A FAST IMPLEMENTATION OF PARALLEL SNAPSHOT ISOLATION

UNA IMPLEMENTACIÓN RÁPIDA DE PARALLEL SNAPSHOT ISOLATION

Borja Arnau de Régil Basáñez

Trabajo de Fin de Grado del Grado en Ingeniería Informática

Facultad de Informática,
Universidad Complutense de Madrid

Junio 2020

Director: Maria Victoria López López
Co-director: Alexey Gotsman

Contents

Acknowledgements

To my advisors Maria Victoria López López at Universidad Complutense de Madrid, and Alexey Gotsman and Manuel Bravo at the IMDEA Software Institute, for their guidance and support, and for giving me the opportunity to work along them.

I also thank Christopher Meiklejohn, who allowed me to work with him, and gave me the opportunity to discover the IMDEA Software Institute. To my colleagues at IMDEA, thank you for giving me advice, and for offering a helping hand.

Finally, to my girlfriend Paula, and my family, for their love, patience and support.

Abstract

Most distributed database systems offer weak consistency models in order to avoid the performance penalty of coordinating replicas. Ideally, distributed databases would offer strong consistency models, like serialisability, since they make it easy to verify application invariants, and free programmers from worrying about concurrency. However, implementing and scaling systems with strong consistency is difficult, since it usually requires global communication. Weak models, while easier to scale, impose on the programmers the need to reason about possible anomalies, and the need to implement conflict resolution mechanisms in application code.

Recently proposed consistency models, like Parallel Snapshot Isolation (PSI) and Non-Monotonic Snapshot Isolation (NMSI), represent the strongest models that still allow to build scalable systems without global communication. They allow comparable performance to previous, weaker models, as well as similar abort rates. However, both models still provide weaker guarantees than serialisability, and may prove difficult to use in applications.

This work shows an approach to bridge the gap between PSI, NMSI and strong consistency models like serialisability. It introduces and implements fastPSI, a consistency protocol that allows the user to selectively enforce serialisability for certain executions, while retaining the scalability properties of weaker consistency models like PSI and NMSI. In addition, it features a comprehensive evaluation of fastPSI in comparison with other consistency protocols, both weak and strong, showing that fastPSI offers better performance than serialisability, while retaining the scalability of weaker protocols.

Keywords

Consistency models, Transactions, Parallel Snapshot Isolation, Non-Monotonic Snapshot Isolation, Concurrency control.

Resumen

La mayoría de las bases de datos distribuidas ofrecen modelos de consistencia débil, con la finalidad de evitar la penalización de rendimiento que supone la coordinación de las distintas réplicas. Idealmente, las bases de datos distribuidas ofrecerían modelos de consistencia fuerte, como *serialisability*, ya que facilitan la verificación de los invariantes de las aplicaciones, y permiten que los programadores no deban preocuparse sobre posibles problemas de concurrencia. Sin embargo, implementar sistemas escalables que con modelos de consistencia fuerte no es fácil, pues requieren el uso de comunicación global. Sin embargo, aunque los modelos de consistencia más débiles permiten sistemas más escalables, imponen en los programadores la necesidad de razonar sobre posibles anomalías, así como implementar mecanismos de resolución de conflictos en el código de las aplicaciones.

Dos modelos de consistencia propuestos recientemente, Parallel Snapshot Isolation (PSI) y Non-Monotonic Snapshot Isolation (NMSI), representan los modelos más fuertes que permiten implementaciones escalables sin necesidad de comunicación global. Permiten, a su vez, implementar sistemas con rendimientos similares a aquellos con modelos más débiles, a la vez que mantienen tasas de cancelación de transacciones similares. Aun así, ambos modelos no logran ofrecer las mismas garantías que *serialisability*, por lo que pueden ser difíciles de usar desde el punto de vista de las aplicaciones.

Este trabajo presenta una propuesta que busca acortar la distancia entre modelos como PSI y NMSI y modelos fuertes como *serialisability*. Con esa finalidad, este trabajo presenta fastPSI, un protocolo de consistencia que permite al usuario ejecutar de manera selectiva transacciones serializables, reteniendo a su vez las propiedades de escalabilidad propias de modelos de consistencia débiles como PSI o NMSI. Además, este trabajo cuenta con una evaluación exhaustiva de fastPSI, comparándolo con otros protocolos de consistencia, tanto fuertes como débiles. Se muestra así que fastPSI logra un rendimiento mayor que *serialisability* sin por ello renunciar a la escalabilidad de protocolos más débiles.

Palabras clave

Modelos de Consistencia, Transacciones, Parallel Snapshot Isolation, Non-Monotonic Snapshot Isolation, Control de Concurrencia.

Chapter 1

Introduction

1.1. Motivation

Modern cloud applications are characterised by being globally available, and users expect to use the services provided by these applications with low latency, and in a reliable manner. To satisfy these requirements, programmers usually resort to distributed databases and storage, that allow to partition application data and place it geographically close to the users that need it. For example, a social media site would place data related to European users on data centers located in the same region, and the same for users in the United States. Under this design, however, those users requesting data from another region would suffer from high latency, as requests travel across different geographical regions. To this end, these data partitions are also replicated across different geographical regions, to ensure both low latency for all kinds of user requests, and fault tolerance, which allows applications to ensure a smooth operation even if an entire region goes offline.

These approaches, however, add significant complexity to the design and implementation of applications and the underlying databases. The presence of multiple replicas raises the question of how to keep them *consistent*, that is, reflecting an unified version of the data they contain. Traditional mechanisms to deal with database consistency prove harder to implement in efficient ways in distributed databases. For example, transactions should satisfy a set of desirable properties, commonly known as ACID: Atomicity, Consistency, Isolation, and Durability. These properties allow programmers to reason about concurrency as a set of isolated, atomic operations, but in geo-distributed scenarios it requires the coordination of multiple replicas in order to apply their updates.

The usual approach to bridge these problems involves relaxing the consistency guarantees that databases offer programmers [40]. Indeed, the CAP Theorem [16, 23] proves it is impossible to build applications that continue operating in the presence of network partitions without sacrificing consistency guarantees. However, the degree to which these guarantees can be relaxed offers a trade-off: on the one hand, weak consistency allows to build scalable applications without loss of availability, but proves difficult to reason about given that it allows non-serialisable behaviours called *anomalies*, and forces programmers to deal with

inconsistent data at the application level; on the other hand, strengthening consistency guarantees can reduce performance and hurt application availability, while being much easier to reason about.

Until recently, systems that offered weak consistency guarantees did not provide transactions (e.g. Dynamo [20]). In the recent years, however, a large number of transactional consistency models have been proposed for large-scale databases [6, 10, 32, 39]. Given the proliferation of different consistency models, it can be hard to choose which one is appropriate for a particular application, as it requires the programmer to think about the possible anomalies that can arise during an execution and about how they can interfere with application logic. Ideally, one would want to run all applications under strong consistency models, like *serialisability* [15], as programmers only need to check that application invariants hold as if transactions executed one after the other, without worrying about concurrency. Unfortunately, guaranteeing a serialisable execution in distributed databases is not possible without requiring global communication, which increases latency and limits availability [23].

This leaves the programmers with the responsibility of choosing an adequate consistency model for their applications. However, programmers often lack techniques to ensure that a given consistency model is safe to use for a particular application. One way to address this problem is to rely on the notion of *application robustness* [14, 22]: an application is robust against a particular consistency model if it behaves in the same way whether using a database providing this model or serialisability. When an application is robust, the programmer can take advantage of the scalability properties of a weak consistency model without paying the price of anomalous behaviours. Previous work has focused on static analysis of applications [29, 34], which let programmers know which parts of their applications are susceptible to anomalies. In these cases, programmers can selectively run transactions under serialisability: Fekete et al. [21, 22] propose several techniques that allow transactions executing under *snapshot isolation* (SI) [13] to exhibit serialisable behaviours, effectively making them equivalent to transactions running under serialisability.

Most recently, Bernardi and Gotsman [14] proposed a way to check the robustness of *parallel snapshot isolation* (PSI)[1] [39], which relaxes the consistency guarantees of snapshot isolation to allow more efficient implementations for distributed databases. PSI is also the strongest model that is weaker than SI [17], thus it is an obvious candidate to investigate its impact on the correctness of applications. Although there are several implementations that guarantee PSI [6, 33, 39], none of them were implemented with the focus on exploring the relation between PSI and application robustness.

1.2. Goals

The goal of this work is to help programmers bridge the gap between weak and strong consistency protocols, without sacrificing application correctness. To that end, fastPSI is proposed, an implementation of Parallel Snapshot Isolation that allows to selectively enforce serialisability for transactions through careful grouping of database objects into *entity*

[1] Also known as *non-monotonic snapshot isolation* [6]. This is discussed in §2.2.5.

groups [11]: transactions accessing objects in the same group execute as if they were running under a system guaranteeing SI (instead of the weaker PSI). Following the techniques proposed by Fekete et al. [22], these transactions can be further constrained so that they execute as if running under serialisability.

As such, the contributions of this work are:

- A hybrid consistency protocol that allows mixing Snapshot Isolation with Parallel Snapshot Isolation, by relying on entity groups. This protocol allows programmers to combine the scalability of Parallel Snapshot Isolation with the familiarity and intuitiveness of well-known consistency models like Snapshot Isolation and serialisability.
- A comprehensive evaluation of the proposed protocol, and a comparison against alternative implementations of both weak and strong consistency models.
- An exposition of the drawbacks and trade-offs of the protocol, and a discussion of how their impact can be minimised.

1.3. Work Plan

The work carried out for this project was divided in the following phases:

- Explore previous contributions. It's necessary to get familiarised with existing terminology, as well as previous work and implementations.
- Delimit project scope. After having the necessary knowledge to carry out the work, the limits and specific contributions of the work are established. In addition, the hypotheses that the final implementation should validate are proposed.
- Implementation phase. After setting clear objectives and milestones, the bulk of the implementation is done. Throughout this phase, testing and validation of the software is done, both with unit tests and with model checking techniques.
- Benchmark and validation. With the implementation complete, representative benchmarks are designed, as well as scenarios to validate the performance of the implementation. The results are used validate previous hypotheses, and also framed in the context of the existing literature.

1.4. Document Structure

The rest of this document is structured as follows. Chapter 2 provides an overview of previous work and the state of the art with regards to relevant consistency models, as well as basic notions that will be used throughout this document. Chapter 3 introduces fastPSI,

a PSI protocol that allows the programmer to selectively enforce serialisable executions. It also discusses some potential shortcomings of its design. Chapter 4 presents the results of the work, with a comprehensive evaluation of fastPSI and a comparison against alternative consistency models. Chapter 5 compares the approach in this work with the previous work outlined in Chapter 2, and highlights the main differences. Finally, Chapter 6 provides the conclusions.

1.5. Sources and Repositories

The relevant sources related to this document can be found online at the following repositories:

- The server-side of the fastPSI protocol, to be found at `https://github.com/ergl/antidote/tree/pvc`.
- The client-side library of the fastPSI protocol, to be found at `https://github.com/ergl/pvc/tree/v0.8.0`.
- The software used to benchmark fastPSI, modified with extra features, to be found at `https://github.com/ergl/lasp-bench/tree/coord`.
- The benchmark data, along with several scripts used to generate the plots on Chapter 4, to be found at `https://github.com/ergl/antidote-evaluation`.

1.6. Related Program Courses

Several courses during the Computer Science program were of great help to learn the foundations and the necessary skills to carry out the work in this document:

- *Databases* and *Advanced Databases*, which taught the foundational concepts of transactions, consistency and concurrency control. In particular, the latter covered the implementation and design of the storage subsystems of databases.
- *Networks* and *Advanced Operating Systems and Networks*, which taught the necessary skills to design and implement an efficient client-server protocol on top of TCP.
- *Technology and Organisation of Computer Systems*, which taught the necessary cache and distributed memory designs that allowed an increased performance of the fastPSI server implementation.
- *Probability and Statistics* and *Evaluation of Computer Systems*, which taught the foundational concepts needed for an effective interpretation of benchmark data, along with theoretical models—such as queueing theory—necessary to tune performance and to design effective performance experiments.

Chapter 2

Preliminaries

This chapter begins with an introduction to the notation and basic concepts used throughout this document. It also reviews several *strong* and *weak* consistency models, based on the anomalies that are observable in each one.

2.1. Notation

This section defines the elements used throughout this chapter, such as transactions, histories, and relations. It follows the models used by Saeida Ardekani [7], Adya [4] and Bernstein et al. [15].

2.1.1. Objects and Replication

Following the definitions of Cerone et al. [17] and Saeida Ardekani [7], a database stores ***objects*** $\mathsf{Obj} = \{x, y, \ldots\}$, which are assumed to be integer-valued.

Under such definition, a system consisting of independent processes $\Pi = \{p_1, \ldots p_n\}$ offers ***full replication*** if every process in Π holds a copy of the entire set of objects Obj, and ***partial replication*** if processes hold a subset of Obj. In partially-replicated systems, processes might hold disjoint sets of Obj, although it is not necessary.

2.1.2. Transactions

Clients interact with the database via ***transactions*** $\mathsf{Tx} = \{T_i \mid i \in \mathbb{N}\}$, with i being the *transaction identifier* of T.

A transaction is a totally ordered sequence of read or write operations, followed by a *terminating* operation: either commit or abort. This order follows the order in which the client invoked such operations. Given an object x and a transaction T_i, x_i represents the ***version*** i of x written by T_i. A transaction T_i writing a version i of x is denoted by $w_i(x_i)$, and $r_i(x_i)$ denotes when T_i reads a version i of x. Finally, c_i denotes when T_i commits, and a_i when it aborts. It is assumed that an initial transaction T_0 exists, such that it

writes the initial versions of every object in the database. Without loss of generality, one can assume that no transaction performs *blind updates*, that is, for every write operation $w_i(x)$ performed by T_i, there's always a preceding read operation $r_i(x)$. A transaction is ***read-only*** if its set of operations does not include writes, and ***update*** otherwise.

2.1.3. Histories

A ***history*** h represents the finite set of all transactions with disjoint identifiers issued against a database. The set of all possible histories is denoted by $\mathcal{H}$. Two orders are defined over a history h: a ***real-time*** $<_h$ total order, which relates operations by their wall-clock execution time; and a ***happens-before*** $\lhd_h$ strict partial order, such that for any two transactions T_i and T_j, if $w_i(x_i)$ and $r_j(x_i)$, $T_i \lhd_h T_j$ (that is, T_j reads the version i of x written by T_i).

