brevity, in both these cases we refer to this transaction as *non-local*), the read operations can result in an *outdated* object version. Walter achieves high performance for read-only workload at the cost of reading arbitrarily old data in case accessing non-local objects by read-only transactions, which is an acceptable trade-off for applications with social networking characteristics.

In the following, we overview Walter concurrency control APIs, since the implementation of EPSI in this dissertation relies on Walter concurrency control to perform operations on the date repository.

Walter's Overview and Metadata

When transaction T starts on a node, the $WalterBegin(T)$ API is called by the Walter concurrency control. For operating a read operation on object key, the $WalterRead(T, key)$ API is invoked and every write operation on object key needs calling $WalterWrite(T, key, val)$ to update the corresponding value of key to the value val.

A read-only transaction T in Walter only calls $WalterRead(T, key)$ one or multiple times before it completes. If T is an update transaction, T can be tried to be committed by relying on the Two-Phase Commit protocol (2PC) [85]. The new versions of objects written by T are installed in the data repository if the commit decision is made by 2PC protocol and therefore, T successfully completes.

Figure 7.1 presents the set of metadata associated with every transaction and every node in Walter's concurrency control. Every node in Walter is assigned with a unique number between 0 and $n - 1$, where n is the total number of nodes in the system. This identifier is called $node_{id}$ (i.e., N_{id}). The node with the identifier N_i is also assigned a scalar called $CurrSeqNo_i$, which represents the sequence number of the latest transaction issued and

Transaction T's State:
$T.writeset$
$T.VC$
$T.commitVC$
$T.seqNo$

Node's State:
$node_{id}$
$node.CurrSeqNo_{id}$
$node.VC_{id}$

Figure 7.1: Transaction metadata and node metadata in Walter.

committed at node N_i.

Since Walter synchronizes transactions using vector clocks [69], each node N_i is associated with a vector clock, called $node.VC_i$. Note that the size of all used vector clocks in Walter is equal to the number of nodes in the system. The j^{th} entry of this vector clock in node N_i represents the last transaction from node N_j that was committed at node N_i.

The state of transaction T is described by two different vector clocks, one called $T.VC$ and the other called $T.commitVC$. $T.VC$ encapsulates the knowledge of T with respect to the logical timestamps of other nodes. In practice, $T.VC$ is used as visibility bound for all versions accessible by T. When the commit decision for transaction T issued by N_i is made, the $CurrSeqNo_i$ is incremented and $node.VC$ of N_i (i.e., $node.VC_i$) is updated at the i^{th} position and the updated value of $node.VC_i$ is assigned to transaction commit vector clock (i.e., $T.commitVC$). Every transaction T during the execution and before commit buffers its written objects in a private buffer called $T.writeset$.

Walter Concurrency Control APIs

Algorithm 13 shows the summary of steps taken by the implementation of each API in Walter concurrency control. When $WalterBegin(T)$ is called, $T.VC$ is initialized by the value of the $node.VC$ of the node where the client invokes the operations. This initialization sets the visibility bound of T for the subsequent read operations. In order to read key k, $WalterRead(T, k)$ is used and the state of k is either returned from the local data repository, if present, or it is returned by a remote node that stores k. The returned state of k should comply with the visibility bound of T or $T.VC$ (Lines 3-4 of Algorithm 13). The API $WalterWrite(T, k, val)$ simply buffers the value val in $T.writeset$ without installing it into the data repository (Lines 5-6 of Algorithm 13).

Algorithm 13 Walter API description in the node N_i

```
1: function WalterBegin(Transaction T)
2:     $T.VC \leftarrow N_i.VC$

3: function WalterRead(Transaction T, key k)
4:     return the state of k from T.writeset and all versions in the local storage visible to
       T.VC if it is locally stored, or all versions in the remote node storage visible to T.VC

5: function WalterWrite(Transaction T, key k, value val)
6:     append (k, val) to T.writeset

7: function WalterPrepare(Transaction T, Set participants)
8:     if (a write-conflicting transaction has committed after T started on a participant)
   then
9:         return Abort
10:    else
11:        if (a write-conflicting transaction is currently executing on a participant) then
12:            return Abort
13:        else
14:            return PrepareOK

15: function WalterCommit(Transaction T, Set participants)
16:    $T.seqno \leftarrow ++CurrSeqNo_i$
17:    for all ($N_j \in participants$) do
18:        update(T.writeset, T.seqNo, T.commitVC)
19:        wait until transactions with lesser sequence numbers commit
20:        $node.VC_j[i] \leftarrow T.seq$
21:        return true

22: function WalterPropagate(Transaction T, Set participants)
23:    for all ($N_j \notin participants$) do
24:        send Propagate[T, T.seqNo] asynchronously to all $N_j$
.
```

After finishing all operations of T, the commit phase of T is started by invoking the $WalterPrepare$ API. This API starts running 2PC by contacting the nodes which store $T.writeset$ to prepare these keys if there is not any other transaction that is concurrent with T and conflicting (Lines 7-14 of Algorithm 13). If there is no concurrent transaction that its $writeset$ has a conflict with $T.writeset$, $WalterCommit$ can be called by the coordinator of

T. The coordinator first increments its associated $currSeqNo$ and sets that as the sequence number of transaction (Line 16 of Algorithm 13). The coordinator then contacts all the participant nodes to install the updates and update their corresponding $NodeVC$ (Lines 17-21 of Algorithm 13).

At this stage T completes and the outcome of T can be propagated to the rest of the nodes in the system by calling $WalterPropagate$ API (Line 22-24 of Algorithm 13).

7.2.2 Erasure Coding

Different models exist in the literature for providing fault tolerance and durability in the presence of transactional modifications. In a replicated model [21,32,50], each object (e.g., key, value, associated metadata) is stored by more than one node in the system. In case a failure occurs, objects stored in the failed node can be retrieved from the other non-faulty nodes. Typically, in a replicated scheme, to tolerate M failures, M additional storage nodes and their corresponding processing power is needed. For example, if M is equal to 2, the system can tolerate two node failures, however, the storage efficiency of a key-value store can only reach 33%, meaning about 66% of the storage is accounted for the redundancy.

Erasure coding (EC) is an alternative redundancy technique to replication in storage systems due to its space efficiency and the continuous reduction in its computation overhead [57,107]. In erasure coding, the data is divided into a set of fixed-size coded units called *coding elements* through a process called *encoding*.

The coding elements are built using two configurable parameters called M and N ($N < M$). Here, M is the total number of coding elements produced from encoding the original data; N is the number of *data coding elements*; $(M - N)$ is the number of *parity coding elements*. Using an erasure coded data, any N of the M coding elements can decode the original data. The latter property of erasure codes is called *maximum distance separable (MDS)* [55].

Reed-Solomon codes (RS-code) [55] is a commonly used of erasure coding scheme which computes parity coding elements according to its data coding elements [55]. We denote the corresponding RS-code scheme using M total coding elements and N data coding elements as $RS(N, M)$. $RS(N, M)$ can tolerate $M - N$ node failures at most.

7.3 System Model

EPSI assumes a system made of a set of nodes that share neither memory nor a global clock. Nodes communicate through message passing over reliable asynchronous channels, meaning messages are guaranteed to be eventually delivered unless a crash happens at the sender or receiver node. There is no assumption on the speed and on the level of synchrony among nodes. We consider the classic crash-stop failure model: nodes may fail by crashing, but do not behave maliciously. A node that never crashes is correct; otherwise, it is faulty.

Every node maintains shared objects (or keys) adhering to the key-value model [32, 50, 57, 58]. The data repository is multi-versioned, meaning each shared object keeps a list of previous versions. Each version stores the value and the commit vector clock of the transaction that produced the version.

The entire data repository is sharded. Each shard is highly available through encoding all the objects in that partition across a set of nodes that belong to that shard.

EPSI deploys erasure coding to guarantee fault tolerance and the durability of objects stored by a shard. We assume each shard to be composed of M nodes. The value v of a key is therefore divided into M coding elements represented in a vector $[c_1, ..., c_M]$. Each c_i corresponds to a specific key stored in node i. For encoding, first v is divided into N elements $(v_1, ..., v_N)$. If the size of v is s, then each v_i has the size equal to s/N. The encoding function takes the $v_1, ..., v_N$ as input and produces a vector in the form of $[c_1, ..., c_M]$. The vector $[c_1, ..., c_M]$ stands as the vector of *code words* corresponding to the value v. The size of each c_i is equal to s/N.

In the erasure coded scheme of EPSI, we store one coded element c_i of a key per node in the appropriate shard. As a consequence of this process, a single node in a shard does not own enough information to rebuild the value of a shared object; it always needs a certain number of other coding elements to rebuild the object value, depending on the configuration of erasure coding technique used.

Within a shard, EPSI elects one node, called the *leader* of the shard. This node is in charge of performing commit procedures on objects stored in the shard.