Intuitively $T_i \lhd_h T_j$ means that T_j is aware of the updates performed by T_i, and thus the outcome of the operations in T_j may depend on the effects of T_i. In this case, T_i is a ***causal dependency*** of T_j. A transaction T_i is ***pending*** in h if $(c_i \vee a_i) \notin h$.

The history and the relations between operations and transactions can be represented as a graph, following Bernstein et al [15]. A history can also be represented as a permutation of operations to show the real-time order, e.g. $h = r_1(x_0).w_1(x_1).r_2(y_0).w_2(y_2).c_2.c_1$.

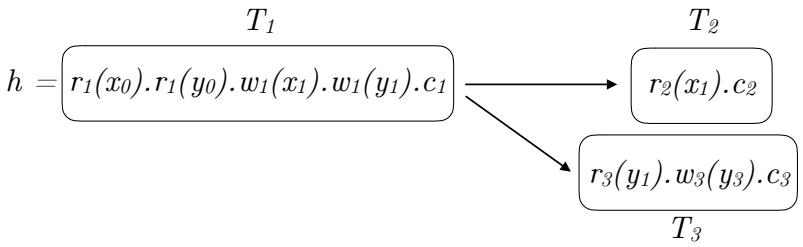


Figure 2.1: *Example of history represented as a graph. Boxes group operations intro transactions, and arrows represent causal dependencies between transactions. Adapted from* Saeida Ardekani et al. [6].

Figure 2.1 above shows an example, with $T_1 \lhd_h T_3$ and $T_1 \lhd_h T_2$. Two transactions T_i and T_j are ***concurrent*** in h (denoted by $T_i \parallel T_j$) if neither $T_i \lhd_h T_j$ nor $T_j \lhd_h T_j$. In the figure above, T_2 and T_3 are concurrent.

2.2. Consistency Models

This section defines what a consistency model is, placing them in two categories: *strong* and *weak*. It also gives an overview of several models, and compares them in terms of their undesirable effects.

A consistency model is defined as a subset of histories $\mathcal{C} \subseteq \mathcal{H}$. Intuitively, a consistency model *constrains* the set of possible histories by specifying how operations interleave in any

given history. A given history h *satisfies* a consistency model $\mathcal{C}$ if $h \in \mathcal{C}$. Otherwise, h *violates* $\mathcal{C}$.

In the context of databases, the definition of a consistency model maps to the concept of an *isolation level* (I in ACID), which specifies the degree to which concurrent transactions in a database are aware of each other [4]. The term consistency is used throughout this document, in accordance with Adya [4].

Traditionally, the different consistency models have been defined in terms of *anomalies* [13], that map to a set of undesirable histories that are observable by the system. Intuitively, one can distinguish between *strong* and *weak* consistency models depending on the number of anomalies they disallow, with stronger models restricting the set of possible histories more than weak ones. The following sections give a brief overview of the traditional anomalies following the definitions advanced by Berenson et al. [13], as well as different consistency models that preclude them.

Definition 2.1 (Dirty Write)**.** A *Dirty Write* occurs in a history h when a transaction T_i modifies an object x that was previously modified by another pending transaction T_j. If any of the two transactions commit or abort, it is not clear what value the object x should have. A Dirty Write can be represented by a history such as $w_1(x_1).w_2(x_2).(c_1 \vee a_1).(c_2 \vee a_2)$, where the termination order of the transactions can be arbitrary. The anomaly occurs even if any of the transactions abort.

Definition 2.2 (Dirty Read)**.** A *Dirty Read* occurs in a history h when a transaction T_i reads an object x modified by a pending transaction T_j. If T_j ultimately aborts in h, its updates never take place. Thus, the value shouldn't be observed by T_i. This is represented by a history such as $r_1(x_0) \dots w_1(x_1) \dots r_2(x_1) \dots c_2 \dots a_1$.

Both *Dirty Writes* and *Dirty Reads* can be prevented at the implementation level by making the changes made by a transaction visible only after the transaction commits. For example, the updates of a transaction can be buffered locally. All the consistency models described in the sections that follow preclude dirty writes and reads. Transactions executing under such models offer *atomicity* (A in ACID).

2.2.1. Read Committed (RC)

Read Committed is the simplest consistency model that satisfies the atomicity guarantee of ACID transactions, by preventing dirty reads and writes. Although this model specifies that all or none of the updates by a transaction are applied to the database, it does not prevent concurrent transactions from observing only a subset of those updates. As a result of the simplicity of this model, systems operating under Read Committed might observe anomalous behaviour, such as non-repeatable reads or lost updates, as described below.

Definition 2.3 (Non-Repeatable Read)**.** A non-repeatable read—also called a *fuzzy read*—occurs whenever a transaction T_i observes different values for an object x on subsequent read operations when interleaved by a commit by another transaction T_j. This can be seen

a history such as $r_1(x_0) \ldots r_2(x_0).w_2(x_2).c_2 \ldots r_1(x_2)$. In the previous history, T_1 observes two different values of x: the initial version x_0 and the version x_2 written by T_2.

The non-repeatable read anomaly can be prevented in an easy way: a transaction can keep a cache of already read values. Subsequent read operations can return values from this read cache.

Definition 2.4 (Lost Update)**.** A *Lost Update* occurs in a history h when two concurrent transactions update the same object x and successfully commit. In this case, the update written by whichever transaction commits first is "lost" after the second transaction commits. This can be seen in a history such as $r_1(x_0).r_2(x_0) \ldots w_1(x_1).c_1 \ldots w_2(x_2).c_2$. After T_2 commits, version x_1 is no longer visible to subsequent transactions.

Compared to the Dirty Write anomaly, a Lost Update can occur even if the changes of a transaction are only made visible after it commits. For the purposes of this document, and following the definition used by Saeida Ardekani [7], a consistency model is *strong* if it prevents concurrent transactions from modifying the same object. That is to say, a strong consistency model precludes Dirty Writes and Dirty Reads, along with Lost Updates. A model that allows concurrent transactions to modify the same object is *weak*. The Read Consistency model, already introduced, is the weakest consistency model covered in this work.

2.2.2. Serialisability (SER)

Serialisability restricts the set of possible histories to those equivalent to some *serial* execution. That is to say, under a traditional system offering serialisability, the happens-before relation introduced in 2.1.3 is strengthened with a *strict total order* over the transactions in a history. Such an order does not need to follow real-time order, thus, a system under serialisability is free to reorder transactions as long as the resulting history is sequential. A history such as the one in Figure 2.1 is serialisable: one can order T_2 to occur before T_3 (or vice versa); while the one depicted in Figure 2.2 is not.

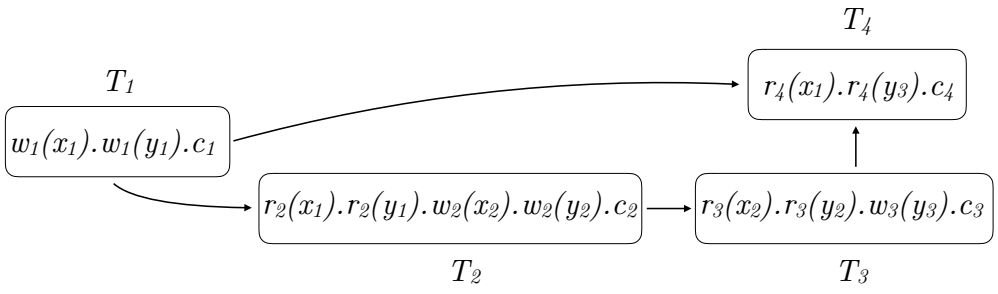


Figure 2.2: *Example of a non-serialisable history. T_4 observes version y_3 written by T_3, but fails to observe version x_2 written by T_2. This result can't be obtained by executing the transactions in any sequence, and thus the history is not serialisable.*

Because of the CAP theorem, distributed databases can't offer serialisability without sacrificing availability [16, 23]. Although some database systems offer it [19], guaranteeing serialisability in geo-replicated systems is difficult, requiring global communication and consensus mechanisms, like Paxos [31].

2.2.3. Snapshot Isolation (SI)

Proposed by Berenson et al. [13], Snapshot Isolation (SI) relaxes the total ordering guarantees of serialisability by requiring only a partial order among committed transactions. To preclude the Lost Update anomaly, SI introduces the concept of a *snapshot*: a private view of the data written by committed transactions. This snapshot is created at the time the transaction starts, called its *start timestamp*. A transaction T is not able to see updates made by other transactions that commit after T's start timestamp. When a transaction T is ready to commit, it gets assigned a *commit timestamp*, larger than any previous start or commit timestamp. Transaction T commits successfully only if no other concurrent transaction updated the same objects as T. Under SI, two transactions are concurrent if their start–commit timestamps intervals overlap, and read-only transactions always commit. Two transactions ***write-conflict*** if they update a non-disjoint set of objects.

Due to the partial order among transactions, and the condition that concurrent transactions only abort if their set of updated objects overlap, a system under Snapshot Isolation is able observe the so-called *Write Skew* anomaly, defined below. This anomaly occurs when an external invariant is violated as a result of concurrent non-conflicting transactions. As an example, suppose that objects x and y are related by a constraint such that $x+y \leq 10$, with $x = 5$ and $y = 2$ as the initial state. In such a setting, a history such as the following can violate the invariant: $h = r_1(x = 5).r_1(y = 2).r_2(x = 5).r_2(y = 2).w_1(x = 7).w_2(y = 4).c_1.c_2$. Both T_1 and T_2 start their execution and observe a consistent state that upholds the invariant. Then, T_1 updates the object x to the value 7, which still maintains the invariant, as $7 + 2 \leq 10$. Concurrently, T_2 proceeds as T_1, updating y to 4, which also maintains the original invariant ($5 + 4 \leq 10$). However, as the result of both T_1 and T_2 successfully committing, the invariant is violated, with $7 + 4 \not\leq 10$.

Definition 2.5 (Write Skew)**.** A *Write Skew* occurs whenever non-conflicting transactions update objects related with an external invariant, successfully commit, and as a result violate the original invariant.

Due to the fact that transactions are assigned monotonically-increasing *start* and *commit* timestamps, this imposes a total order among the commit times of transactions, i.e. a total ***commit order***. In addition, given that the updates of a transaction T must be immediately visible to subsequent transactions, this means that T must wait to commit until all its updates have been propagated to all remote replicas. These two facts make Snapshot Isolation unsuitable for geo-replicated database systems without relying on global communication, which makes them hard to scale.

2.2.4. Parallel Snapshot Isolation (PSI)

Parallel Snapshot Isolation (PSI), proposed by Sovran et al. [39], is a consistency model aimed at solving the scalability limits of classical Snapshot Isolation in geo-replicated systems. While concurrent conflicting transactions are not allowed, PSI allows non-conflicting transactions to exhibit a relative commit order that varies between replicas. This means that PSI can propagate transactions to replicas in *causal* order, sidestepping another scalability limit of SI.

However, allowing different commit orders for non-conflicting transactions at different replicas (or *sites*) makes PSI susceptible to the *Long Fork* anomaly [39]. Consider the history depicted in Figure 2.3. If transactions T_4 and T_5 execute in different replicas, they are allowed to observe different commit orders for T_2 and T_3.

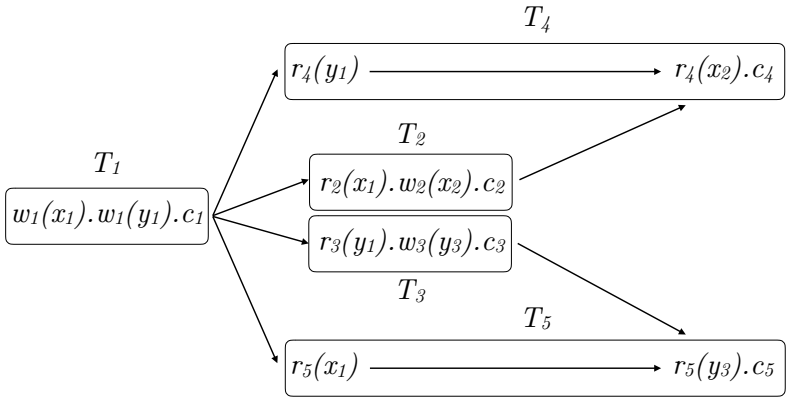


Figure 2.3: *Example of a history showing the Long Fork anomaly. Transaction T_4 observes $T_1 \lhd T_2 \lhd T_3$ while T_5 observes $T_1 \lhd T_3 \lhd T_2$.*

Definition 2.6 (Long Fork)**.** A *Long Fork* occurs whenever transactions are able to observe different commit orders of previous non-conflicting update transactions.

Like SI, PSI exhibits *base freshness* [8]: a transaction T is limited to read only versions of objects written by transactions that committed before T started. In the geo-replicated scenarios that PSI is meant to address, this limitation leads to a high number of stale data reads. Consider the example of two sites s_1 and s_2 separated by a high latency link: if a transaction starts in s_1 and subsequently tries to read an object located at s_2, it might be the case that the version it is forced to read due to base freshness has already been overwritten by other transactions. Saeida Ardekani et al. [38; Theorem 4] prove that base freshness requires replicas that do not replicate data accessed by a transaction T to coordinate in order to commit T, which results in lower system performance and limits scalability. Indeed, the original implementation advanced by Sovran et al. communicates with all the replicas in the system [39].

2.2.5. Non-Monotonic Snapshot Isolation (NMSI)

The Non-Monotonic Snapshot Isolation (NMSI) consistency model also alleviates the total commit order scalability problem in classical Snapshot Isolation. NMSI was proposed by Saeida Ardekani et al. [6] as an improvement over the previously described Parallel Snapshot Isolation model, by exhibiting *forward freshness*: a transaction T is allowed to read versions written by transactions that committed after T started, as long as those reads form a *causally consistent snapshot*. A transaction T observes a causally consistent snapshot if all the versions read by T are written by its direct causal dependencies, i.e. if a transaction T_i performs $r_i(x_j)$ such that it depends on T_j, then there's no such $w_k(x_k)$ such that $T_j \triangleleft T_k \triangleleft T_i$.

In spite of this, the set of possible anomalies produced by both Parallel Snapshot Isolation and Non-Monotonic Snapshot Isolation are the same, as can be seen in Figure 2.4. In addition, both models can be proved to be equivalent, as shown by A. Cerone (2016, personal communication with the author), with the only difference being the choice of the concurrency control algorithm.

2.2.6. Anomaly Comparison

Figure 2.4 summarises the consistency models reviewed, together with the anomalies that they allow.

Anomalies	Consistency Models				
	SER	SI	PSI	NMSI	RC
Dirty Write	x	x	x	x	x
Dirty Read	x	x	x	x	x
Non-Repeatable Read	x	x	x	x	✓
Lost Update	x	x	x	x	✓
Write Skew	x	✓	✓	✓	✓
Long Fork	x	x	✓	✓	✓

Figure 2.4: *Anomaly Comparison of Consistency Models* (x:*disallowed*, ✓:*allowed*). *Adapted from* Saeida Ardekani et al. [6].

Chapter 3

The fastPSI protocol

This chapter describes **fastPSI**, a transactional protocol that implements Parallel Snapshot Isolation and allows stronger consistency guarantees for transactions accessing objects inside *entity groups*. The chapter begins with an overview of what entity groups are, and by explaining the consistency guarantees of fastPSI for transactions executing both within and across different groups. It follows with a summary of how the different participants of the protocol interact with each other, and with a description of the different data structures involved. Next, it shows how the protocol is structured by going over the execution of a transaction. The chapter concludes with a discussion of the possible drawbacks of the design.

3.1. Consistency Guarantees

The fastPSI protocol considers a system in which objects are aggregated in *entity groups* [11]. An entity group σ is defined as a proper partition of objects Obj. This means that two properties hold: (i) $\forall \sigma, \sigma' \implies \sigma \cap \sigma' = \emptyset$, and (ii) $\forall x \in \mathsf{Obj}.\ \exists \sigma.\ x \in \sigma$. The first property says that the sets of objects covered by different entity groups are *disjoint*, while the second property states that any object that exists in the system is part of an entity group.

The goal of fastPSI is as follows: transactions that only access objects inside a single entity group should satisfy Snapshot Isolation (SI), while transactions that access objects across entity groups should satisfy Parallel Snapshot Isolation (PSI). Intuitively, if one has a single entity group that encompasses every object, any execution of fastPSI is equivalent to an execution under Snapshot Isolation, thereby precluding the Long Fork anomaly. Conversely, if one has an entity group per object in the system, then any execution of fastPSI is equivalent to an execution under Parallel Snapshot Isolation. The decision of which objects to place into different entity groups is left to the programmer, who should take application requirements into account.

Recall from Section 2.2.3 that transactions executing under SI are only partially ordered, in contrast with serialisability, where they are totally ordered. However, the requirement for

transactions to take monotonic *start* and *commit* timestamps induces a total *commit order* for transactions, even for those that are not in conflict with each other. In the presence of different entity groups, this requirement would require transactions to communicate with every group in order to determine a monotonic timestamp. In fastPSI, this requirement is relaxed so that transactions have multiple, independent timestamps, one per entity group.

In order to guarantee Parallel Snapshot Isolation across entity groups, the protocol incorporates the notion of forward freshness [6], which allows a transaction T to read versions of objects written by transactions that committed after T starts. The fastPSI protocol accomplishes this by making a transaction T fix its start timestamp at a particular entity group only when T reads an object from that group. This also allows a transaction to acquire start timestamps only at the groups it reads from, thus avoiding coordination with other groups.

The fastPSI protocol leverages both approaches to offer its consistency guarantees: a transaction is able to read from versions written by latter transactions as long as those reads form a *causally consistent snapshot*. When a transaction T performs its first read operation in an entity group, it fixes a snapshot that includes all the transactions that committed before T's read occurred. As T performs further read operations on other entity groups, the snapshots that T fixes are restricted to versions written by transactions that are not causally dependent on the transactions that T already included in its previous snapshots.

3.2. Overview and System Model

The protocol consists of three components: *client* processes that provide the system interface for managing transactions, *server* processes that handle the individual operations of transactions, and *entity groups*—managed by a server—which store individual data objects. Given that entity groups properly divide the range of objects into disjoint *partitions*, for the remainder of this chapter the term partition is used as a shorthand for entity group.

Both client and server processes are considered reliable and connected by reliable channels[1]. Processes communicate with each other using an asynchronous message-passing system. Server processes are denoted as a set $\mathcal{S} = \{s_1, \ldots, s_N\}$, and clients as $\mathcal{C} = \{c_1, \ldots, c_M\}$. Data objects are denoted by a set Obj, split into N partitions, each stored by a server process. In addition, $\mathsf{partition}(x)$ represents the index of the partition the object x belongs to, such that it is managed by server $s_{\mathsf{partition}(x)}$. For simplicity, it is assumed that server processes only manage a single partition.

Clients provide the transactional interface of the protocol through the `start`, `read`, `write` and `commit` operations. Transactions are *interactive*, i.e., when a transaction starts, the client does not know which operations it will perform in advance. In fastPSI, the `start` and `write` operations are local to the client. Clients issue `read` operations to servers, which return the values and metadata associated with the objects the client requested. Clients

[1]Fault-tolerance concerns are orthogonal to the problem addressed, although several approaches are discussed in Chapter 5.

handle `write` operations locally by storing the updates in a buffer, called the *write-set* of a transaction. At `commit` time, the client acts a coordinator of a *two-phase commit protocol (2PC)* [15], issuing `prepare` and `decide` operations to all the participating servers. The written values buffered locally are transmitted to the servers at `prepare` time, together with the accumulated metadata for the objects that the client read. The decision to commit or abort a transaction is taken based on this metadata.

Servers handle three operations issued by clients: `read`, `prepare` and `decide`. When a server handles `read` operations, it forwards the request to the partition responsible for the object being requested. When receiving a `prepare` request for a certain transaction, the server checks for conflicts with other transactions waiting to be decided, which are stored in a commit queue. If the transaction contained in the client request does not conflict, it is added to the queue, and the server replies to the client with a COMMIT vote. If, on the other hand, the transaction is found to be conflicting, an ABORT vote is sent instead.

Partitions are responsible for fulfilling read requests on behalf of servers, and for maintaining causally consistent snapshots on behalf of the clients. Partitions store multiple *versions* of an object in accordance with a multi-version concurrency control protocol. When executing a read request for an object x, a partition finds and returns the most recent causally consistent version of x, along with some metadata of the chosen version.

Each partition stores multiple versions of an object represented by a tuple $\langle val, vid \rangle$, where val is the value of a given version, and vid is a logical identifier for the transaction that committed this version. This logical identifier is represented using *version vectors* [35]. Such a vector consists of N entries, one for each partition, storing a non-negative integer. Each entry $vid[i]$ in the vector can also be represented by a pair (s_i, k), where k is the actual value of the i-th entry of the vector. These pairs are also called dots [12]. Version vectors are compared according to the following relation, showing when one vector covers more dots than another: $V_1 \sqsubseteq V_2 \iff \forall i.\ V_1[i] \leq V_2[i]$. In addition, there exists a *join* operation on vectors, taking their entry-wise maximum, which will be denoted by max from now on. The set of all version vectors is denoted by VerVector, and the vector with all entries set to 0 by $\vec{0}$.

3.3. Server data structures

Each server s_i maintains five main data structures, summarised in Figure 3.1. This section now follows with a more detailed explanation of each of them.

LastPrep is a counter of the number of update transactions that initiated their commit phase at a given server. When a transaction commits at a server s_i, it gets assigned the value of the counter as a *sequence number* k, which induces the *commit order* of transactions at s_i. A transaction computes its *commit vector* V_c from these sequence numbers, such that the vector represents the set of dots $\{(s_i, k) \mid k \leq V_c[i]\}$ which identify the writes by previous transactions whose sequence number at s_i is no higher than $V_c[i]$.

VersionLog is a mapping of objects to a list of versions, called the *database*. As noted before, each version is a tuple $\langle$val, Vcomm$\rangle$ such that val is a value and Vcomm is the

Data Structures at a server s_i		
LastPrep	Integer	The number of update transactions that tried to commit at the server.
VersionLog	Map[Object, Set[⟨Value val, VerVector Vcomm⟩]]	Database: a mapping from objects to lists of pairs of a value and the commit vector of the transaction that wrote it. The lists are ordered by the i-th component of the commit vectors.
CommitLog	Sequence[⟨Tx T, VerVector Vaggr⟩]	Log of update transactions T committed at the server, ordered by Vaggr[i]. Here Vaggr is the aggregate vector of T: the join of the commit vectors of all transactions up to T in CommitLog.
Vtotal	VerVector	The join of the commit vectors of all transactions in CommitLog.
CommitQueue	Sequence[⟨Tx, PENDING, WriteSet⟩ ∪ ⟨Tx, DECIDED, WriteSet, VerVector⟩]	Queue containing information about update transactions trying to commit at the server.

Figure 3.1: *Data structures used by servers in the protocol. The orders of entries in* CommitLog*,* VersionLog *and* CommitQueue *are consistent with the commit order of the associated transactions. Components of various tuples are selected using the names given in the figure.*

commit vector of the transaction that wrote val. The VersionLog at s_i is ordered by the i-th component of the commit vector of each version, which follows the commit order of transactions at the server. The most recent entry in the list for the object x is denoted by VersionLog[x].last.

CommitLog is an ordered list that maintains a tuple $\langle T,$ Vaggr$\rangle$ for each update transaction that committed at a server, such that T is the identifier of a committed transaction and Vaggr is an *aggregate vector*. The aggregate vector represents the join of the commit vectors of all the transactions up to T in CommitLog. Entries in the log at s_i are totally ordered according to the i-th entry of their aggregate vectors, Vaggr[i], which also follows the commit order of transactions at the server. These aggregate vectors are stored for efficiency, and are used to compute the snapshot of a transaction at the server. The aggregate vector of the last committed transaction is stored in Vtotal. Initially, the CommitLog contains a single placeholder entry $\langle \_, \vec{0} \rangle$.

Finally, CommitQueue is an ordered queue of transactions trying to commit updates at the server. The queue has entries of two types. An entry $\langle T, \text{PENDING}, WS \rangle$ means that T is successfully prepared to commit at s_i, but the final decision on it is not yet known; WS is the *write-set* of the transaction, containing object-value pairs. An entry $\langle T, \text{DECIDED}, WS, V \rangle$ in the queue means that T has been decided to commit with a commit vector V, but its writes have not yet been added to VersionLog. The order of transactions in CommitQueue follows the commit order at the server.

3.4. Protocol Description

The protocol is now described in detail by following the execution of a transaction. This section begins by describing how a transaction is initialised, along with the mechanism to build a causally consistent snapshot. Later, it shows the mechanism to validate and commit a transaction.

3.4.1. Transaction Execution

The fastPSI protocol uses optimistic concurrency control, that is, the execution of a transaction is speculative. Clients read objects from servers and buffer writes locally. At the end of the execution, the decision whether to commit or abort a transaction is taken based on the existence of conflicts with concurrently executing transactions. Clients executing a transaction maintain a transaction *context* including several pieces of data, summarised in Figure 3.2 and explained below.

In the following, the steps taken by both clients and servers to execute T refer to the algorithms in Figures 3.3 and 3.4.

Context for a transaction T at a client c_i		
T.WS	WriteSet	Write-set of T.
T.HasRead	Vector[Bool]	Mapping showing whether T has read a given partition.
T.Vsnap	VerVector	Snapshot vector: determines snapshots fixed at partitions T has read from and possible causal dependencies at all other partitions.
T.Vdep	VerVector	Dependency vector, representing all causal dependencies developed by T during its execution.

Figure 3.2: *Data structures used in the transaction context, kept by the clients in the protocol. In the table,* $\mathsf{WriteSet} = \mathsf{Set}[\langle \mathsf{Object}, \mathsf{Value} \rangle]$

When a client starts a transaction T, it first initialises its context (line 1). When a transaction T writes a value v to an object x (line 3), the client buffers this write in T's *write-set*, $T.\mathsf{WS}$, while discarding any previously written value of x.

```
1 function start()
2     return new Tx(WS = ∅, HasRead = ⊥⃗, Vsnap = 0⃗, Vdep = 0⃗);
3 function write(T, x, v)
4     T.WS ← (T.WS \ {⟨x, _⟩}) ∪ {⟨x, v⟩};
```

Figure 3.3: *Initialisation of a transaction and update of an object* x *at client* c_i.

When the transaction T issues a read operation on an object x (line 5), the client first checks $T.\mathsf{WS}$ (line 6): if T has already written to x, the value stored in the write-set is

returned. Otherwise, and assuming that $j = \mathsf{partition}(x)$, the client sends a READREQUEST message to the server s_j to fetch the value of the object (line 9).

When the transaction T reads an object from a partition j for the first time, the server s_j fixes a *snapshot* of versions from which it will serve all future reads by T. This snapshot is defined by an integer k: it will include the versions written by all the transactions that committed at the server with a sequence number up to k. The client keeps this information in the transaction context, by storing k in the j-th entry of a *snapshot vector* $T.\mathsf{Vsnap}$, and by marking the current partition as read in $T.\mathsf{HasRead}$, a boolean mapping its j-th entry to $\top$ if T read an object from s_j, and $\bot$ otherwise. If $T.\mathsf{HasRead}[j] = \top$, then the vector $T.\mathsf{Vsnap}$ is equal to the join of the commit vectors of all transactions committed at s_j with a sequence number no higher than $\mathsf{Vsnap}[j]$.

```
5  function read(T, x)
6      if ⟨x, v⟩ ∈ T.WS then
7          return v;
8      j ← partition(x);
9      send READREQUEST(x, T.Vsnap, T.HasRead) to s_j;
10     wait receive READRETURN(m) from s_j;
11     if m = ABORT then
12         throw ABORT;
13     else if m = ⟨v, Vdep, Vaggr⟩ then
14         T.HasRead[j] ← ⊤;
15         T.Vdep ← max(T.Vdep, Vdep);
16         T.Vsnap ← max(T.Vsnap, Vaggr);
17         return v;
18 when received READREQUEST(x, Vsnap, HasRead) from c_j
19     if HasRead[i] then
20         V ← Vsnap;
21     else
22         wait until Vtotal[i] ≥ Vsnap[i];
23         r ← max{r ∈ CommitLog | ∀j. HasRead[j] ⟹ (r.Vaggr[j] ≤ Vsnap[j])};
24         if r.Vaggr[i] < Vsnap[i] then
25             send READRETURN(ABORT) to c_j;
26             return;
27         V ← r.Vaggr;
28     ver = max{ver ∈ VersionLog | ver.Vcomm[i] ≤ V[i]};
29     send READRETURN(ver.val, ver.Vcomm, V) to c_j;
```

Figure 3.4: *Local and remote read of object* x.