For object reachability, EPSI assumes the existence of a local look-up function using consistent hashing [101], a commonly used technique to map keys to nodes, or shard as the case of EPSI.

Transactions that do not execute any write operation are called read-only, otherwise, they are update transactions. Read-only transactions are expected to be identified by the programmer.

We model transactions as a sequence of read and write operations on shared objects (or keys), preceded by a *begin* operation, and followed by a *commit* or *abort* operation. A client application begins a transaction on any node. If the transaction is an update transaction, then it is forwarded to the leader of the shard for execution. If the transaction is read-only, then it can execute on any node. No apriori knowledge on the accessed keys is assumed.

In terms of consistency level, EPSI preserves Parallel Snapshot Isolation since internally it uses Walter [21] to read and write coding elements.

7.4 Overview and General Architecture

At the core of EPSI there is a concurrency control middleware that exposes transactional APIs to the application and leverages the APIs of an existing distributed concurrency control to read and write data stored in the data repository. In our design for EPSI, we decide to leverage Walter as the underlying distributed concurrency control to perform operations on the erasure-coded data repository.

$EPSIBegin()$
$EPSIRead(key) \rightarrow object$
$EPSIWrite(key, value)$
$EPSICommit()$
$EPSIAbort()$

(a) EPSI APIs

$WalterBegin()$
$WalterRead(key) \rightarrow object$
$WalterWrite(key, value)$
$WalterCommit()$
$WalterAbort()$

(b) Walter APIs

Table 7.1: Application interfaces of EPSI and Walter.

Table 7.1(a) lists the APIs that EPSI exposes to the application layer. On the other hand, Table 7.1(b) shows the APIs of Walter, which are internally used by EPSI. EPSI's application interfaces implement the logic of the layered concurrency control and their primary goal is to invoke the internal APIs of Walter, listed in Table 7.2, on the specific nodes of the systems.

Non-leader's interfaces:
$EPSIBegin(T)$
$EPSIRead(T, key) \rightarrow object$
Leader's interfaces:
$EPSIBegin(T)$
$EPSIRead(T, key) \rightarrow object$
$EPSIWrite(T, key, value)$
$EPSICommit(T, participants)$
$EPSIAbort(T)$

(a) EPSI internal interfaces

Node's interfaces:
$WalterBegin(T)$
$WalterRead(T, key) \rightarrow object$
$WalterWrite(T, key, value)$
$WalterPrepare(T, participants) \rightarrow$
$PrepareOKorAbort$
$WalterCommit(T, participants)$
$WalterPropagate(T, participants)$
$WalterAbort(T)$

(b) Walter internal interfaces

Table 7.2: Internal interfaces of Walter, used by EPSI's concurrency control.

In order to decouple Walter's concurrency control from EPSI's, the data distribution is entirely provided by EPSI, while Walter's handlers are simply scheduled for execution on selected nodes to implement EPSI's protocol logic.

In a nutshell, EPSI processes transactions over highly available data and optimizes storage cost by exploiting the erasure coding technique. Metadata associated with the shared objects in the data repository are not stored using erasure coding; instead, object values are encoded as follows. Let us assume for simplicity in the explanation that each shard S_i is made of M nodes (i.e., $M = |S_i|$). EPSI deploys Reed Solomon technique, specified as $RS(N, M)$, which divides an object o into M coding elements, and requires N of them to rebuild the original value. As a result of this scheme, some of M nodes, denoted with N_{Dk}, store a fraction of the value of the stored object; the remaining nodes in M, denoted by N_{Pk}, store parity information associated with o.

Read operations in EPSI are served by contacting all nodes (i.e., M) within a shard and waiting for a quorum of responses of size N to be able to decode the original value of the read object. This way of executing read operations showcases the full integration of erasure coding into EPSI concurrency control. In fact, by relying on quorum reads, EPSI can overcome the performance bottleneck due to one or more slow nodes, depending on the selected erasure coding configuration. Effectively, the read requests for a single object are sent in parallel to the nodes of the target shard, without incurring in any additional sequential cost.

Erasure coding directly impacts the way new versions of written objects are stored. In fact, when an update transaction reaches the stage where its commit has been decided, all new versions of written objects should be encoded in order to produce the M coding elements to be stored into the M nodes of each target shard.

The process of encoding is a CPU intensive task and its overhead is not incurred by other data repositories that do not rely on erasure coding for optimizing storage cost. Although encoding/decoding enables reducing storage cost, the performance impact might offset the benefits in a system that aims at high performance as first priority. Interestingly, from a system perspective, the encoding process accepts an object value in input and produces a set of coding elements as output, each of which is effectively just a stream of bytes. The actual value represented by this stream of bytes by itself is unusable without other coding elements and the decode procedure. In other words, this stream of bytes can be immediately used by the key-value store to be transmitted over network, as well as stored on the data repository as is placed. Using this intuition, the key-value store is able to save the CPU intensive task of serializing/deserializing object values over the network. As a result, EPSI is able to compensate the overhead introduced by erasure coding with the capability of saving expensive network (de)serialization operations.

Another advantage of the way EPSI integrates the erasure coding technique in its read and write operations is the better network utilization when compared with existing solutions that do not use erasure coding for transaction processing, such as [21,57]. In fact, because of the encoding process, a single object value is split into multiple coding elements that need to be transferred independently to different nodes. Although this approach indeed increases the number of messages to be sent over the network, which contradicts the well-known benefits of batching [82], it becomes advantageous when the object size increases significantly, as illustrated below with a practical example.

Let us assume object values of 16 KB (Facebook in [7] reports value sizes between 4 KB and 10 KB). When broken down in six coding elements, each of them will have a size of 4 KB. This size is still large enough to well utilize the network infrastructure, but it is not so large to over stress the operating system's network stack of the receiving node. At the same time, since coding elements are sent in parallel, the latency to transmit 16 KB to a single node is divided into multiple smaller and parallel streams directed to different nodes.

As also empirically confirmed in our evaluation study in Section 7.6, EPSI performance scales better than competitors increasing the size of objects values stored in the data repository. This characteristic is particularly important for social applications because clients can produce posts whose content can be of arbitrarily high size given the heterogeneous nature of the content (e.g., image).

EPSI supports read-only transactions that never abort due to concurrency. This is enabled by the reliance on Walter as underlying concurrency control. With Walter, read operations always read from a consistent snapshot, which however can be not updated to the latest snapshot available. This decision matches the application target for EPSI, which is social applications. In these applications, it is understandable for a read operation to miss some posts written by recent write operations. On the positive side, this decision enables very low-latency reads because of the reduced protocol synchronization with write operations.

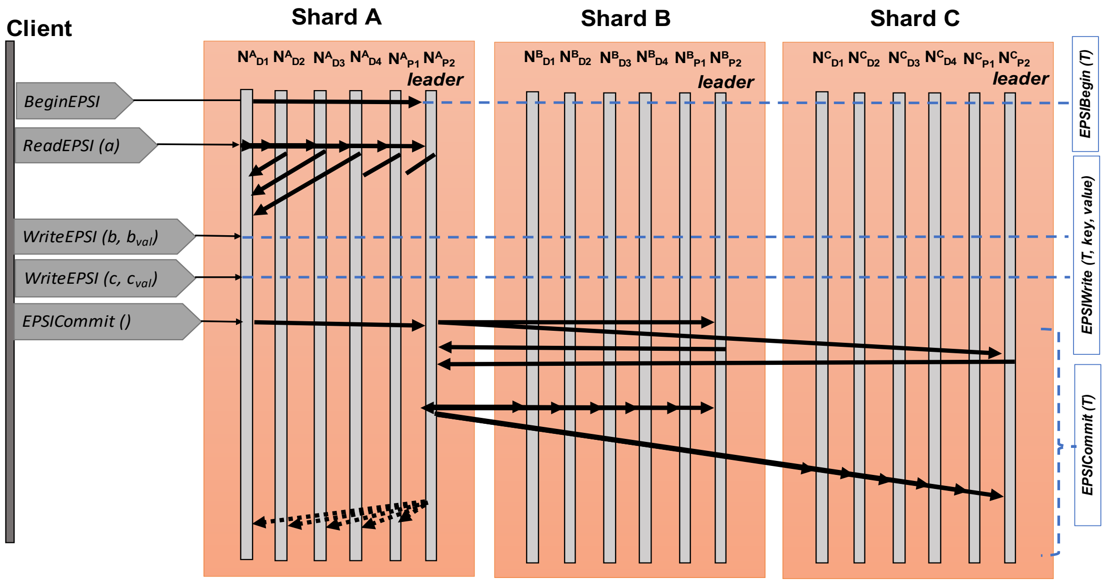


Figure 7.2: Example execution of a write transaction in EPSI. Dashed blue lines separate phases of transactional execution. Dashed arrows represent asynchronous messages. Overlapped arrows show parallel messages.