Thus, the entries in the snapshot vector for partitions that T has not yet read from

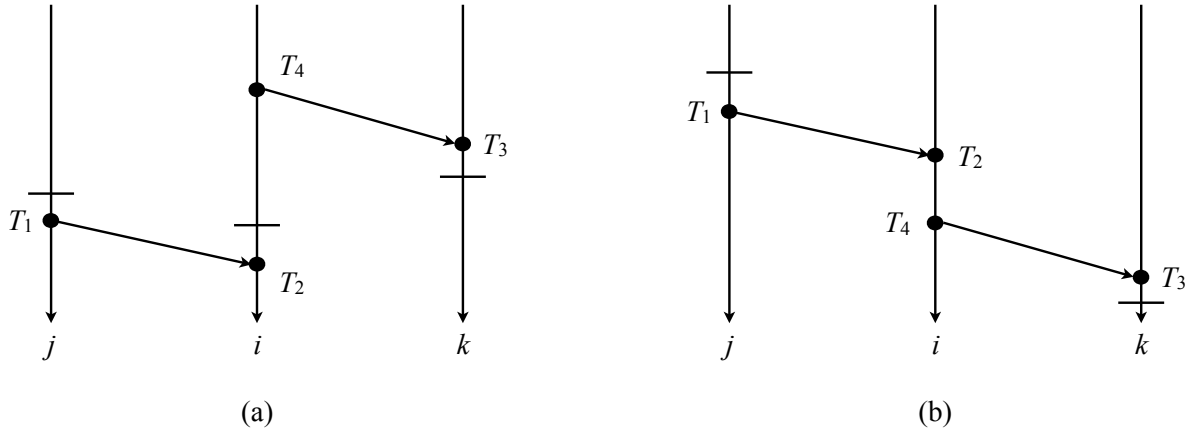


Figure 3.5: *Illustrations of the snapshot computation. Vertical lines depict the commit order at the corresponding partitions (top to bottom) and horizontal lines the cut-offs of various snapshots. Arrows between partitions depict causal dependencies.*

delimit all the possible causal dependencies T may develop at these partitions.

Both T.Vsnap and T.HasRead are supplied by the client when issuing a read operation on object x, by using them as parameters to the `READREQUEST` message sent to a server s_i. When the server receives this message (line 18), it first checks, using the HasRead mapping, if the transaction has read from it before (line 19). In this case, the snapshot is determined by Vsnap[i], and the server returns the latest version *ver* of the object x written by a transaction in the snapshot, i.e., with a sequence number no higher than Vsnap[i] (line 28). This version is determined by examining the i-th entry of the commit vectors in VersionLog. The server then replies to the client with a `READRETURN` message containing the value of the chosen version and its associated version vector, as well as the unmodified snapshot vector provided by the client: since the server used a previously fixed snapshot, no updates to the vector are required.

In the case when the client reads from the server s_i for the first time (line 21), it is necessary to fix the snapshot for the transaction T at this server. Choosing a suitable snapshot is complicated by the fact that transactions are allowed to be interactive—that is, it is not know in advance which objects a transaction will read in the future. The snapshot is hence fixed in such a way that *any* later read from this snapshot will be *causally consistent* with any other read from the snapshots that T has already fixed, as specified by HasRead and Vsnap. To ensure this, the selected snapshot has to satisfy two requirements, depicted in Figure 3.5.

On the one hand, the snapshot cannot be too fresh. For example, suppose a transaction T_1 that wrote to partition j is excluded from the snapshot chosen by T at j. Then, the snapshot chosen by T at partition i cannot contain T_1, nor any other transaction that causally depends on it, like T_2 (Figure 3.5a). If the snapshot chosen by T included T_2, it would be able to read some of T_2's writes at i, thereby forcing T to read the writes by T_2's causal dependencies, including T_1; but T cannot see these writes, because it excluded

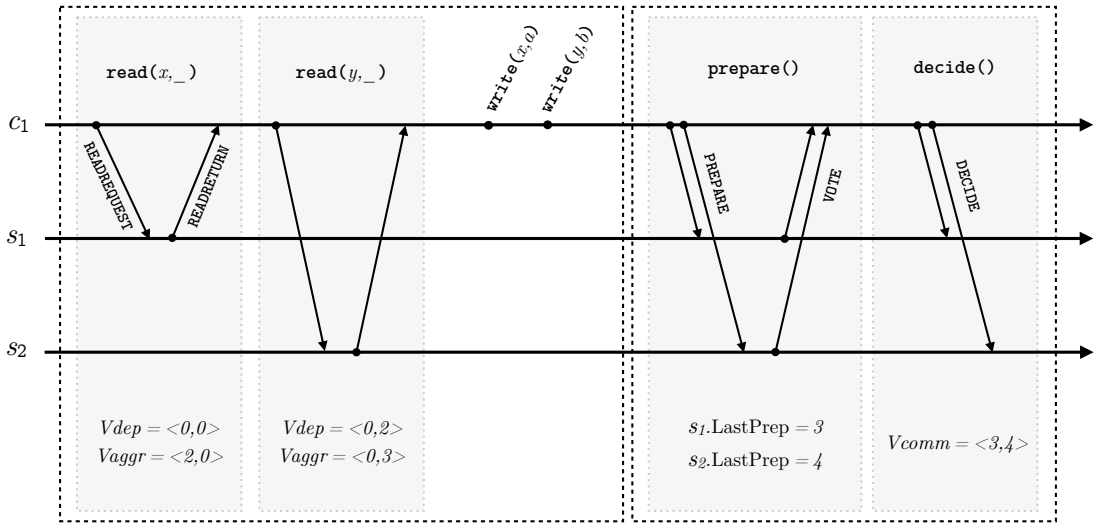
them from the snapshot at partition j. Thus, when building the snapshot, the server needs to take into account the snapshots taken by T at the partitions it already read; the server selects the longest prefix of CommitLog transactions, such that their writes—and the ones by their causal dependencies—are included in T's previous snapshots. This prefix is denoted by r (line 23), and is computed using the Vaggr vector included in each of the entries of CommitLog, summarising the causal dependencies of all transactions up to a given record in the log. Thus, the snapshot at s_i includes all transactions with sequence numbers up to $r.\mathsf{Vaggr}[i]$.

On the other hand, the snapshot selected cannot be too stale. Continuing with the previous example, if a transaction T_3 is included in the snapshot taken by T at some partition k, then the snapshot of T at partition i has to include the writes by T_3 and its causal dependencies, e.g., the transaction T_4 in Figure 3.5a. The snapshot vector $T.\mathsf{Vsnap}$ summarises the updates of the transactions (and of its causal dependencies) included in the snapshots fixed by T. Thus, after determining the appropriate snapshot in line 23, the server s_i checks that this snapshot covers transactions with sequence numbers at i up to $\mathsf{Vsnap}[i]$ (line 24). To maximise the chances of passing this check, before allowing a transaction T to proceed with a read, the server s_i waits until the writes from the prefix up to $\mathsf{Vsnap}[i]$ have been incorporated into its state (line 22).

It may be the case that it's impossible to satisfy both of the above requirements when selecting a snapshot; e.g., in the situation illustrated in figure 3.5b. Assuming that the transaction T has fixed a valid snapshot at j and k, it is impossible to build a consistent snapshot at partition i; given that T included T_3 at partition k, it is forced to read T_4's writes at i. At the same time, because it excluded T_1 from the snapshot at j, T can't read the writes by T_2 at i. In this case, without the second requirement, the server s_i would build a snapshot excluding both T_2 and T_4, violating T's causal dependency on T_3. In this case the server sends to the client a `READRETURN` message with a special value ABORT (line 25), which will cause the client to abort the transaction (line 12).

Once the server fixes a new snapshot, it selects the most recent version of the object x, defined by $e.\mathsf{Vaggr}[i]$ (line 28), to return to the client. The server replies with a `READRETURN` message, carrying a triple of the value of the object, its associated version vector, and the aggregate vector for $e.\mathsf{Vaggr}$, summarising the causal dependencies of all the transactions in the snapshot. When the client receives the message (line 13), it first sets the j-th entry of $T.\mathsf{HasRead}$ to $\top$, to indicate that T has read an object at partition j, and joins the returned aggregate vector to $T.\mathsf{Vsnap}$. The client also joins the commit vector associated with the version read to a *dependency vector* $T.\mathsf{Vdep}$, which represents all causal dependencies developed by T during its execution. This ensures that, upon reading a version of object x, T will causally depend on the transaction T' that wrote that version of x, along with the causal dependencies of T'.

Consider the example depicted in Figure 3.6a, which shows a complete execution of the protocol. Client c_1 issues a pair of read requests to servers s_1 and s_2. In turn, these servers reply with the value of the object requested, along with its version vector and the new aggregate vector for the transaction, denoted by Vdep and Vaggr at the bottom. Since this is the first read request issued on behalf of this transaction, the server s_1 replies with its



(a) Transaction Execution (b) Transaction Termination

Figure 3.6: *Message flow diagram of a sample execution, with message arrows labeled with their names in the protocol. The different interactions between a client and servers are shown in the shadowed regions. The operation performed by the client is shown at the top, and the state observed by the client at the end of each message exchange is summarised at the bottom.*

most recent aggregate vector, stored in Vtotal. In addition, it replies with the most recently updated version of x—the object being read—which in this case is the empty version vector, $\vec{0}$, signalling the client that the object x has never been updated before. The aggregate vector $\langle 2, 0\rangle$ signals the client that the transaction snapshot must cover committed transactions with a sequence number at s_1 no larger than two, and that those transactions have no causal dependencies at s_2, indicated by having the entry for s_2 set to zero. The client incorporates these two vectors to the transaction context, and issues a new read request to s_2. The server, in turn, must use the transaction's snapshot vector, T.Vsnap, to compute the snapshot of T. Following the requirements already introduced, the snapshot at s_2 must cover the transactions already included in T's snapshot at s_1. That is, it must include transactions committed at s_2 with a sequence number at s_1 no larger than two. Since at s_2 no committed transactions have causal dependencies on transactions that committed at s_1, the server is free to return its most recent aggregate vector, equal to $\langle 0, 3\rangle$, along with the most recent version of the object requested. The client updates the context of the transaction with the dependency and aggregate vectors sent by s_2, which leaves the transaction with a dependency vector equal to $\langle 0, 2\rangle$ and a snapshot vector equal to $\langle 2, 3\rangle$.

3.4.2. Transaction Termination

To terminate a transaction, clients submit the transaction *context* to the servers for validation against both previous and concurrently executing transactions. This validation is described by referring to the algorithms in Figures 3.7 and 3.8.

A client executing a transaction T tries to commit it by calling the `commit` function (line 30). Read-only transactions are committed without any additional checks, since they read a causally consistent snapshot (line 31). To commit an update transaction T, the client executing T serves as the coordinator for a two-phase commit among the processes that store the objects written by T.

In more detail, to commit a transaction T, the client first sends a `PREPARE` message to all servers storing the objects written by the transaction (line 33). The message contains T's write-set and its dependency vector $T.\mathsf{Vdep}$. When a server s_i receives this message (line 47), it *validates* the transaction, deciding whether it should commit or abort due to conflicting concurrent transactions. The server then replies to the client with a *vote* representing its decision. Based on the votes, the client determines the final decision on the transaction—the transaction commits if all votes are positive—and distributes the decision to the relevant servers.

```
30 function commit(T)
31     if T.WS = ∅ then
32         return COMMIT;
33     forall s_j ∈ partitions(T.WS) do
34         send PREPARE(T, T.WS, Vdep) to s_j;
35     Vcomm ← T.Vdep;
36     decision ← COMMIT;
37     forall s_j ∈ partitions(T.WS) do
38         wait receive VOTE(m) from s_j;
39         if m = ⟨T, ABORT⟩ then
40             decision ← ABORT;
41             break;
42         else if m = ⟨T, COMMIT, k⟩ then
43             Vcomm[j] ← k;
44     forall s_j ∈ partitions(T.WS) do
45         send DECIDE(T, Vcomm, decision) to s_j;
46     return decision;
```

Figure 3.7: *Commit protocol of a transaction at client c_i. The client acts as a coordinator for the commit.*

The mechanism to compute a vote on a transaction at server s_i is now described (line 48). The vote for a transaction T ensures the write-conflict freedom property of PSI:

- *For any pair of distinct transactions T_1 and T_2 writing to an object x, if T_1 precedes T_2 in the commit order, then T_2 must see a version of x at least as up-to-date as the one written by T_1.*

```
47 when received PREPARE(T, WS, Vdep) from c_j
48   if (∃T'. (⟨T', PENDING, WS'⟩ ∈ CommitQueue
                ∨ ⟨T', DECIDED, _, _⟩ ∈ CommitQueue)
                ∧ (WS' ∩ WS ≠ ∅)
       ∨ (∃x. x ∈ WS ∧ (VersionLog[x].last.Vcomm[i] > Vdep[i]))
     then
49     send VOTE(t, ABORT) to c_j;
50     return;
51   LastPrep ← LastPrep + 1;
52   CommitQueue.put(T, PENDING, WS);
53   send VOTE(T, COMMIT, LastPrep) to c_j;
54 when received DECIDE(T, Vcomm, decision) from c_j
55   if decision = COMMIT then
56     CommitQueue.update(⟨T, DECIDED, _, Vcomm⟩);
57   else
58     CommitQueue.remove(T);
59 upon ⟨T, DECIDED, WS, Vcomm⟩ = CommitQueue.head()
60   forall {⟨x, v⟩ | ⟨x, v⟩ ∈ WS ∧ partition(x) = i} do
61     VersionLog.add(⟨x, v, Vcomm⟩);
62   Vtotal ← max(Vtotal, Vcomm);
63   CommitLog.add(T, Vtotal);
64   CommitQueue.remove(T);
```

Figure 3.8: *Commit protocol of a transaction at server s_i.*

To ensure this property, the server s_i first check whether, for the objects that T updated, the versions that T read are still the most up-to-date ones in the database of s_i at the time of validation. The server then checks whether T conflicts with any transactions present in CommitQueue, which have started committing at the server, but whose writes have not been applied to the database yet (line 48). If any such transaction T' in CommitQueue writes to the same object as T, the server should abort T. The writes of transactions in CommitQueue may be applied to the database before the writes of T, so checking for such conflicts ensures that the reads by T will still be up-to-date when its writes are applied to the database.