Update transactions are committed using the well-known Two-Phase Commit protocol (2PC) [85]. The parallel execution of concurrent transactions writing common objects is prevented by relying on the two-phase locking standard algorithm, in which locks are acquired during the 2PC `prepare phase`. In EPSI, the lock table is maintained by the leader node of the shard where the object is stored. Because of that, the 2PC process only involves all leader nodes of the shards where new versions of the written objects need to be installed on all the nodes of its corresponding shard. When all locks on the written objects have been acquired at the end of a successful `prepare` phase, new object values are encoded and the produced coding elements are distributed during the `decide` phase of the 2PC protocol to all nodes in the involved shards.

Figure 7.2 summarizes the overviewed protocol. In the pictured example, a client application performs a transaction T, which reads object a and writes objects b and c, each of them stored in the respective shard. Let us assume the client is connected to node N_{D1}^{A} from $ShardA$ and it uses the APIs listed in Table 7.1(a) to executes T. Each shard implements the Reed Solomon coding scheme where every block of data consists of four data coding elements and two parity coding element (i.e., $RS(4, 6)$), in which the crash of up to two nodes can be tolerated.

At first, since T is a write transaction, it should be forwarded to the leader of $ShardA$, namely N_{P2}^{A}, in order to establish the logical clock that will be used by T for its read operations. The read operations on a is sent to all nodes of $ShardA$ and its value is decoded as soon as 4 out of the 6 nodes in $ShardA$ return the related coding elements. For the sake of clarity, we defer to the protocol details in Section 7.5 the discussion about which version to be selected upon a read operation.

Since EPSI implements an optimistic concurrency control (OCC) design [90] similar to Walter, the write operations on objects b and c are buffered locally by T. The 2PC protocol is then executed by node N_{P2}^{A} to validate and commit T. The `prepare` phase of the 2PC involves the leaders of the two shards involved, namely N_{P2}^{B} and N_{P2}^{C}. On the other hand, the `decide` phase of the 2PC distributes the encoded coding elements by contacting all nodes of $ShardB$ and $ShardC$.

Later on, asynchronously, T's coordinator (N_{P2}^{A}) forwards the updated logical clock to the other nodes of $ShardA$ to establish the effect of T on the other nodes of $ShardA$.

7.5 Protocol Details

In the following, we show the details of EPSI protocol and architecture, including the steps of its transactional operations. As it is stated in Section 7.4, EPSI leverages Walter as the underlying distributed concurrency control to execute transactional operations on the erasure-coded key-value store.

7.5.1 Terminologies

In the following, we assume that the number of nodes inside each shard in EPSI is M and from this M nodes per a shard, N of them are `data nodes`, meaning they store data coding elements, and $M - N$ of them are called `parity nodes` due to preserving parity coding elements. Every node N belonging to $shard_i$ is represented as N_{shardi}. The leader of $shard_i$ is represented as $leader(shard_i)$. For any key k, $shard(k)$ returns the M nodes that store the M coding elements of k. If we have a set of keys (e.g., $keySet$), then $shards(keySet)$ returns all nodes that store each key in $keySet$ and $leaders(keySet)$ returns all nodes that store each key in $keySet$ and are the leaders of their corresponding shards. We also note that if node N maintains a key k in its corresponding data repository, then $k \in N$.

7.5.2 Transactional Begin Operation

Algorithm 14 shows the pseudo-code of the begin operation of transaction T in EPSI. Let us assume the begin operation is received by node N, which belongs to $shard_i$ (i.e., N_{shardi}). In order to do so, EPSI calls $EPSIBegin(T)$, shown in Algorithm 14.

Algorithm 14 The procedure of $EPSIBegin(T)$ in node N_{shard_i}

```
1: function EPSIBegin($Transaction\ T$)
2:     if ($T$ is a read-only) then
3:         $WalterBegin(T)$
4:     else
5:         if ($N_{shard_i}$ is the $leader(shard_i)$) then
6:             $WalterBegin(T)$
7:         else
8:             send $Forward[T]$ to $leader(shard_i)$
```

Lines 2-3 of Algorithm 14 show the steps taken by EPSI for starting a read-only transac-

Algorithm 15 Handling the procedure of $EPSIBegin(T)$ for the forwarded transaction T in $leader(shard_i)$

```
1: upon receive $Forward[T]$ do
2:     $WalterBegin(T)$
3: end
```

tion. If T is a read-only transaction, EPSI calls $WalterBegin(T)$ API (described in Lines 1-2 of Algorithm 13) in N_{shardi}. For update transactions, EPSI design choice is to only involve leaders' nodes before the commit decision is made. Therefore, T needs to be forwarded, by sending a `Forward` message, to the leader of the $shard_i$ (i.e., $leader(shard_i)$) (Line 8 of Algorithm 14) and when the `Forward` message is received by the leader, $WalterBegin(T)$ is invoked. As it is shown in Lines 1-3 of Algorithm 15, the leader of $shard_i$ is the coordinator of T for the rest of the execution.

7.5.3 Transactional Write Operation

Algorithm 16 represents the pseudo-code for a write operation in EPSI.

Algorithm 16 The procedure of write operation in EPSI

```
1: function EPSIWRITE($Transaction\ T$, $key\ k$, $value\ val$)
2:     $WalterWrite(k, val)$
```

Update transactions in EPSI implement lazy update [90] as Walter protocol mandates, meaning their written keys are not immediately visible and accessible at the time of the write operation; they are logged into the transactions write-set and become visible only at commit time.

The $WalterWrite(T)$ API of Walter is invoked by EPSI for write operation. This API buffers the written key by T with its corresponding values in T's private buffer (see Lines 5-6 of Algorithm 13 for $WalterWrite(T)$).

7.5.4 Transactional Read Operation

Algorithm 17 represents the details of steps taken by $EPSIRead$ API for reading object k. In EPSI, the read API of Walter is called (i.e., $WalterRead(k)$) which is detailed in Section 7.2.1, and presented in Algorithm 13, Lines 3-4.

Algorithm 17 Read Operation in the node N_i using EPSI

```
1: function EPSIRead($Transaction\ T$, $key\ k$)
2:     $targetNodes \leftarrow \{N_j : N_j \in shard(k)\}$
3:     $target \leftarrow Q : Q \subseteq targetNodes : |Q| \geq N$
4:     $WalterRead(k)$
5:     wait receive $ReadReturn[c_i]$ from all $N_j \in target$
6:     $val = DECODE(c_1, c_2, .., c_{|Q|})$
7:     return $val$
```

Since EPSI storage is based on the Reed Solomon technique (i.e., $RS(N, M)$), in order to read k, and rebuild the value of k from the returned coding elements, N number of coding elements need to be returned by the nodes in the target shard. After receiving a quorum of size M of *ReadReturn* messages, the original value can be constructed by invoking the decode function from the given Backblaze Reed Solomon implementation [108].

7.5.5 Transactional Commit Operation

Algorithm 18 represents the implementation of the *EPSICommit* API of EPSI, which is called when the commit phase of transaction T is started. EPSI never aborts read-only transactions, meaning read-only transactions can commit without going through any the commit phase (Lines 2-3 of Algorithm 18).

Algorithm 18 Commit of transaction T in $leader(shard_i)$ (i.e., N_i) using EPSI

```
1: function EPSIPrapre(Transaction T)
   // Check if T is a read-only transaction
2:     if ($T$ is a read-only) then
3:         return $true$
4:     $WalterPrepare(T,\ leaders(T.writeset)) \rightarrow output$
5:     if ($output$ is $Abort$) then
6:         $WalterAbort(T)$
7:     else
8:         $EPSICommit(T,\ shards(T.writeset) \cup N_i)$
9: function EPSICommit($Transaction\ T$, $Set\ participants$)
10:    for all ($< k, val > \in T.writeset$) do
11:        $[c_1, c_2, ..., c_M] = ENCODE(val)$
12:        $T.writeset \leftarrow T.writeset \setminus \{< k, val >\}$
13:        $T.writeset \leftarrow T.writeset \cup \{< k,\ [c_1, c_2, ..., c_M] >\}$
14:    $WalterCommit(T,\ participants)$
15:    $WalterPropagate(T,\ participants)$
```

For update transactions, EPSI calls *WalterCommit(T,participants* and *WalterPropagate(T,participants)* (discussed in Section 7.2.1 and detailed in Algorithm 13). The execution of 2PC is carried by T's coordinator, which is the leader of the shard where T's client connected for processing T. The 2PC involves all the leaders of the remote shards where T's written keys. Because of this design choice of EPSI, meaning that write transactions can be committed only by shard leaders, the size of the vector clocks used for ensuring the correctness of EPSI's protocol is equal to the number of shards in the distributed system, as opposed to the existing solutions (e.g., Walter [21], GMU [50] and SSS [32]) where vector clocks have a size equal to the total number of nodes in the system.