If the validation for transaction T fails, the server sends a VOTE message with the vote ABORT, which tells the client to abort the transaction (line 49). If the validation succeeds, the server generates a sequence number for the transaction by incrementing LastPrep. Then, it sends to the client a VOTE message with this sequence number and a vote COMMIT (line 53). The server also adds an entry $\langle T, \text{PENDING}, T.\text{WS}\rangle$ to CommitQueue, to record the fact that

it is trying to commit T at the server (lines 51–53).

The client waits until it receives `VOTE` messages from all involved servers (line 38). If all the servers voted COMMIT, the client constructs the final commit vector for T by changing its dependency vector T.Vdep, so that it not only covers the updates of T's causal dependencies, but also the updates of T. These writes are identified by the sequence numbers provided by the servers as part of their `VOTE` messages. Thus, the commit vector is defined by setting the entries in T.Vdep for partitions written by T to these sequence numbers (line 43). The client then sends a `DECIDE` message to all the involved servers with the final decision on T along with its commit vector (line 45).

Upon receiving a COMMIT decision on a transaction T (line 54), a server updates the entry associated with T in CommitQueue to change its status to DECIDED and record its commit vector. If the transaction is aborted, the server removes T from the queue.

Committed transactions are applied to the database in their order in CommitQueue, i.e., the commit order. When a DECIDED transaction T gets to the head of the queue (line 59), the server dequeues it and adds its writes to VersionLog, tagged with its commit vector. The server then joins T's commit vector to Vtotal, and adds the transaction to CommitLog, along with Vtotal as its aggregate vector.

3.5. Consistency Tradeoffs and Read Aborts

As mentioned in 3.4.1, a transaction T reads a causally consistent snapshot from a partition when two properties are maintained: on the one hand, if the snapshot of T excludes a particular transaction T', then T must exclude T', together with any other transaction that causally depends on it, from the snapshots T fixes at any other partition. On the other hand, if the snapshot of T includes a transaction, that transaction—along with its causal dependencies—must be included in any other snapshot fixed by T at any other partition. These two properties provide fastPSI with part of its consistency guarantees: transactions accessing objects in a single partition are executed under Snapshot Isolation (SI), whereas transactions accessing objects in different partitions are executed under Parallel Snapshot Isolation (PSI).

However, it might be that both consistency models cannot be guaranteed at the same time. Since Snapshot Isolation requires a strict order between snapshots, this means that fastPSI must provide a strict per-partition order of snapshots, which is represented by the commit order at a partition. At the same time, under Parallel Snapshot Isolation, partitions are allowed to commit transactions in different order as long as those transactions are non-conflicting. When a transaction cannot build a snapshot that satisfies both consistency models, it is forced to abort.

Recall from 3.3 the usage of version vectors to represent causal dependencies, such that a transaction T_k causally depends on T_j if $T_j.\mathsf{Vcomm} \sqsubseteq T_k.\mathsf{Vdep}$, i.e., T_k is causally dependent on T_j if T_k observes the updates of T_j. At the same time, the commit order of transactions at a partition i is derived from the sequence numbers of transactions at i, such that T_j committed before T_k if $T_j.\mathsf{Vcomm}[i] < T_k.\mathsf{Vcomm}[i]$.

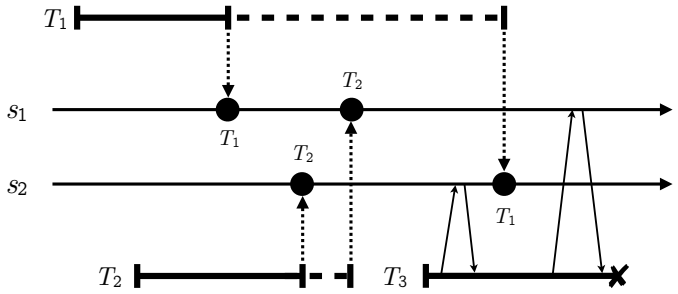


Figure 3.9: *Example read abort execution. Bold horizontal lines represent executions of transactions over time (left to right), the black circles represent the commit times of the transactions at different servers, and the dotted bold lines represent time it takes for the commit of a transaction to propagate to the other servers. Arrows between transactions and servers represent messages. The cross in the execution of T_3 marks its abort time.*

In practice, since the dependency vector Vdep of a transaction is used as the initial value of its commit vector (line 35), this makes any two transactions committing at the same partition causally dependent on each other, even if they are not aware of each other's updates. Nevertheless, this causal dependency is only established after both transactions are committed, and thus it doesn't interfere in their validation process.

This limitation comes from the strict per-partition commit order requirement of SI and from the coarse-grained nature of version vectors. When combined with the property of PSI that allows different commit orders across partitions, it can cause situations such that any snapshot that observes a particular order of transactions will not be causally consistent. When this happens, any transaction that observes such order will need to abort.

The example in Figure 3.9 illustrates such an order, with T_1 and T_2 being a pair of concurrent, non-conflicting transactions, both committing at servers s_1 and s_2. Due to the strict per-partition order requirement inherited from SI, both servers order T_1 and T_2 according to their local sequence number. In addition, the weaker consistency guarantees of PSI allow servers to commit transactions in different order, such that s_1 observes $T_1 \lhd T_2$, and s_2 observes $T_2 \lhd T_1$. Usually, this divergence in the commit order of transactions doesn't affect the ability of subsequent transactions to form a causally consistent snapshot: as long as a transaction is able to observe both T_1 and T_2 at every partition, their respective commit orders are irrelevant. However, consider the execution of T_3 in Figure 3.9. The transaction reads an object from server s_2, including T_2 in its snapshot; however, the read operation of T_3 arrives at s_2 before the commit of T_1 propagates to the server, and therefore T_3 excludes T_1 from its snapshot. Following this, T_3 attempts to read an object from s_1. Since T_3 included T_2 in its snapshot at s_2, it must also include it in its snapshot at s_1; on the other hand, since T_3 excluded T_1 at s_2, its snapshot must not include T_1 at s_1. However, due to the fact that s_1 observed the order of both transactions as $T_1 \lhd T_2$, the two requirements to

form a causally consistent snapshot for T_3 are contradictory: including T_2 in its snapshot would force T_3 to observe T_1, whereas excluding T_1 would force T_3 to also exclude any other transaction that causally depends on T_1, i.e., T_2. In this situation, T_3 cannot form a causally consistent snapshot at s_1, and thus it must abort.

Chapter 4

Implementation and Evaluation

This chapter offers an evaluation that attempts to explore the overhead of fastPSI's strong consistency model compared to the weak consistency of Read Committed. The evaluation also shows how fastPSI is able to outperform a protocol implementing the stronger serialisability consistency model. Finally, it evaluates the scalability of fastPSI as more servers and partitions are added to the system, and discusses some of the limitations of the protocol.

4.1. Implementation

The implementation of fastPSI consists of a client-side library [1] and a server [3], the latter being written as a plug-in transactional protocol for Antidote [5], a reference platform for evaluating consistency protocols. Both the client library and the server are written in the Erlang programming language, with a total of 6K lines of code. The Antidote platform provides a key-value database, supports both in-memory and disk-based storage, and implements full replication. For simplicity, the implementation of fastPSI only supports in-memory storage, and lacks a replication mechanism. The client-side library communicates with the server using Google's Protocol Buffers [2]. To enhance network efficiency, client messages are transmitted in periodic batches to the servers.

To validate the results of the evaluation, two alternative protocols are also implemented, satisfying the Read Committed and serialisability consistency models, called **naiveRC** and **naiveSER**, respectively. Both are built on top of the original fastPSI implementation and are as efficient as possible. The pseudocode for both implementations can be found in Appendix A.

As the implementation of fastPSI doesn't target replicated scenarios, this document refrains from comparing against previous implementations of Parallel Snapshot Isolation. Since the protocols and implementations as described in the literature are influenced by the choice of replication mechanisms, a comprehensive evaluation of fastPSI against other implementations of PSI is deferred to future work, which could explore incorporating either partial or full replication to fastPSI.

Given that fastPSI requires the use of multiple versions per object, it becomes necessary to prevent an unbounded growth of the number of versions in the VersionLog database, and in the number of entries in the per-partition CommitLog. To that end, a simple garbage collection mechanism in the implementation ensures a fixed number of versions, and regularly prunes the oldest versions from the state of a partition.

4.2. Evaluation

This section evaluates the performance of fastPSI using several workloads inspired by the Yahoo! Cloud Serving Benchmark (YCSB) [18], modified to generate transactional workloads [6, 9]. The implementation of Read Committed is used as a baseline for comparison, in order to show the maximum possible performance. Figure 4.1 describes the workloads used. All experiments are run on a cluster consisting of machines running Debian 4.19.67-2 (Stretch) with 3.80 GHz to 4.70 GHz Intel Xeon processors with six cores, 32 GB of RAM, and one gigabit network port. The cluster is partitioned in up to three different sites, with four server machines and four client machines at each site. Thus, there is no shared memory between clients and servers, as if clients were acting as proxies in the same data centre as servers. Since all the machines are located in the same local network, the `tc` Linux command is used to artificially add latency between sites.

In all benchmarks, the system is loaded with one million random keys and 256-byte values prior to receiving any operations from the clients. Partitions are distributed uniformly across servers, such that a server might be responsible for multiple partitions. Keys are mapped to partitions using consistent hashing [30], with clients being aware of the distribution of keys to partitions and server machines. Thus, clients can directly address the correct server and partition for a specific key. Each client machine spawns multiple concurrent threads that execute transactions and communicate with servers in a closed-loop fashion. When transactions read more than one object, clients perform those operations serially. For the experiments that involve more than one site, the latency between sites is of 10 ms.

	Key Selection Distribution	Operations	
		Read-Only Tran.	Update Tran.
B	Uniform	4 Reads	3 Reads, 1 Update
C	Uniform	2 Reads	1 Read, 1 Update
D	Uniform	3 Reads	3 Reads, 1 Update
E	Uniform	3 Reads	3 Reads, 3 Updates

Figure 4.1: *Transactional YCSB Workload Types.*

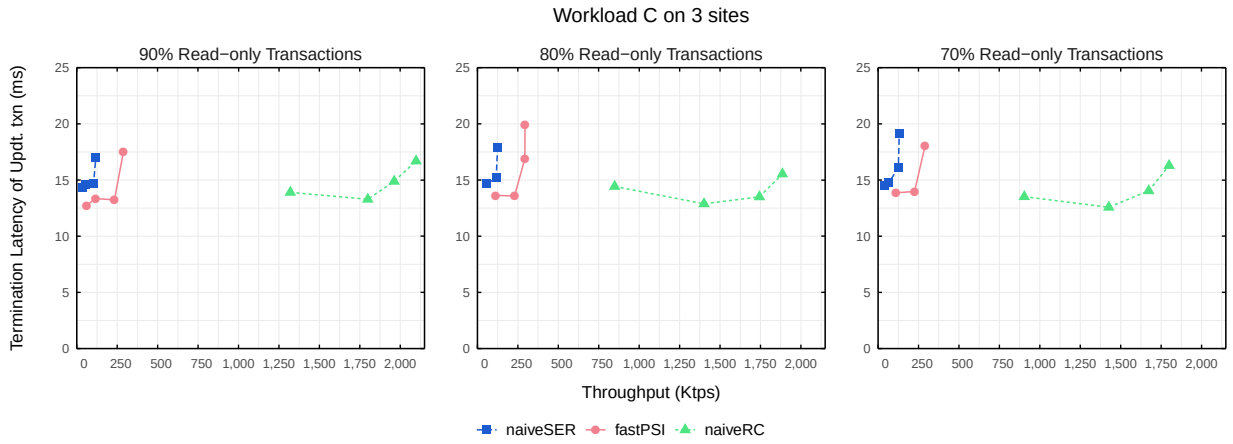


Figure 4.2: *Comparison of throughput and termination latency of update transactions.*

4.2.1. Performance & Scalability Limits

Performance. The first experiment serves to investigate the overall performance profile of the implementations. The throughput and latency of the different protocols is measured and compared as the number of update transactions increases, while keeping the number of sites constant. For this experiment, the number of concurrent client threads varies such that the resources of the CPU never saturate. The aim is to explore the overall overhead of fastPSI in comparison with naiveRC, as well as the performance benefits it offers in comparison with naiveSER. Figure 4.2 shows the results of using Workload C and three sites, with 64 partitions uniformly distributed across sites. It measures the termination latency of update transactions (i.e., the amount of time spent on the validation of a transaction) as the ratio of read-only to update transaction ranges from 90%/10% to 70%/30% (left to right). Since the latency is measured in the client, it only reflects the amount of time spent on the *prepare* phase of the commit validation, as the client does not need to wait until the changes of a transaction are committed to the partition state.

Transactions as executed by naiveRC need minimal synchronisation during its commit phase, and no synchronisation at all during read operations. This is reflected in its high performance, with the validation of update transactions as the only bottleneck in the system. Thus, as the proportion of update transactions increases, the impact on overall throughput is pronounced, dropping by as much as 20%.

For both naiveSER and fastPSI, read operations from a transaction T must wait until the causal dependencies of T commit at a particular partition, bounded in the worst case by the maximum latency across sites. In addition, read operations accessing the same partition suffer from having to synchronise while fixing a snapshot, as the implementation of CommitLog is not thread-safe. These two shortcomings explain the overall low throughput in comparison with naiveRC. Nevertheless, both implementations exhibit stable performance as the proportion of update transactions increases.

By comparing fastPSI with naiveSER, one can observe that the latter implementation is limited by its need to validate every transaction, in comparison with fastPSI, which only validates update transactions. In addition, the weaker consistency model offered by fastPSI

allows it to outperform naiveSER in all cases by approximately 150%, while showing similar latencies.

Scalability. The next experiment explores the overall scalability of fastPSI, by examining how the maximum performance of each protocol changes as the number of machines in the system is increased. Workload B is used, with a fixed ratio of 10% update transactions, and the number of sites is varied from one to three, while keeping the number of partitions fixed to 64. Figure 4.3 shows the overall performance of the protocols.