The first step of commit in EPSI is to call $WalterPrepare$ API, which performs the `prepare phase` of the 2PC protocol. If in this phase a write-conflicting transaction is detected, then T is aborted retried (Lines 4-6 of Algorithm 18). If the `prepare phase` is successfully completed by all the leaders of the involved shards, then $EPSICommit$ is called, which aims at establishing the written keys and the updated vector clocks of T in all the nodes of the involved shards, in addition to T's leader.

Since the data repository of each node store the coding elements, the written keys of T are needed to be encoded before the $WalterCommit$ API is called (Lines 10-13 of Algorithm 18). After this step $WalterCommit(T,$ *participants*) is called in Line 14 of Algorithm 18 to complete the commit process. $WalterCommit(T,$ *participants*) establishes T's updates and installs its commit vector clock into the involved nodes' vector clocks. When all the involved nodes apply T's updates, T successfully completes and its client is notified regarding T's completion.

EPSI calls $WalterPropagate(T$ *participants*) to propagate the effect of T's commit to the nodes that were not involved in the 2PC for committing T. This procedure propagates the information regarding the logical clock updated as a consequence of the commit of transaction T to all the nodes in the system. This is done to ensure that subsequent read operations issued by any nodes will be able to read the objects written by T.

7.6 Evaluation Study

7.6.1 Configuration and System Parameters

As testbed, we use 30 nodes of type m510 taken from the Cloudlab public platform [93]. Each node is a physical machine with eight-core Intel Xeon D-1548 at 2.0 GHz and 64GB of RAM. Nodes are interconnected using a 10Gb/s network. All the results are the average of 5 trials.

There are 10 application threads (i.e., clients) per node injecting transactions in the system in a closed-loop (i.e., a client issues a new request only when the previous one has returned). We configure each of the 10 clients to send objects of the same value size, which we vary from 12 bytes to 16 KB across different tests.

In terms of erasure coding configuration, we use $RS(4, 6)$ erasure coding in all the experiments. With this setting, we can tolerate up to two failures. For the competitors that do not use erasure coding but replication to provide fault tolerance and availability, we set the replication degree to three for a fair comparison with EPSI.

7.6.2 Benchmarks and Workload

The performance of EPSI, with its integration of Walter's concurrency control, has been evaluated using two well-known benchmarks for key-value stores, namely YCSB [92] and Retwis [68]. In both the benchmarks we varied the size of objects values from a small 12 bytes up to 16 KB and the size of objects keys is set to 4 bytes. In both benchmarks, transactions select keys to be accessed using a uniform distribution, which entails accesses might or might not be to the local data repository.

YCSB has been configured to produce two transaction profiles: one is read-only, with a configurable number of read operations; the other is update, in which two write operations are included, along with two read operations. Varying the number of read operations allows us to test performance with short and long read-only transactions. Retwis is a Twitter clone application with operations to post content on users' walls and read users' walls. There is a single table and each row is a key-value pair. We support two transactions, namely, (*i*) `PostTweet` and (*ii*) `GetTimeline`. A user can post a tweet to the social network via the

PostTweet transaction. The GetTimeline transaction returns the tweets from a user and his/her followers. The PostTweet profile updates 5 shared objects and reads 3 objects. The latter profile is read-only and accesses random number of objects between 1 and 10.

7.6.3 Competitors

We compare EPSI against the following competitors: Cocytus [57], Walter [21] and 2PC-baseline. All these competitors offer transactional semantics over key-value APIs. EPSI and its competitors have been developed from the ground up in Java, sharing the same underlying infrastructural components, such as the implementation of the communication layer. This has been done in order to provide them with fair and comparable optimizations.

Cocytus [57] is a transactional key-value store that provides Serializability [37]. Cocytus stores objects into shards and each node in Cocytus runs both parity processes and data processes in order to make all the nodes busy in both update-intensive or read-mostly workload. A transaction reads/modifies the whole data block by accessing data processes. Update transactions generate the parity coding elements corresponding to their updated data blocks in order to distribute them among parity processes.

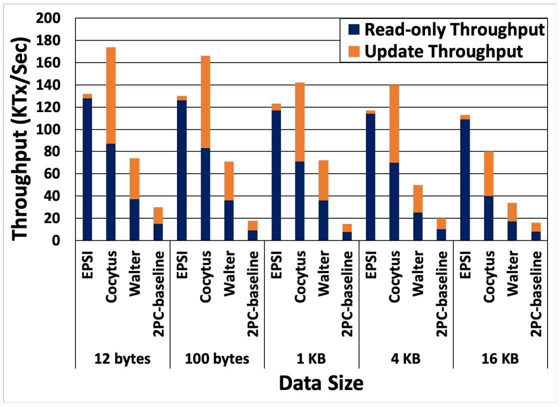
2PC-baseline is a serializable key-value store and a single-version system where all transactions execute optimistically and rely on the Two-Phase Commit protocol to commit both update and read-only transactions.

All competitors have been enhanced with the one-shot read technique [109]. With that, all read operations of both read-only and write transactions are issues at the same time at the beginning of the transaction execution. This is an optimization allowed when the application is aware of the target objects to be read in advance.

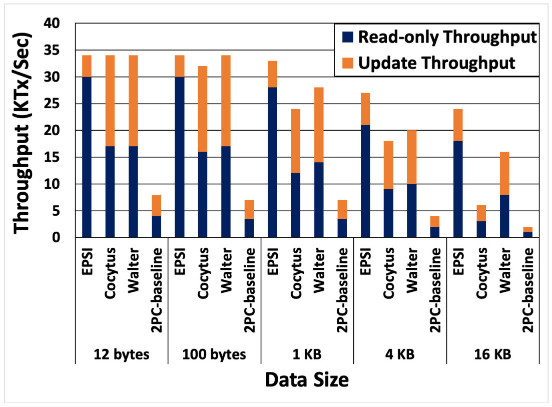
7.6.4 Experimental Results

YCSB

Workload of 50% read-only transactions. We first analyze the results using a low-contention case, in which 1M objects have been deployed in the distributed data repository and the measured abort percentage is below 6%. Figure 7.3 includes the throughput results, with the breakdown between read-only and update throughput, using 50% read-only workload. The different clusters in each plot show results using different sizes for object value, in the range 12 bytes up to 16 KB. Also, read-only transactions perform 2 read operations, in Figure 7.3(a), and 32 read operations, in Figure 7.3(b).



(a) 2 read operations inside read-only transactions.



(b) 32 read operations inside read-only transactions.

Figure 7.3: Throughput of YCSB varying the size of object values and 50% read-only transactions.

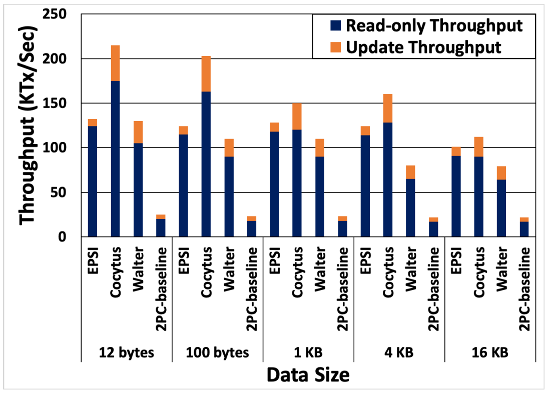
In Figure 7.3(a), the read-only profile reads two objects and the update profile reads and updates two objects. Under 50% read-only workload, Cocytus outperforms EPSI, by up to 25%, when the object size is up to 4 KB. This is mainly because of the higher update throughput of Cocytus. In fact, in Cocytus every update transaction can be served by any node in the system. On the other hand, EPSI allows only shard leaders to process update transactions. As a result, the write throughput of EPSI is generally lower than Cocytus's; and this observation is confirmed by the trends on other plots as well. It is also important to note that, in all configurations, EPSI's read-only throughput is higher than any other competitor. We attribute this advantage to the capability of exploiting quorum reads, which prevent slow down due to possibly overloaded nodes.

For the case of 16 KB, EPSI outperforms Cocytus by 40%. This is because of two reasons in the presence of a high percentage of update transactions. First, Cocytus produces and needs to transmit over the network an excessive amount of parity coding elements. Second, the large values are transferred to a single node while in EPSI, these large values are split across nodes in the shard, which optimizes network utilization and provides a more balanced load across nodes.

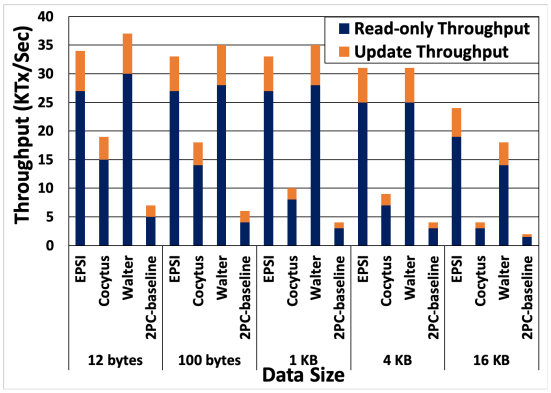
With respect to the other competitors, EPSI is faster than Walter, by up to 3×, and faster than 2PC-baseline, by up to 7×. The speedup with respect to Walter is due to the fact that reads can be served by a quorum, especially in the case of high value sizes, and because with a single leader per shard, update transactions are aborted less frequently since the leaders' reading snapshot is always more updated than any other node, as the case of Walter.