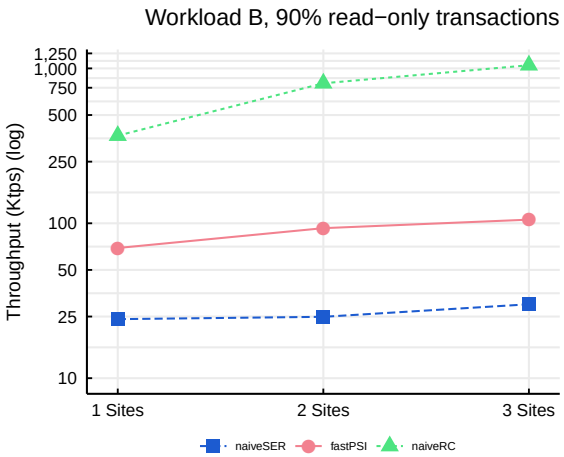


Figure 4.3: *Maximum Throughput of Consistency Models.*

As before, the performance of naiveRC increases almost in a linear fashion as more servers are added, as explained by its minimum need for synchronisation. Although the scalability of fastPSI is limited by the fixed number of partitions, it benefits moderately from increasing the number of machines: as the overall number of partitions per machine decreases, servers free resources, and can thus fulfil more client requests. This is reflected in its performance at three sites being 1.52 times its base throughput at a single site. In contrast, the overall performance of naiveSER stays almost constant as the number of sites is increased, reflecting its need to validate every transaction, which requires greater levels of synchronisation.

As the number of sites increases, so does the difference between naiveSER and fastPSI. Overall, fastPSI manages to outperform naiveSER by a factor of 2.88 with a single site, and by a factor of 3.52 at three sites.

Parameter	**Range**	**Default**
Sites	1–3	3
Update Tran. Proportion	10%–30%	10%

Figure 4.4: *Parameter space used in the comparison workload.*

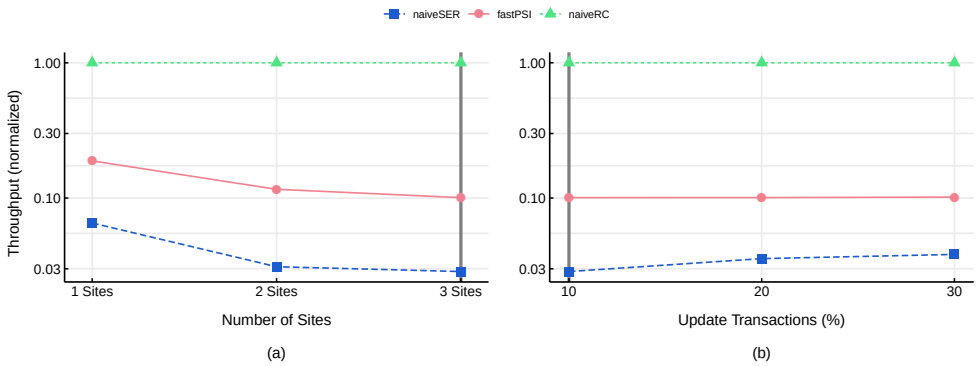


Figure 4.5: *Parameter space exploration to reflect the performance comparison of the protocols. Each experiment varies one parameter while keeping the other fixed at its default value (represented by the grey vertical line). Throughput is shown normalised compared to naiveRC.*

Overall comparison. The last two experiments have shown how the performance and scalability of the implementations is determined by the proportion of update transactions and the number of sites. To better visualise the relationship between workload choice and performance, the next experiment explores the parameter space described in Figure 4.4 when using Workload B. As in the previous experiment, the number of partitions is kept constant as the number of sites increases. The results are shown in Figure 4.5, with the throughput depicted normalised compared to the performance of naiveRC.

As shown in Figure 4.5a, fastPSI and naiveSER have different scalability properties. Although both implementations suffer in comparison with naiveRC, fastPSI exhibits much better scalability in comparison with naiveSER. For naiveSER, the need to validate every transaction imposes a performance penalty that increases as more sites are added, and consequently increases the overall latency of the commit phase for every transaction. In contrast, the impact on fastPSI is less severe, as the increased latency only affects the read operations, since the proportion of update transactions is low.

Figure 4.5b shows the performance comparison as the proportion of update transactions increases. While the difference in throughput between naiveRC and fastPSI stays constant, the overall performance of fastPSI is hindered by the need to compute a causally compatible snapshot. Nevertheless, the fact that the relative performance stays constant in comparison with naiveRC shows that the impact of the validation process does not grow with the number of update transactions. The impact of update transactions on naiveSER is less pronounced, and its performance difference with fastPSI gets smaller as the proportion of updates increases. This is explained by the choice of workload: read-only transactions perform four reads, while update transactions perform three. Since naiveSER also validates read-only transactions, as the proportion of updates grows, the average number of partitions that participate in the voting phase shrinks from four to three, which decreases the runtime cost of validation.

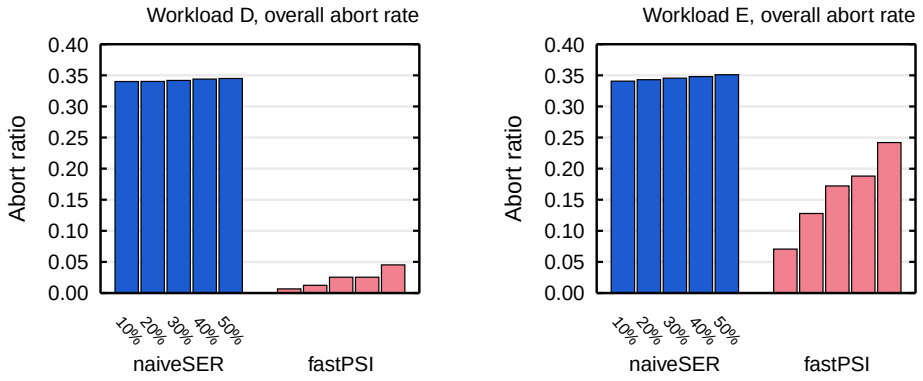


Figure 4.6: *Overall transaction abort ratio for different consistency models and workloads.*

4.2.2. Abort Ratio

This section focuses on another advantage of the relaxed consistency model of fastPSI in comparison with serialisability, namely, the ratio of aborted transactions. One of the characteristics of fastPSI is that the snapshots of transactions exhibit *forward freshness*: a transaction T is able to read object versions written by transactions that committed after T started. In contrast, a transaction T under serialisability or classical Snapshot Isolation can only read versions written by transactions that commit before T's start time. This limitation leads to a high number of aborted transactions due to stale reads in high latency settings.

As such, this section aims to explore the advantages of forward freshness in fastPSI in comparison with naiveSER, and how different workloads affect it. Figure 4.6 shows how the abort ratio of transactions varies—under different workloads—with the number of update transactions. The results were obtained with two sites. The graph on the left shows the advantages of the forward freshness of transactions, with the abort ratio of fastPSI being 30 percentage points better than naiveSER, on average. However, as the number of updates increases, so does the number of conflicting transactions. This is reflected in an increase of aborted transactions, from less than one percent to around five percent. The graph on the right, however, shows different results: as the number of update transactions grows, so does the overall abort ratio of fastPSI, from 7% to 25%. Overall, in the worst case the abort ratio of fastPSI is only 10 percentage points better than naiveSER.

The difference between workloads is explained in the two graphs depicted in Figure 4.7, which explores the reasons why transactions abort. In fastPSI, a transaction might abort for two reasons: by updating an object that is overwritten by a concurrent transaction, or by failing to form a causally consistent snapshot, as detailed in 3.5. By virtue of having the same implementation of causally consistent snapshots, naiveSER inherits the same reasons, and adds a third: a transaction is forced to abort if it reads a version of an object that is later overwritten. For both protocols, a transaction might abort at two points during its execution: read aborts occur when a transaction fails to fix a causally consistent snapshot, and otherwise the transaction aborts during validation, i.e., during the termination of the

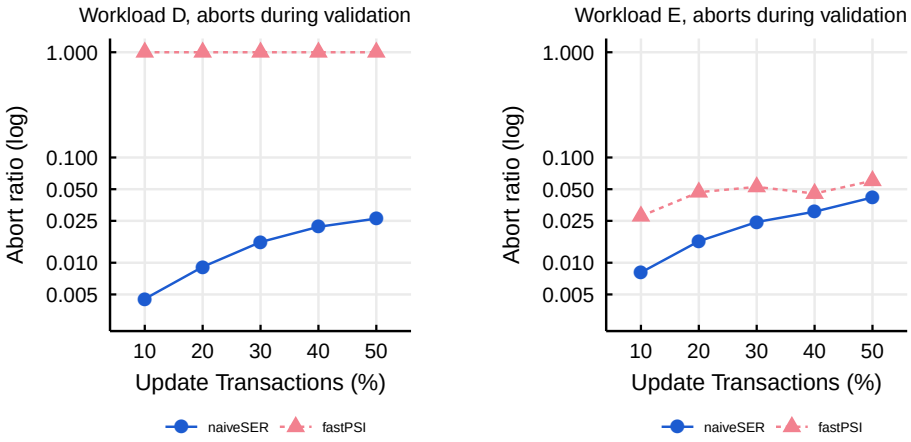


Figure 4.7: *Ratio of aborts that happen during the validation phase, for different consistency models and workloads.*

transaction. As seen on the left graph in Figure 4.7, in Workload D all aborted transactions in fastPSI do so during the validation phase, while for naiveSER, only a small percentage of aborted transactions (between 0.5% and 2.5%) do so during termination. In contrast, fastPSI exhibits a very different behaviour with Workload E, as shown in the graph on the right. With this workload, transactions abort for the same reason in both protocols: they are unable to build causally consistent snapshots. In both cases, the amount of transactions that abort while attempting to fix a snapshot grows as the number of update transactions increases.

Recall that in Workload D, update transactions update one single key, while in Workload E, they update three. This small difference in the number of updated keys explains the difference in aborted transactions for fastPSI. As described in 3.5, a transaction T will be unable to fix a causally consistent snapshot when a) different partitions commit non-conflicting transactions in different order, and b) T fixes a snapshot in one of those partitions before the updates of all the non-conflicting transactions become visible. Since update transactions in Workload D only update a single key, update transactions will only commit at a single partition, and therefore, no transaction can observe a different commit order. This explains why there are no aborted transactions due to inconsistent snapshots when executing this workload for fastPSI. In contrast, update transactions in Workload E update three keys, and therefore commit at three different partitions,[1] such that the probability of partitions committing transactions in different order grows. In addition, since transactions read three keys, the probability of observing different commit orders also grows. Thus, as the number of update transactions increases, so does the probability of transactions attempting to fix inconsistent snapshots. In naiveSER transactions also commit at the partitions they read from, meaning that update transactions in Workload D commit at three partitions.

[1]Since transactions choose keys following an uniform distribution, most keys will be managed by distinct partitions.

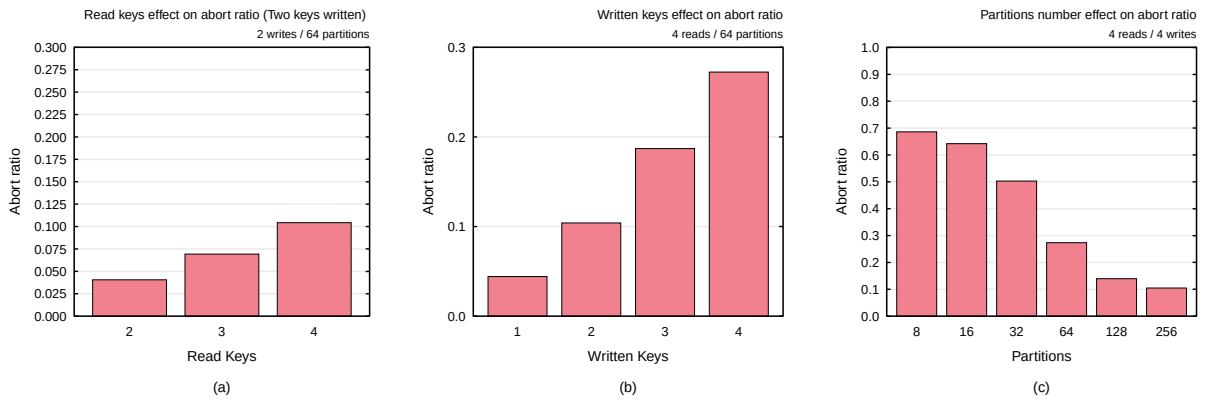


Figure 4.8: *Results from exploring the abort ratio of fastPSI across different workload and deployment scenarios.*

This explains why its abort ratio is similar in both workloads.

Since small changes in workload choice are able to affect the overall abort ratio of fastPSI in a significant manner, the evaluation is concluded with an exploration of the parameters that have the highest impact on the number of aborted transactions. As explained previously, one of the reasons why transactions observe inconsistent snapshots is because partitions commit transactions in different orders. Thus, by changing the number of keys updated by a transaction, one can modify the number of partitions involved, which affects the chances of different commit orders. Another reason for inconsistent snapshots is that transactions are able to observe the transactions that commit in different order, which leads to change the number of keys read by the transactions. If transactions read from a small amount of partitions, the probability that they observe different commit orders will also be small. Finally, the most important factor is the number of partitions, or rather, the amount of keys per partition. When a few partitions manage all the keys, most transactions will commit at the same partitions. Therefore, the probability of having different commit orders grows, as well as the probability that a given transaction observes those orders. Conversely, when the number of partitions is large, or partitions manage only a small amount of keys, the probability of different commit orders shrinks, as does the probability of them being observed by transactions.

Figure 4.8 shows the results of modifying each of the parameters mentioned, noting that, for simplicity, a single site is used while modifying the number of partitions, instead of changing the total amount of keys in the database. The percentage of update transactions is set to 50%, to show the worst possible abort ratio. The effect of changing the number of keys read by a transaction, shown in Figure 4.8a, is small. Although the overall abort ratio never grows larger than 10%, modifying the number of read keys results, at best, in an improvement of 7.5 percentage points. Blind updates are not allowed, and as such the minimum number of read keys is two, since transactions need to update at least two keys in two different partitions to introduce inconsistent snapshots in the system. In contrast, changing the number of keys updated by a transaction produces a bigger impact, as shown

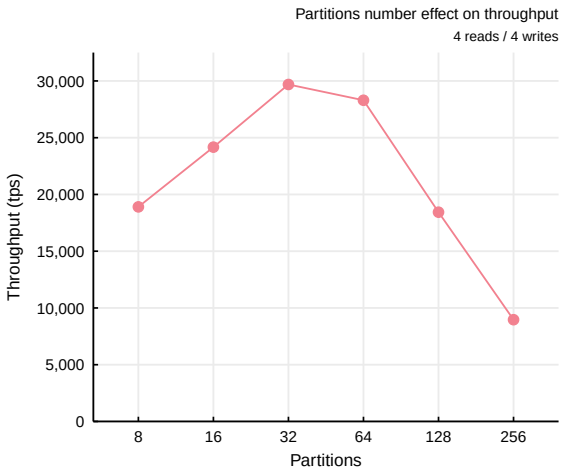


Figure 4.9: *Performance degradation of fastPSI as the number of partitions increases, with a fixed number of servers.*

in Figure 4.8b, with an overall change of 23 percentage points in the number of aborted transactions. When transactions update a single key, all transactions abort during validation. At this point, the system shows the best possible abort ratio for 64 partitions, at 5%. Finally, modifying the number of partitions yields the biggest change in the number of aborted transactions, as shown in Figure 4.8c. With 8 partitions, the overall abort ratio is of almost 70%, which serves as an extreme example of the importance of this parameter. As the number of partitions grows, the system reaches an abort ratio of 27%. By increasing the number of partitions to 256, the result is an abort ratio of 10%. With an overall difference of 60 percentage points in the proportion of aborted transactions, this shows that the number of partitions is the biggest influence in the abort ratio of transactions in fastPSI. It is important to note, however, that increasing the number of partitions also has a big impact on the performance of the system, as shown in Figure 4.9. During the experiment, the maximum throughput is reached at 32 partitions across 4 machines. With a small number of partitions, the increased contention causes the throughput to drop. With a big number of partitions, each server machine is also responsible for a big number of partitions. As a result, the system overloads. Thus, the number of available machines constraints the number of partitions.

Chapter 5

Related Work

This chapter gives an overview of the previous work in the settings of transactional protocols, consistency, and application robustness. It also highlights the main differences between the contributions of this work and previous approaches.

Application Robustness. The notion of robustness as applied to databases was first investigated by Fekete et al. [22], proposing a way to analyse if applications were robust against Snapshot Isolation (SI) [13]. The work of Fekete et al. has resulted in the proliferation of static analysis tools for detecting the presence of anomalies in applications [29], as well as several run-time techniques for ensuring serialisable transactions [37]. More recently, Bernardi and Gotsman [14] proposed several robustness criteria for a variety of consistency models, including Parallel Snapshot Isolation (PSI) [39]. The work on fastPSI builds on the robustness criteria of Parallel Snapshot Isolation and of Snapshot Isolation to build a hybrid protocol that allows programmers to selectively strengthen consistency guarantees for individual transactions.

Entity Groups. The concept of *entity* was introduced by Helland [28] to refer to a self-contained data object, with application-defined boundaries, and uniquely identified by an entity key. In addition, Helland argued for the entity to be the largest *scope of transactional serialisability*: transactions can only guarantee atomicity for objects held within the same entity, and are prevented from modifying objects across entities.

Later systems, such as Megastore [11], introduced the concept of *entity groups* as disjoint aggregations of individual entities. Such systems allowed transactions to access distinct entities, and offered strong consistency for transactions accessing entities within a group, while offering almost no consistency guarantees for transactions that accessed different groups. Such systems thus broadened the scope of transactional serialisability to encompass entire entity groups. In contrast, fastPSI provides PSI for all transactions, even those that access objects in multiple entity groups. At the same time, it strengthens the consistency guarantees further for transactions accessing only individual entity groups, by providing SI.

Strong Consistency Protocols. Although there is a large variety of protocols providing strong consistency, this section focuses on those that provide similar guarantees to those in fastPSI, as well as others from which inspiration was drawn during the design of the fastPSI protocol.

Walter, designed by Sovran et al. [39], is a transactional replicated key-value store that offers Parallel Snapshot Isolation. To track causality relations between transactions, Walter uses vector timestamps with one entry per replica, compared to the version vectors used by fastPSI, with one entry per partition. In contrast with fastPSI, Walter can only offer snapshots with base freshness, as transactions fix a start timestamp at the beginning of their execution. As a result, the system may experience a higher number of aborted transactions due to stale reads. Walter employs a single server per replica to assign monotonic timestamps and to manage transactions, which limits the overall scalability. In fastPSI, timestamps are assigned independently by each partition, and further allows any server to perform the role of transaction coordinator. Although Walter offers partial replication, the combination of replication and the choice of monotonic start timestamp leads the system to perform global communication in the background with all replicas.

Jessy, proposed by Saeida Ardekani et al. [6], is a similar transactional key-value store, and the first system to provide Non-Monotonic Snapshot Isolation (NMSI). Unlike Walter, however, transactions in Jessy observe forward freshness in their snapshots, allowing greater scalability. To support consistent snapshots, Jessy uses a data structure called dependence vector, also similar to a version vector, with a number of entries either equal to the number of objects, or equal to the number of partitions. The former approach suffers from a big overhead, although it allows transactions to read versions of objects as fresh as possible. With one entry per partition, Jessy offers greater scalability, at the cost of introducing spurious conflicts between transactions. The fastPSI protocol follows a similar approach, using version vectors with one entry per partition. Like Walter, Jessy also supports partial replication. However, Jessy relies on atomic multicast primitives [26], instead of the two-phase commit protocol used in Walter and fastPSI.

Another protocol that provides NMSI is Blotter, proposed by Moniz et al. [33]. In contrast with Jessy, Blotter supports full replication through the use of Paxos commit [25]. The design of Blotter shows how the properties of NMSI, like forward freshness, can be used to design protocols with full replication without loss of scalability.

Peluso et al. [36] proposed GMU, a transactional protocol that uses vector clocks to track causal dependencies. Much of the design of fastPSI is inspired by GMU, including its clock mechanism, as well as the approaches to build causally consistent snapshots. The consistency guarantees are different, however, as transactions in GMU satisfy the *Extended Update Serialisability* (EUS) [4, 27] consistency model. EUS guarantees serialisability for update transactions, while read-only transactions can observe different commit orders for non-conflicting transactions, similar to the guarantees offered by the causally compatible snapshots of fastPSI. Even though GMU offers a stronger consistency model, it always commits read-only transactions, in contrast with the possibility of aborts under fastPSI when transactions cannot satisfy SI and PSI for the same snapshot.

Chapter 6

Conclusions and Future Work

6.1. Conclusions

This work has presented fastPSI, a transactional protocol with hybrid consistency that combines Parallel Snapshot Isolation (PSI) with stronger consistency models like Snapshot Isolation (SI) or serialisability. The core of fastPSI is formed by the notions of entity groups and causally consistent snapshots, which allow forward freshness for transactions while allowing serialisable transactions inside individual entity groups. The performance of fastPSI is evaluated in comparison with both strong and weak consistency protocols, demonstrating that fastPSI is able to outperform a serialisable protocol while having similar consistency guarantees. On the other hand, the need to reconcile SI and PSI makes fastPSI susceptible to abort read-only transactions, a major drawback of the protocol in comparison with other proposals. While fastPSI doesn't achieve the performance and scalability of weak consistency protocols like Read Committed, the stronger guarantees provided by fastPSI's hybrid consistency should offer an attractive alternative for applications where both strong consistency and scalability are needed.

6.2. Future Work

In the future, fastPSI should be compared against existing implementations of protocols of Parallel Snapshot Isolation [39] and Non-Monotonic Snapshot Isolation [6, 33], in order to explore the scalability properties and overhead of the contributions of this work in comparison with previous approaches. As most previous protocols provide either partial or full replication, a first step towards a fair comparison would be to explore ways of providing replication in fastPSI, with the work of Moniz et al. [33] on Blotter providing a possible solution.

As discussed in 3.5, one of the main limitations of fastPSI is the possibility of read-only transactions having to abort, due the impossibility of observing a causally consistent snapshot. This limitation is caused by the combination of the hybrid consistency guarantees of the protocol, combined with the coarse grained nature of entity groups. This limitation

can be minimised, as shown in 4.2.2, by increasing the number of entity groups in the system at the cost of lower performance. Future work could investigate ways to further minimise the number of aborts due to inconsistent snapshots without having to change the number of entity groups. One such approach could be similar to the one proposed by Bailis et al. [10], where transactions are able to detect inconsistent reads at run-time, and perform a second round of communication with servers in order to repair the snapshot.

Finally, in the current implementation of fastPSI, the choice of both the number and the size of entity groups is left to the application programmer, as it is influenced by the requirements of the domain. As such, another possibility of future research is the development of tools that aid programmers in making such choices by inferring the entity groups required for application correctness. The work of Gotsman et al. [24] and Najafzadeh et al. [34] pave the way to build static analysis tools that work for hybrid consistency protocols such as fastPSI.

Appendix A

Serialisable and Read Committed Protocols

This appendix gives a quick overview of the protocols implemented to compare against fastPSI. For simplicity, both protocols are minor modifications of the original one, although the implementations are still efficient. For each protocol, the most relevant changes with respect to fastPSI are summarised, accompanied by the full pseudocode. The implementation for both protocols is freely available on GitHub [1, 3].

A.1. Serialisability

Recall from 2.2.2 that under serialisability transactions appear to execute one after the other. However, it does not guarantee a real-time order among them: an implementation is free to reorder the commit order of transactions as long as the resulting execution still satisfies serialisability. Since the original protocol guarantees Parallel Snapshot Isolation, it is sufficient to prevent the *Write Skew* and the *Long Fork* anomalies. To do so, fastPSI is expanded with several changes. A summary of the state of a server is reflected in Figure A.1, and the pseudocode of the protocol can be found in Figures A.2–A.4.

Track versions of read objects. Read-only transactions need to observe consistent data that it's not too stale, or overwritten by concurrent transactions. In order to enforce this guarantee, a new piece of information is added to the transaction context: the *read-set* of a transaction keeps track of the objects that it reads, along with the commit time of the read version of the object. When a client receives a `READRETURN` message after issuing a read of an object x on behalf of a transaction T, it incorporates the commit time of x in T's read-set, $T.\mathsf{RS}$ (line 15). It suffices to store the j-th entry of x's *commit vector*, that is, the *sequence number* assigned to the transaction that wrote the version of x read by T. The read-set of a transaction is initialised to the empty set (line 2) when the transaction starts.

Avoid stale reads. Recall from Section 2.2.4 that the Long Fork anomaly occurs whenever concurrent transactions are able to observe different orderings of non-conflicting transactions. This anomaly can be precluded by forcing transactions to observe the latest version of an object. The protocol enforces this property during the validation of a transaction. When a transaction T prepares to commit, it performs a two-phase commit among the servers storing the objects read and written by the transaction (lines 32–33). Upon receiving a `PREPARE` message, a server s_i checks that, for the objects that T read, the version of those objects is the most up-to-date in the database of s_i (line 47, last condition). It does so by comparing the version of the object stored in the transaction's read-set (RS) against the latest version as reflected by the VersionLog of the object ($\mathsf{VersionLog}[x].\mathsf{last}.\mathsf{Vcomm}$). The server also needs to perform the check against concurrently-committing transactions (line 47, first condition): a server checks that the read-set of a transaction T does not overlap with the write-set of any other concurrently-committing transaction, thereby making T's read stale. If the version that T read is stale, the server s_i votes ABORT (line 48). Since it is assumed that transactions read an object before writing to it (and thus, $T.\mathsf{WS} \subseteq T.\mathsf{RS}$) the stale read check also precludes stale writes.

Avoid read-write conflicts. The Write Skew anomaly, introduced in Section 2.2.3, can be precluded by forcing transactions to observe the writes of concurrent transactions as soon as those updates are reflected in the state of the database. A history that exhibits Write Skew, such as $h = r_1(x_0).r_1(y_0).r_2(x_0).r_2(y_0).w_1(x_1).w_2(y_2).c_1.c_2$, can only be serialisable by aborting either T_1 or T_2. In the protocol, this is avoided by precluding *read-write* conflicts: in the previous history, T_2 *overwrites* the version of y that T_1 read. Also, T_1 overwrites the version of x that T_2 reads, which is also considered a read-write conflict. Both kinds of conflict can be precluded during the validation of a transaction (line 47, first condition): a server checks that the write-set of a transaction T does not overlap with the read-set of any other concurrently-committing transaction (thereby invalidating the read of such concurrent transaction). If any of the checks is true, the server votes ABORT (line 48).

Variables at a server s_i		
Name	**Domain**	**Description**
LastPrep	Integer	The number of update transactions that tried to commit at the server.
VersionLog	Map[Object, Set[⟨Value val, VerVector Vcomm⟩]]	Database: a mapping from objects to lists of pairs of a value and the commit vector of the transaction that wrote it. The lists are ordered by the i-th component of the commit vectors.
CommitLog	Sequence[⟨Tx T, VerVector Vaggr⟩]	Log of update transactions T committed at the server, ordered by Vaggr[i]. Here Vaggr is the aggregate vector of T: the join of the commit vectors of all transactions up to T in CommitLog.
Vtotal	VerVector	The join of the commit vectors of all transactions in CommitLog.
CommitQueue	Sequence[⟨Tx, State, ReadSet, WriteSet⟩] where State = {PENDING, DECIDED}	Queue containing information about update transactions trying to commit at the server.
Context for a transaction T at a client c_i		
T.RS	ReadSet	Read-set of T.
T.WS	WriteSet	Write-set of T.
T.HasRead	Vector[Bool]	Mapping showing whether T has read a given partition.
T.Vsnap	VerVector	Snapshot vector: determines snapshots fixed at partitions T has read from and possible causal dependencies at all other partitions.
T.Vdep	VerVector	Dependency vector, representing all causal dependencies developed by T during its execution.