In Figure 7.3(b), the read-only profile reads 32 objects and the update profile reads and updates 2 objects. EPSI outperforms Cocytus up to 4×, when object size is greater than 12 bytes. The reason is related to the higher network utilization of Cocytus given the increased read operations. In fact, EPSI can serve them exploiting quorums, balancing the load across nodes, while Cocytus is forced to read from a single node per object. The gap between EPSI and Walter decreases to up to 50% with respect to the gap in Figure 7.3(a). EPSI is faster than 2PC-baseline by up to 12× since, when the number of read operations increases, 2PC-baseline needs more network communications for read-only transactions compared to the case of small read-only transactions (i.e., Figure 7.3(a)).

Workload of 80% read-only transactions. In Figure 7.4 we compare the throughput of EPSI against the rest of the competitors using a workload made of 80% read-only transactions.



(a) 2 read operations inside read-only transactions.



(b) 32 read operations inside read-only transactions.

Figure 7.4: Throughput of YCSB varying the size of object values and 80% read-only transactions.

In Figure 7.4(a), with short transactions, EPSI's performance matches Cocytus's performance only for large object size (i.e., 16 KB). In all other cases, performance is lower. The reason for such a trend is twofold: (*i*) number of read operations is small and therefore reading from a quorum of nodes does not carry benefits, and; (*ii*) with only 20% update transactions the network overhead of transmitting parity coding elements is negligible.

When the size of read-only transactions increases to 32, results shown in Figure 7.4(b), the advantages of EPSI's concurrency control and erasure coding technique kick in, therefore allowing up to 6× performance improvement over Cocytus.

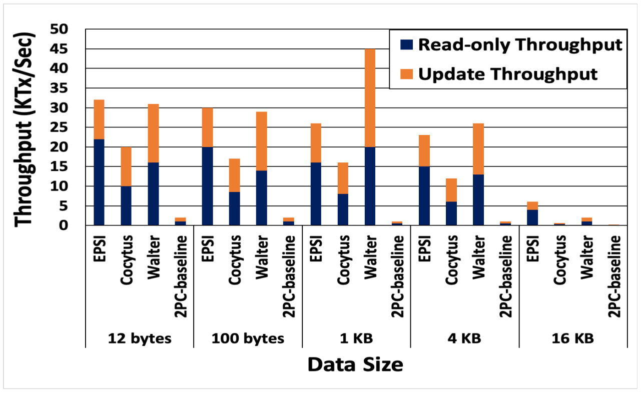
Analyzing the results of EPSI against the other competitors, we found that, in the case of small read-only transactions 7.4(a), EPSI's throughput is higher than Walter, by up to 55%. In the case of having long read-only transactions, Figure 7.4(b), Walter is slightly faster than EPSI (up to 8%) for objects less than 1 KB. The latter configuration represents Walter's performance sweet spot (i.e., long read-only intensive workload).

EPSI outperforms 2PC-baseline using both small and long read-only transactions. EPSI is up to 12× faster than 2PC-baseline for long read-only transactions (see Figure 7.4(b)) due to the additional network communications needed to commit those types of transactions.

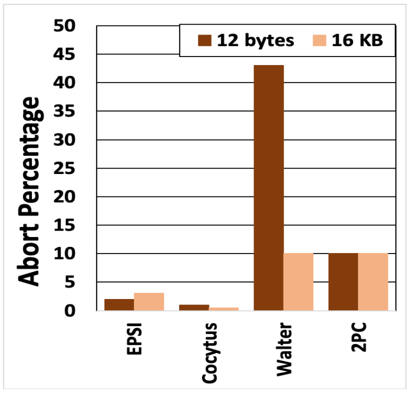
Increasing Contention. Figure 7.5 shows the behavior of EPSI and the competitors when 10K objects are distributed among 5 shards (i.e., 2K per shard). In this scenario, contention increases because there are 50% update transactions, a lower number of objects, and long read-only transactions.

Figure 7.5(a) presents the measured throughput and Figure 7.5(b) quantifies the measured abort rate using these settings and by varying the size of object values.

Generally, the best performing competitor with increased contention, except for the case of large object size, is Walter followed by EPSI. This is mainly due to the single-leader constraint of EPSI. In fact, the update throughput of Walter is almost twice the update throughput for EPSI. Although Figure 7.5(b) reports Walter with a higher abort rate than EPSI, it's update transaction commit latency is much lower, which allows transactions to still commit quickly even after getting aborted. The benefits of EPSI kick in at object size 16 KB, where it outperforms Walter by 3×.



(a) Throughput of YCSB varying the size of object value.



(b) Abort rate of YCSB varying the size of object value.

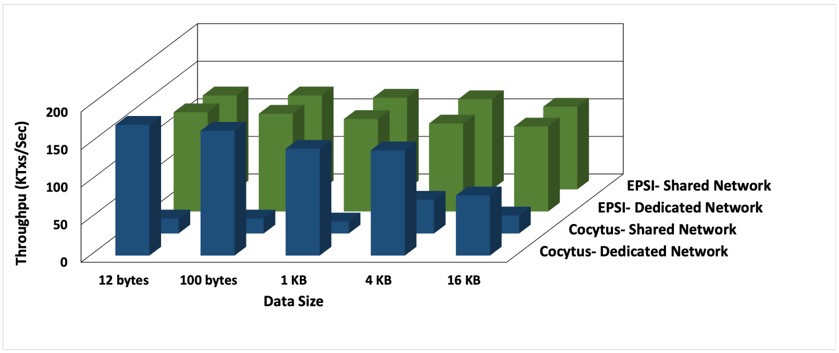
Figure 7.5: Throughput and Abort rate of YCSB varying the size of object values and 50% read-only transactions under the high contention (i.e., 10K keys).

The performance trend of Cocytus in Figure 7.5(a) is similar to the case of low contention with long read-only transactions, with the aggravation of higher abort rate due to contention.

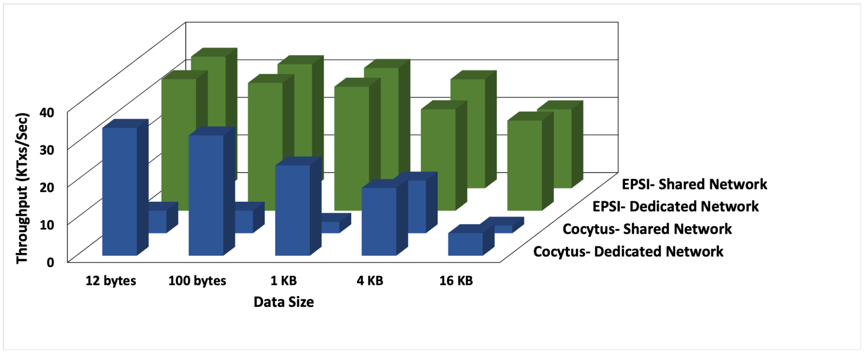
2PC-baseline is the worst performing competitor because of its high abort rate, exacerbated by the replication degree set to three, and the need of coordinating the commit phase of read-only transactions, as opposed to Walter and EPSI.

Erasure Coding and Network Utilization. Figures 7.6 and 7.7 show the impact of the different ways EPSI and Cocytus use to integrate erasure coding, for 50% and 80% read-only workload when the size of read-only transactions increases from 2 (i.e., small read-only) to 32 (i.e., long read-only). We measure that by changing the network configuration between nodes, from a dedicated network to a network shared with other applications. With a dedicated network, the entire bandwidth is reserved to support the functionality offered by EPSI and Cocytus. On the other hand, with a shared network, foreign workloads can influence network performance, penalizing the competitor that requires long transmission sessions.

Figure 7.6 shows the throughput of EPSI and Cocytus by varying the network configuration from dedicated to shared, using 50% read-only workload. Using the shared network, Cocytus throughput is between 4× and 8× slower than the throughput when using the dedicated network, as is shown in Figure 7.6(a). The performance of EPSI for each data size is



(a) 2 read operations inside read-only transactions.



(b) 32 read operations inside read-only transactions.