Figure A.1: *List of variables used in the Serialisable protocol, where* ReadSet = Set[⟨Object, Integer⟩] *and* WriteSet = Set[⟨Object, Value⟩]

```
1 function start()
2     return new Tx(WS = ∅, RS = ∅, HasRead = ⊥⃗, Vsnap = 0⃗, Vdep = 0⃗);
3 function write(T, x, v)
4     T.WS ← (T.WS \ {⟨x, _⟩}) ∪ {⟨x, v⟩};
```

Figure A.2: *Initialisation of a transaction and update of an object* x *at client c_i under serialisability.*

```
5  function read(T, x)
6      if ⟨x, v⟩ ∈ T.WS then
7          return v;
8      j ← partition(x);
9      send READREQUEST(x, T.Vsnap, T.HasRead) to s_j;
10     wait receive READRETURN(m) from s_j;
11     if m = ABORT then
12         throw ABORT;
13     else if m = ⟨v, Vdep, Vaggr⟩ then
14         T.HasRead[j] ← ⊤;
15         T.RS ← (T.RS \ {⟨x, _⟩}) ∪ {⟨x, Vdep[j]⟩};
16         T.Vdep ← max(T.Vdep, Vdep);
17         T.Vsnap ← max(T.Vsnap, Vaggr);
18         return v;
19 when received READREQUEST(x, Vsnap, HasRead) from c_j
20     if HasRead[i] then
21         V ← Vsnap;
22     else
23         wait until Vtotal[i] ≥ Vsnap[i];
24         r ← max{r ∈ CommitLog | ∀j. HasRead[j] ⟹ (r.Vaggr[j] ≤ Vsnap[j])};
25         if r.Vaggr[i] < Vsnap[i] then
26             send READRETURN(ABORT) to c_j;
27             return;
28         V ← r.Vaggr;
29     ver = max{ver ∈ VersionLog | ver.Vcomm[i] ≤ V[i]};
30     send READRETURN(ver.val, ver.Vcomm, V) to c_j;
```

Figure A.3: *Serialisable local and remote read of object* x

```
31 function commit(T)
32     forall s_j ∈ partitions(T.RS ∪ T.WS) do
33         send PREPARE(T, T.RS, T.WS, Vdep) to s_j;
34     Vcomm ← T.Vdep;
35     decision ← COMMIT;
36     forall s_j ∈ partitions(T.RS ∪ T.WS) do
37         wait receive VOTE(m) from s_j;
38         if m = ⟨T, ABORT⟩ then
39             decision ← ABORT;
40             break;
41         else if m = ⟨T, COMMIT, k⟩ then
42             Vcomm[j] ← k;
43     forall s_j ∈ partitions(T.RS ∪ T.WS) do
44         send DECIDE(T, Vcomm, decision) to s_j;
45     return decision;
46 when received PREPARE(T, RS, WS, Vdep) from c_j
47     if (∃T'. (⟨T', PENDING, RS', WS'⟩ ∈ CommitQueue
                 ∨ ⟨T', DECIDED, _, _, _⟩ ∈ CommitQueue)
                 ∧ (WS' ∩ RS ≠ ∅ ∧ RS' ∩ WS ≠ ∅)
       ∨ (∃x. ⟨x, vsn⟩ ∈ RS ∧ (VersionLog[x].last.Vcomm[i] > vsn))
      then
48         send VOTE(t, ABORT) to c_j;
49         return;
50     LastPrep ← LastPrep + 1;
51     CommitQueue.put(T, PENDING, RS, WS);
52     send VOTE(T, COMMIT, LastPrep) to c_j;
53 when received DECIDE(T, Vcomm, decision) from c_j
54     if decision = COMMIT then
55         CommitQueue.update(⟨T, DECIDED, _, _, Vcomm⟩);
56     else
57         CommitQueue.remove(T);
58 upon ⟨T, DECIDED, _, WS, Vcomm⟩ = CommitQueue.head()
59     forall {⟨x, v⟩ | ⟨x, v⟩ ∈ WS ∧ partition(x) = i} do
60         VersionLog.add(⟨x, v, Vcomm⟩);
61     Vtotal ← max(Vtotal, Vcomm);
62     CommitLog.add(T, Vtotal);
63     CommitQueue.remove(T);
```

Figure A.4: *Serialisable termination protocol.*

A.2. Read Committed

Read Committed (RC) is the weakest consistency model that satisfies the *isolation* property required by ACID transactions. It forbids concurrent transactions from observing any data that has not been committed, but it does not place any restriction on the ordering of transactions, and does not preclude write-write conflicts. Thus, transactions may be ordered in any way. Figure A.5 shows a summary of the data structures involved in the protocol.

Variables at a server s_i		
Name	**Domain**	**Description**
CommitQueue	Sequence[⟨Tx, State, WriteSet⟩] where State = {PENDING, DECIDED}	Queue containing information about update transactions trying to commit at the server.
Database	Set[⟨Object, Value⟩]	Set representing the key-value store as a mapping from objects to values.
Context for a transaction T at a client c_i		
T.WS	WriteSet	Write-set of T.

Figure A.5: *List of variables used in the Read Committed protocol, where* WriteSet = Set[⟨Object, Value⟩].

Since transactions only need to observe the last committed version of an object, it is sufficient to store only one version. Thus, the VersionLog mapping can be substituted with a Database that simply maps an object to its latest version. In addition, transactions don't need to observe a consistent snapshot of the state of a partition, and therefore all data structures related to computing a snapshot can be removed. This is reflected in the execution of a transaction, as can be seen in Figure A.7. A server s_i executing a remote read on behalf of a transaction T simply fetches the currently available value of the requested object, and returns it to the client (line 13).

A protocol satisfying Read Committed still needs to offer atomic visibility. To do so, the implementation uses two-phase commit to guarantee that a transaction commits at every partition (line 14). Servers that participate during the commit phase always vote COMMIT (line 30), since RC does not preclude write-write conflicts. After a successful commit phase, all servers incorporate the updates of the transaction to its partition state (line 38).

```
1 function start()
2     return new Tx(WS = ∅);
3 function write(T, x, v)
4     T.WS ← (T.WS \ {⟨x, _⟩}) ∪ {⟨x, v⟩};
```

Figure A.6: *Initialisation of a transaction and update of an object* x *at client c_i under Read Committed.*

```
5  function read(T, x)
6      if ⟨x, v⟩ ∈ T.WS then
7          return v;
8      j ← partition(x);
9      send READREQUEST(x) to s_j;
10     wait receive READRETURN(v) from s_j;
11     return v;
12 when received READREQUEST(x) from c_j
13     send READRETURN(Database_i.get(x)) to c_j;
14 function commit(T)
15     if t.ws = ∅ then
16         return COMMIT;
17     forall s_j ∈ partitions(T.WS) do
18         send PREPARE(T) to s_j;
19     decision ← COMMIT;
20     forall s_j ∈ partitions(T.WS) do
21         wait receive VOTE(m) from s_j;
22         if m = ⟨T, ABORT⟩ then
23             decision ← ABORT;
24             break;
25     forall s_j ∈ partitions(T.WS) do
26         send DECIDE(T, decision) to s_j;
27     return decision;
28 when received PREPARE(T) from c_j
29     CommitQueue.put(T, PENDING, WS);
30     send VOTE(T, COMMIT) to c_j;
31 when received DECIDE(T, decision) from c_j
32     if decision = COMMIT then
33         CommitQueue.update(⟨T, DECIDED, _⟩);
34     else
35         CommitQueue.remove(T);
36 upon ⟨T, DECIDED, WS⟩ = CommitQueue.head()
37     forall {⟨x, v⟩ | ⟨x, v⟩ ∈ WS ∧ partition(x) = i} do
38         Database_i.apply(x, v);
39     CommitQueue.remove(T);
```

Figure A.7: *Read Committed execution protocol.*

Bibliography

[1] fastPSI client-side library. URL https://github.com/ergl/pvc/tree/v0.8.0.

[2] Protocol Buffers. URL https://github.com/protocolbuffers/protobuf.

[3] fastPSI Server. URL https://github.com/ergl/antidote/tree/pvc.

[4] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. Ph.D., MIT, Cambridge, MA, USA, March 1999.

[5] Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguica, and Marc Shapiro. Cure: Strong Semantics Meets High Availability and Low Latency. In *Proceedings of the 36th International Conference on Distributed Computing Systems (ICDCS 2016)*, 2016.

[6] 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*, pages 163–172, Sep. 2013. doi: 10.1109/SRDS.2013.25.

[7] Masoud Saeida Ardekani. *Ensuring Consistency in Partially Replicated Data Stores*. Ph.d., UPMC, Paris, France, September 2014.

[8] Masoud Saeida Ardekani, Marek Zawirski, Pierre Sutra, and Marc Shapiro. The space complexity of transactional interactive reads. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, HotCDP '12, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450311625. doi: 10.1145/2169090.2169094. URL https://doi.org/10.1145/2169090.2169094.

[9] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. G-DUR: A middleware for assembling, analyzing, and improving transactional protocols. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, page 13–24, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327855. doi: 10.1145/2663165.2663336. URL https://doi.org/10.1145/2663165.2663336.

[10] Peter Bailis, Alan Fekete, Joseph M. Hellerstein, Ali Ghodsi, and Ion Stoica. Scalable atomic visibility with RAMP transactions. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 27–38, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450323765. doi: 10.1145/2588555.2588562. URL https://doi.org/10.1145/2588555.2588562.

[11] Jason Baker, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011. URL http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf.

[12] Carlos Baquero and Nuno M. Preguiça. Why logical clocks are easy. *Commun. ACM*, 59(4):43–47, 2016.

[13] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data - SIGMOD '95*, 1995.

[14] Giovanni Bernardi and Alexey Gotsman. Robustness against Consistency Models with Atomic Visibility. In Josée Desharnais and Radha Jagadeesan, editors, *27th International Conference on Concurrency Theory (CONCUR 2016)*, volume 59 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:15, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-017-0. doi: 10.4230/LIPIcs.CONCUR.2016.7. URL http://drops.dagstuhl.de/opus/volltexte/2016/6165.

[15] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[16] Eric A Brewer. Towards Robust Distributed Systems (keynote). In *19th ACM Symposium on Principles of Distributed Computing (PODC)*, July 2000.

[17] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. A framework for transactional consistency models with atomic visibility. In *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1-4, 2015*.

[18] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, New York, NY, USA, 2010.

[19] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, and et al. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3), August 2013. ISSN 0734-2071. doi: 10.1145/2491245. URL https://doi.org/10.1145/2491245.

[20] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205, October 2007. ISSN 01635980. doi: 10.1145/1323293.1294281.

[21] Alan Fekete. Allocating Isolation Levels to Transactions. In *Proceedings of the Twenty-Fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '05, page 206–215, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930620. doi: 10.1145/1065167.1065193. URL https://doi.org/10.1145/1065167.1065193.

[22] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making Snapshot Isolation Serializable. *ACM Trans. Database Syst.*, 30 (2):492–528, June 2005. ISSN 0362-5915. doi: 10.1145/1071610.1071615. URL https://doi.org/10.1145/1071610.1071615.

[23] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002. ISSN 0163-5700. doi: 10.1145/564585.564601. URL https://doi.org/10.1145/564585.564601.

[24] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'Cause I'm Strong Enough: Reasoning about Consistency Choices in Distributed Systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 371–384, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335492. doi: 10.1145/2837614.2837625. URL https://doi.org/10.1145/2837614.2837625.

[25] Jim Gray and Leslie Lamport. Consensus on Transaction Commit. *ACM Transactions on Database Systems*, 31(1):133–160, March 2006. ISSN 0362-5915. doi: 10.1145/1132863.1132867. URL https://doi.org/10.1145/1132863.1132867.

[26] Rachid Guerraoui and André Schiper. Genuine atomic multicast in asynchronous distributed systems. *Theoretical Computer Science (Elsevier)*, 254:297–316, 2001. URL http://infoscience.epfl.ch/record/49965.

[27] R. C. Hansdah and Lalit M. Patnaik. Update Serializability in Locking. In *Proceedings of the International Conference on Database Theory*, ICDT '86, page 171–185, Berlin, Heidelberg, 1986. Springer-Verlag. ISBN 3540171878.

[28] Pat Helland. Life beyond Distributed Transactions: an Apostate's Opinion. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, pages 132–141. www.cidrdb.org, 2007. URL http://cidrdb.org/cidr2007/papers/cidr07p15.pdf.

[29] Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S. Sudarshan. Automating the detection of snapshot isolation anomalies. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, page 1263–1274. VLDB Endowment, 2007. ISBN 9781595936493.

[30] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '97, page 654–663, New York, NY, USA, 1997. Association for Computing Machinery. ISBN 0897918886. doi: 10.1145/258533.258660. URL https://doi.org/10.1145/258533.258660.

[31] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998. ISSN 0734-2071. doi: 10.1145/279227.279229. URL https://doi.org/10.1145/279227.279229.

[32] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 401–416, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450309776. doi: 10.1145/2043556.2043593. URL https://doi.org/10.1145/2043556.2043593.

[33] Henrique Moniz, João Leitão, Ricardo J. Dias, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Blotter: Low Latency Transactions for Geo-Replicated Storage. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, page 263–272, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee. ISBN 9781450349130. doi: 10.1145/3038912.3052603. URL https://doi.org/10.1145/3038912.3052603.

[34] Mahsa Najafzadeh, Alexey Gotsman, Hongseok Yang, Carla Ferreira, and Marc Shapiro. The CISE Tool: Proving Weakly-Consistent Applications Correct. In *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data*, PaPoC '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342964. doi: 10.1145/2911151.2911160. URL https://doi.org/10.1145/2911151.2911160.

[35] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Softw. Eng.*, 9(3):240–247, May 1983.

[36] Sebastiano Peluso, Pedro Ruivo, Paolo Romano, Francesco Quaglia, and Luis Rodrigues. GMU: Genuine Multiversion Update-Serializable Partial Data Replication. *IEEE Transactions on Parallel and Distributed Systems*, 27(10):2911–2925, October 2016. ISSN 1045-9219. doi: 10.1109/TPDS.2015.2510998. URL https://doi.org/10.1109/TPDS.2015.2510998.

[37] Dan R. K. Ports and Kevin Grittner. Serializable Snapshot Isolation in PostgreSQL. *Proc. VLDB Endow.*, 5(12):1850–1861, August 2012. ISSN 2150-8097. doi: 10.14778/2367502.2367523. URL https://doi.org/10.14778/2367502.2367523.

[38] Masoud Saeida Ardekani, Pierre Sutra, Marc Shapiro, and Nuno Preguiça. On the scalability of snapshot isolation. In Felix Wolf, Bernd Mohr, and Dieter an Mey, editors, *Euro-Par 2013 Parallel Processing*, pages 369–381, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-40047-6.

[39] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles - SOSP '11*, 2011.

[40] Werner Vogels. Eventually consistent. *ACM Queue*, 6(6):14–19, October 2008.