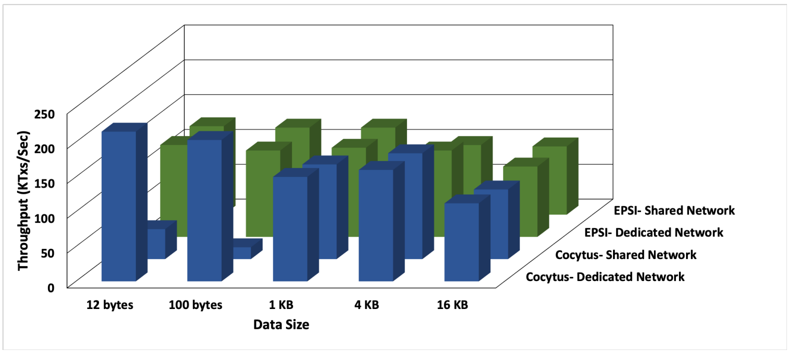
Figure 7.6: Throughput of YCSB varying the size of object values and 50% read-only transactions using both a dedicated and a shared network in EPSI and Cocytus.

almost the same using both network configurations.

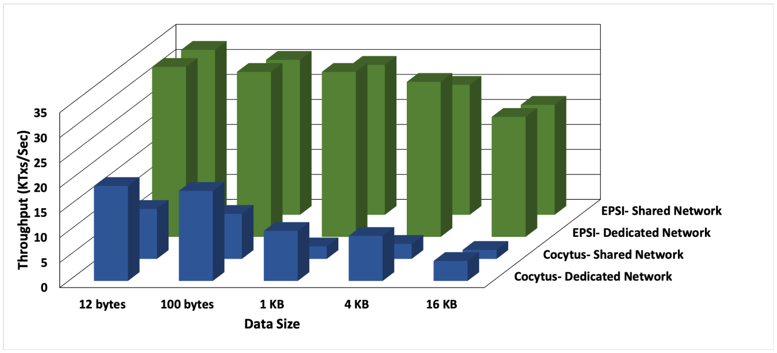
In Figure 7.7 the observed throughput of EPSI and Cocytus is shown using 80% read-only workload and the different size of read-only transactions. Summarizing, EPSI's throughput does not change using different network configurations.

In Figure 7.7(a), the performance degradation of Cocytus is 10× than using a dedicated network, for small object value sizes (e.g., 100 bytes). For larger object value sizes (e.g., 16 KB), Cocytus's throughput does not change. Figure 7.7(b) represents the performance difference with the two network configurations using long read-only transactions. In this case, Cocytus is 2× slower for large size of object values (i.e., 16 KB).

Before going into the details of our analysis, we can summarize our findings by saying that, generally, the combination of erasure coding transactional accesses and the exploitation of quorum read in EPSI makes EPSI less sensible to the network configuration. As a result, EPSI shows a more robust performance than Cocytus, in which transactional operations



(a) 2 read operations inside read-only transactions.



(b) 32 read operations inside read-only transactions.

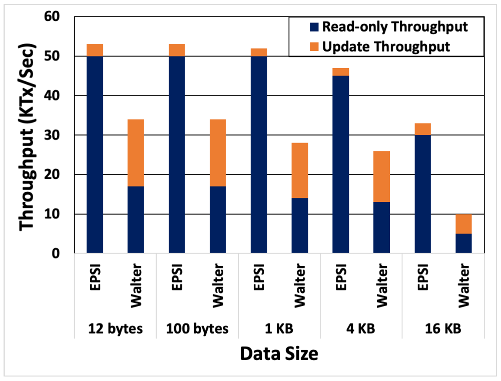
Figure 7.7: Throughput of YCSB varying the size of object values and 80% read-only transactions using both a dedicated and a shared network in EPSI and Cocytus.

access non-erasure coded values. Results confirm this claim under different workloads.

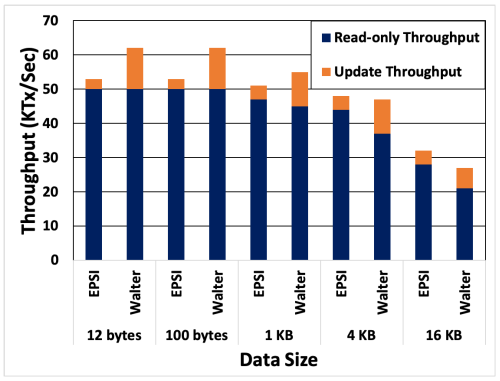
Retwis

Retwis benchmark is a social networking application, which we configure to produce 50% and 80% read-only workload. As opposed to YCSB, in Retwis update transactions do not read and write the same keys, therefore they do not need to be validated and their expected abort rate is lower. Because of that, we focus on the competitors ensuring PSI (EPSI and Walter) since they are expected to provide significantly better performance than competitors enforcing Serializability (Cocytus and 2PC-baseline).

Figure 7.8 plots the results of Retwis benchmark. In Figure 7.8(a), EPSI outperforms Walter by up to 3× for all the object size. This is because Walter incurs in a higher abort rate than EPSI using 50% read-only workload. EPSI's abort rate is less than 2% for all the objects size. Walter's abort rate is 10% when the object size is equal to 12 bytes and is 40%



(a) 50%



(b) 80%

Figure 7.8: Throughput of Retwis varying the size of data using EPSI and Walter.

when the object size is 16 KB.

In Figure 7.8(b), EPSI outperforms Walter only with 16 KB objects size. This is due to the exploitation of quorum reads, which lead to better network utilization.

Chapter 8

Conclusions and Future Work

8.1 Conclusion Remarks

Transactional data repositories certainly represent one of the most important building blocks to develop applications and services that manipulate data in the presence of concurrent client requests. In order to make these repositories able to process realistic workloads, which often involve a massive amount of interacting users, they need to maintain high performance, scalability, and fault tolerance. The research included in this dissertation has a twofold aim, improving *i)* system programmability, which is the capability of a system to provide applications with features that are easy to use; and *ii)* the cost of fault tolerance, namely the space overhead needed to ensure normal operation despite the presence of failures, in distributed transactional systems.

We address system programmability by focusing on two well-known correctness levels, external consistency and Parallel Snapshot Isolation (PSI), and improve the freshness of their read operations without the need of relying on special purpose hardware. The outcome of our innovations allows those transactional systems to be easily adopted and extended by both academia and industry.

The first contribution included in this dissertation and presented in Chapter 4, is SSS. SSS is a transactional key-value store that provides external consistency. SSS ensures high programmability because it prevents incorrect results (e.g., "unrealistic" results or obsolete data) from being propagated to the application (i.e., clients) by preserving external consistency. SSS novelty is to use the snapshot-queuing technique for propagating established serialization orders among concurrent transactions.

Our second contribution is FPSI, presented in Chapter 5. FPSI is a transactional system that addressed an open research question on how to identify a well-specified level of data freshness in Parallel Snapshot Isolation. FPSI was built upon Walter, as a standard implementation of PSI. Walter achieves high performance at the cost of reading arbitrarily old data in case transactions access objects remotely. The novel concurrency control at the core of FPSI allows its abort-free read-only transactions to access the latest version of objects upon their first contact to a node, at no significant performance degradation compared to Walter.

The third contribution of this dissertation formalizes the data freshness of a transactional system as ordering constraint among transactions. In our proposed model, we delivered a condition that can be used to verify the safety of transactional executions in the presence of programmer-provided external dependency (or application invariant). Our unified model overcomes the lack of existing formulations on verifying the correctness of distributed concurrency control implementations based on their data freshness. The model is provided in Chapter 6.

Lastly, in Chapter 7, we showed our discoveries regarding the practicality of deploying a high-performance PSI concurrency control when the storage space is optimized using the well-known erasure coding technique. We presented a novel sharded key-value store named EPSI, which serves transactional operations by accessing erasure coded data, during the normal system functioning and not during recovery upon failures, as traditionally done. The direct consequence of EPSI's erasure coded design is that read operations can be served reading from a quorum of nodes, which improves system performance when objects size is large due to an improved load balancing across nodes.

8.2 Future Works

External consistency assumed that clients outside the system are prohibited from interacting with each other while waiting for a transaction execution to be concluded. There are alternative models in which such restriction is relaxed, therefore clients have the freedom to interact with each other while waiting for their transactions to be completed (e.g., as the case of transactions asynchronously scheduled for execution). In this model, if a transaction completes, a client can inform another waiting client about the outcome of its operation. More formally, the real-time order relation among these two transactions can now be established although these transactions are concurrent in the system.

In order to support the latter model, we are interested in extending SSS to avoid forward anti-dependency to be propagated to clients using an extension of the snapshot-queuing technique. This way SSS's distributed concurrency control will be able to guarantee commit-order [110], a consistency level in which the transaction serialization order matches the order in which transactions commit.

We are also interested in removing the read-only anomaly [44, 46, 96] allowed by Parallel Snapshot Isolation to further improve FPSI. The read-only anomaly prevents a read-only transaction from reading a snapshot that satisfies the Serializability correctness criterion. We propose to remove this anomaly by relaxing the real-time order constraints among two transactions if a concurrent and conflicting write transaction executes. By capturing this overlapped execution, we will be able to force a read-only transaction to access the previous snapshot of the shared state to avoid the materialization of the preconditions that enable the read-only anomaly.

Using the sharded structure of EPSI, update transactions are forwarded to the shard leader in order to be executed. Motivated by the promising performance gain obtained of the current EPSI design, we plan to extend EPSI to support multiple leaders per shard. This extension addresses one of the current limitations of EPSI, which is providing a lower throughput than Walter. Having more nodes withing a shard that can process update transactions can improve update transactions' throughout.

Bibliography

[1] M. Herlihy and V. Luchangco, "Distributed computing and the multicore revolution," *ACM SIGACT News*, vol. 39, no. 1, pp. 62–72, 2008.

[2] M. K. Aguilera, N. Ben-David, I. Calciu, R. Guerraoui, E. Petrank, and S. Toueg, "Passing messages while sharing memory," in *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, 2018, pp. 51–60.

[3] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian, "The end of slow networks: It's time for a redesign," *arXiv preprint arXiv:1504.01048*, 2015.

[4] A. Kalia, M. Kaminsky, and D. G. Andersen, "Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 185–201.

[5] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "Farm: Fast remote memory," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 401–414.

[6] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure coding in windows azure storage," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIXATC 12)*, 2012, pp. 15–26.

[7] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab *et al.*, "Scaling memcache at facebook," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 385–398.

[8] G. Huang, X. Cheng, J. Wang, Y. Wang, D. He, T. Zhang, F. Li, S. Wang, W. Cao, and Q. Li, "X-engine: An optimized storage engine for large-scale e-commerce transaction processing," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 651–665.

[9] X. Qin, "Delayed consistency model for distributed interactive systems with real-time continuous media," *Journal of Software*, vol. 13, no. 6, pp. 1029–1039, 2002.

[10] M. Rajashekhar and Y. Yue, "Twemcache: Twitter memcached," 2012.

[11] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "Farsite: Federated, available, and reliable storage for an incompletely trusted environment," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 1–14, 2002.

[12] N. Schiper, P. Sutra, and F. Pedone, "P-store: Genuine partial replication in wide area networks," in *2010 29th IEEE Symposium on Reliable Distributed Systems.* IEEE, 2010, pp. 214–224.

[13] D. Serrano, M. Patiño-Martínez, R. Jiménez-Peris, and B. Kemme, "Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation," in *13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007).* IEEE, 2007, pp. 290–297.

[14] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[15] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: fast distributed transactions for partitioned database systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data.* ACM, 2012, pp. 1–12.

[16] A. Silberschatz, H. F. Korth, and S. Sudarshan, "Introduction to data base management system."

[17] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li, “Transaction chains: achieving serializability with low latency in geo-distributed storage systems,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 276–291.

[18] Cockroach Labs, “CockroachDB ,” 2017, https://github.com/cockroachdb/cockroach.

[19] S. Bhuiyan, M. Zheludkov, and T. Isachenko, *High Performance In-memory Computing with Apache Ignite.* Lulu.com, 2017.

[20] “HAZELCAST, The Operational In-Memory Computing Platform,” https://hazelcast.com.

[21] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, “Transactional storage for geo-replicated systems,” in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles.* ACM, 2011, pp. 385–400.

[22] J. A. Kreibich, *Using SQLite*, 1st ed. O’Reilly Media, Inc., 2010.

[23] N. Shamgunov, “The memsql in-memory database system,” in *Proceedings of the 2nd International Workshop on In Memory Data Management and Analytics, IMDM 2014, Hangzhou, China, September 1, 2014.*, J. J. Levandoski and A. Pavlo, Eds., 2014.

[24] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems.* Addison-wesley Reading, 1987, vol. 370.

[25] P. A. Bernstein and N. Goodman, *A sophisticate’s introduction to distributed database concurrency control.* Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1982.

[26] W.-T. K. Lin and J. Nolte, “Basic timestamp, multiple version timestamp, and two-phase locking.” in *VLDB*, vol. 83, 1983, pp. 109–119.

[27] H. Kung and J. T. Robinson, “On optimistic methods for concurrency control,” *Readings in database systems*, pp. 209–215, 1994.

[28] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's Globally Distributed Database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, pp. 8:1–8:22, Aug. 2013.

[29] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. Ports, "Building consistent transactions with inconsistent replication," in *Proceedings of the 25th Symposium on Operating Systems Principles.* ACM, 2015, pp. 263–278.

[30] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li, "Extracting more concurrency from distributed transactions." in *OSDI*, vol. 14, 2014, pp. 479–494.

[31] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson, "Scalable consistency in scatter," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles.* ACM, 2011, pp. 15–28.

[32] M. J. Kishi, S. Peluso, H. F. Korth, and R. Palmieri, "SSS: scalable key-value store with external consistent and abort-free read-only transactions," in *ICDCS*, 2019, pp. 589–600.

[33] M. M. Saad, M. J. Kishi, S. Jing, S. Hans, and R. Palmieri, "Processing transactions in a predefined order," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019, pp. 120–132.

[34] Z. Chen, A. Hassan, M. J. Kishi, J. Nelson, and R. Palmieri, "Hats: Hardware-assisted transaction scheduler," in *23rd International Conference on Principles of Distributed Systems (OPODIS 2019).* Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[35] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ansi sql isolation levels," *ACM SIGMOD Record*, vol. 24, no. 2, pp. 1–10, 1995.

[36] A. Adya and B. H. Liskov, "Weak consistency: a generalized theory and optimistic implementations for distributed transactions," Ph.D. dissertation, Massachusetts Institute of Technology, Dept. of Electrical Engineering and . . . , 1999.

[37] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Computing Surveys (CSUR)*, vol. 13, no. 2, pp. 185–221, 1981.

[38] T. Harris and S. Jones, "Transactional memory with data invariants," 2006.

[39] M. Javidi Kishi, A. Hassan, and R. Palmieri, "Brief announcement: On the correctness of transaction processing with external dependency," in *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[40] D. K. Gifford, "Information storage in a decentralized computer system," Ph.D. dissertation, Stanford University, 1981.

[41] A. Cerone and A. Gotsman, "Analysing snapshot isolation," *Journal of the ACM (JACM)*, vol. 65, no. 2, p. 11, 2018.

[42] M. Pratt and P. McElroy, "Oracle9i replication," *White paper, June*, 2001.

[43] K. L. Tripp, "Sql server 2005 beta ii snapshot isolation," 2005.

[44] D. R. Ports and K. Grittner, "Serializable snapshot isolation in postgresql," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1850–1861, 2012.

[45] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, 2012, pp. 53–64.

[46] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha, "Making snapshot isolation serializable," *ACM Transactions on Database Systems (TODS)*, vol. 30, no. 2, pp. 492–528, 2005.

[47] J. Du, S. Elnikety, and W. Zwaenepoel, “Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks,” in *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on.* IEEE, 2013, pp. 173–184.

[48] S. Peluso, P. Romano, and F. Quaglia, “SCORe: A scalable one-copy serializable partial replication protocol,” in *Middleware 2012*, 2012, pp. 456–475.

[49] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, “The primary-backup approach,” *Distributed systems*, vol. 2, pp. 199–216, 1993.

[50] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues, “Gmu: Genuine multiversion update-serializable partial data replication,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 10, pp. 2911–2925, 2016.

[51] M. Mohamedin, M. J. Kishi, and R. Palmieri, “Shield: A middleware to tolerate cpu transient faults in multicore architectures,” in *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA).* IEEE, 2017, pp. 1–9.

[52] L. Lamport, “The part-time parliament,” in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 277–317.

[53] S. Hirve, R. Palmieri, and B. Ravindran, “Archie: a speculative replicated transactional system,” in *Proceedings of the 15th international Middleware Conference*, 2014, pp. 265–276.

[54] P. A. Alsberg and J. D. Day, “A principle for resilient sharing of distributed resources,” in *Proceedings of the 2nd international conference on Software engineering*, 1976, pp. 562–570.

[55] I. S. Reed and G. Solomon, “Polynomial codes over certain finite fields,” *Journal of the society for industrial and applied mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[56] D. A. Patterson, G. Gibson, and R. H. Katz, “A case for redundant arrays of inexpensive disks (raid),” in *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, 1988, pp. 109–116.

[57] H. Chen, H. Zhang, M. Dong, Z. Wang, Y. Xia, H. Guan, and B. Zang, "Efficient and available in-memory kv-store with hybrid erasure coding and replication," *ACM Transactions on Storage (TOS)*, vol. 13, no. 3, pp. 1–30, 2017.

[58] M. M. Yiu, H. H. Chan, and P. P. Lee, "Erasure coding for small objects in in-memory kv storage," in *Proceedings of the 10th ACM International Systems and Storage Conference*, 2017, pp. 1–12.

[59] B. Fan, D. G. Andersen, and M. Kaminsky, "Memc3: Compact and concurrent memcache with dumber caching and smarter hashing," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 371–384.

[60] K. M. Konwar, N. Prakash, N. Lynch, and M. Médard, "A layered architecture for erasure-coded consistent distributed storage," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 2017, pp. 63–72.

[61] L. Cheng, Y. Hu, and P. P. Lee, "Coupling decentralized key-value stores with erasure coding," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 377–389.

[62] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," 2010.

[63] M. S. Ardekani, P. Sutra, and M. Shapiro, "Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems," in *2013 IEEE 32nd International Symposium on Reliable Distributed Systems.* IEEE, 2013, pp. 163–172.

[64] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel, "Orbe: Scalable causal consistency using dependency matrices and physical clocks," in *Proceedings of the 4th annual Symposium on Cloud Computing.* ACM, 2013, p. 11.

[65] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro, "Cure: Strong semantics meets high availability and low latency," in

Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on. IEEE, 2016, pp. 405–414.

[66] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, "Gentlerain: Cheap and scalable causal consistency with physical clocks," in *Proceedings of the ACM Symposium on Cloud Computing.* ACM, 2014, pp. 1–13.

[67] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, "The end of an architectural era:(it's time for a complete rewrite)," in *Proceedings of the 33rd international conference on Very large data bases.* VLDB Endowment, 2007, pp. 1150–1160.

[68] C. Leau, "Spring data redis-retwis-j," 2013.

[69] F. Mattern *et al.*, *Virtual time and global states of distributed systems.* Citeseer, 1988.

[70] M. Mohamedin, S. Peluso, M. J. Kishi, A. Hassan, and R. Palmieri, "Nemo: Numa-aware concurrency control for scalable transactional memory," in *Proceedings of the 47th International Conference on Parallel Processing*, 2018, pp. 1–10.

[71] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Computing Surveys (CSUR)*, vol. 36, no. 4, pp. 372–421, 2004.

[72] T. Landes, "Dynamic vector clocks for consistent ordering of events in dynamic distributed applications." in *PDPTA*, 2006, pp. 31–37.

[73] X. Wang, J. Mayo, W. Gao, and J. Slusser, "An efficient implementation of vector clocks in dynamic systems." in *PDPTA*, 2006, pp. 593–599.

[74] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi, "Low-latency multi-datacenter databases using replicated commit," *Proceedings of the VLDB Endowment*, vol. 6, no. 9, pp. 661–672, 2013.

[75] J. Cowling and B. Liskov, "Granola: low-overhead distributed transaction coordination," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIXATC 12)*, 2012, pp. 223–235.

[76] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[77] C. Binnig, S. Hildenbrand, F. Färber, D. Kossmann, J. Lee, and N. May, "Distributed snapshot isolation: global transactions pay globally, local transactions pay locally," *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 23, no. 6, pp. 987–1011, 2014.

[78] S. Elnikety, F. Pedone, and W. Zwaenepoel, "Database replication using generalized snapshot isolation," in *SRDS*, 2005, pp. 73–84.

[79] B. Fitzpatrick, "Distributed caching with memcached," *Linux journal*, vol. 124, 2004.

[80] J. Zawodny, "Redis: Lightweight key/value store that goes the extra mile," *Linux Magazine*, vol. 79, no. 8, pp. 1–10, 2009.

[81] Y. L. Chen, S. Mu, J. Li, C. Huang, J. Li, A. Ogus, and D. Phillips, "Giza: Erasure coding objects across global data centers," in *2017 USENIX Annual Technical Conference (USENIXATC 17)*, 2017, pp. 539–551.

[82] S. Li, Q. Zhang, Z. Yang, and Y. Dai, "Bcstore: Bandwidth-efficient in-memory kv-store with batch coding," *Proc. of IEEE MSST*, 2017.

[83] W. Litwin and T. Schwarz, "Lh* rs: A high-availability scalable distributed data structure using reed solomon codes," in *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, 2000, pp. 237–248.

[84] C. Lai, S. Jiang, L. Yang, S. Lin, G. Sun, Z. Hou, C. Cui, and J. Cong, "Atlas: Baidu's key-value storage system for cloud data," in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2015, pp. 1–14.

[85] J. Gray and L. Lamport, "Consensus on transaction commit," *ACM Transactions on Database Systems (TODS)*, vol. 31, no. 1, pp. 133–160, 2006.

[86] L. Lamport, "Fast paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.

[87] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *IEEE transactions on information theory*, vol. 56, no. 9, pp. 4539–4551, 2010.

[88] J. C. Chan, Q. Ding, P. P. Lee, and H. H. Chan, "Parity logging with reserved space: Towards efficient updates and recovery in erasure-coded clustered storage," in *12th USENIX Conference on File and Storage Technologies (FAST 14)*, 2014, pp. 163–176.

[89] S. Peluso, R. Palmieri, P. Romano, B. Ravindran, and F. Quaglia, "Disjoint-access parallelism: Impossibility, possibility, and cost of transactional memory implementations," in *PODC*, 2015, pp. 217–226.

[90] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 18–32.

[91] M. J. Kishi, S. Peluso, H. Korth, and R. Palmieri, "SSS: Scalable Key-Value Store with External Consistent and Abort-free Read-only Transactions," Lehigh University, Tech. Rep., 2019, URL: http://sss.cse.lehigh.edu/files/pubs/TR-sss.pdf.

[92] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[93] R. Ricci, E. Eide, and C. Team, "Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications," *; login:: the magazine of USENIX & SAGE*, vol. 39, no. 6, pp. 36–38, 2014.

[94] "CloudLab Clemson," 2017, http://docs.cloudlab.us/hardware.html.

[95] M. Pratt and P. McElroy, "Oracle9i replication," *White paper, June*, 2001.

[96] A. Fekete, E. O'Neil, and P. O'Neil, "A read-only transaction anomaly under snapshot isolation," *ACM SIGMOD Record*, vol. 33, no. 3, pp. 12–14, 2004.

[97] T. P. P. Council, "tpc-c benchmark, revision 5.11," 2010.

[98] K. Daudjee and K. Salem, "Lazy database replication with ordering guarantees," in *ICDE*. IEEE, 2004, pp. 424–435.

[99] F. Klein, K. Beineke, and M. Schöttner, "Memory management for billions of small objects in a distributed in-memory storage," in *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2014, pp. 113–122.

[100] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *ACM SIGOPS operating systems review*, vol. 41, no. 6. ACM, 2007, pp. 205–220.

[101] A. Cassandra, "Apache cassandra," *Website. Available online at http://planetcassandra. org/what-is-apache-cassandra*, vol. 13, 2014.

[102] C. Mitchell, Y. Geng, and J. Li, "Using one-sided rdma reads to build a fast, cpu-efficient key-value store," in *2013 USENIX Annual Technical Conference (USENIX-ATC 13)*, 2013, pp. 103–114.

[103] H. Meir, D. Basin, E. Bortnikov, A. Braginsky, Y. Gottesman, I. Keidar, E. Meir, G. Sheffi, and Y. Zuriel, "Oak: a scalable off-heap allocated key-value map," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 17–31.

[104] C. Aniszczyk, "Caching with twemcache," *Twitter Blog, Engineering Blog*, pp. 1–7, 2012.

[105] K. M. Konwar, N. Prakash, M. Médard, and N. Lynch, "Fast lean erasure-coded atomic memory object," in *23rd International Conference on Principles of Distributed Systems (OPODIS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[106] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin, "All about eve: execute-verify replication for multi-core servers," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 237–250.

[107] V. Estrada-Galinanes, E. Miller, P. Felber, and J.-F. Pâris, "Alpha entanglement codes: practical erasure codes to archive data in unreliable environments," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 183–194.

[108] "Backblaze Java Code," https://github.com/Backblaze/JavaReedSolomon, 2017.

[109] H. Lu, C. Hodsdon, K. Ngo, S. Mu, and W. Lloyd, "The snow theorem and latency-optimal read-only transactions." in *OSDI*, 2016, pp. 135–150.

[110] D. K. Gifford, "Information storage in a decentralized computer system," Ph.D. dissertation, Stanford University, 1981.

Vita

Masoomeh Javidi Kishi received her B.Sc. degree in Computer Engineering from Iran University of Science and Technology, Tehran, Iran, in 2010 and her M.Sc. in Computer Engineering from Islamic Azad University, Science and Research Branch, Tehran, Iran, in 2012. In 2014, Masoomeh joined the Computer Science Department at Virginia Polytechnic Institute and State University as a Ph.D. student and in 2017 she transferred her Ph.D degree to Computer Science and Engineering Department at Lehigh University. She has published in many venues including ICDCS, DISC, PPoPP, ICPP, OPODIS, NCA and been awarded ICDCS and DISC conferences' travel award by NSF and IEEE Computer Society's Committee on Distributed Processing.