UPMC
SORBONNE UNIVERSITÉS

THÈSE DE DOCTORAT DE
l'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Masoud SAEIDA ARDEKANI

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Ensuring Consistency in Partially Replicated Data Stores

soutenue le 16 Septembre 2014

devant le jury composé de :

M. Marc SHAPIRO	Directeur de thèse
M. Pierre SUTRA	Encadrant de thèse
M. Willy ZWAENEPOEL	Rapporteur
M. Roberto BALDONI	Rapporteur
M. Douglas B. TERRY	Examinateur
Mme. Maria POTOP-BUTUCARU	Examinateur
M. Nuno PREGUIÇA	Examinateur

Abstract

Cloud-based applications, such as social networking or eCommerce, require to replicate data across several sites to provide responsiveness, availability, and disaster tolerance. Ensuring consistency over a large scale system with slow, and failure prone WANs has become of a paramount importance. This thesis studies this issue.

In the first part, we study consistency in a transactional systems, and focus on reconciling scalability with strong transactional guarantees. We identify four scalability properties as being critical for scalability: (i) only replicas updated by a transaction T make steps to execute T; (ii) a read-only transaction never waits for concurrent transactions and always commits; (iii) a transaction may read versions committed after it started; and (iv) two transactions synchronize with each other only if their writes conflict. We show that none of the strong consistency criteria ensure all four. We define a new scalable consistency criterion called Non-Monotonic Snapshot Isolation (NMSI), while is the first that is compatible with all four properties. We also present a practical implementation of NMSI, called Jessy, which we compare experimentally against a number of well-known criteria. Our last contribution in the first part is a framework for performing fair, and apples-to-apples comparison among different transactional protocols. Our insight is that a large family of distributed transactional protocols have a common structure, called Deferred Update Replication (DUR). Protocols of the DUR family differ only in behaviors of few generic functions. We present a generic DUR framework, called G-DUR, along with a library of finely-optimized plug-in implementations of the required behaviors. Our empirical study shows that: *(i)* G-DUR allows developers to implement various transactional protocols in less than few hundreds lines of code; *(ii)* It provides a fair, apples-to-apples comparison between transactional protocols; *(iii)* By replacing plugs-ins, developers can use G-DUR to understand bottlenecks in their protocols; *(iv)* This in turn enables the improvement of existing protocols; and *(v)* Given a protocol, G-DUR allows to evaluate the cost of ensuring various degrees of dependability.

In the second part of this thesis, we focus on ensuring consistency in non-transactional data stores. We introduce Tuba, a replicated key-value store that dynamically selects replicas in order to maximize the utility delivered to read operations according to a desired consistency defined by the application. In addition, unlike current systems, it automatically reconfigures its set of replicas while respecting application-defined constraints so that it adapts to changes in clients' locations or request rates. We implemented Tuba on top of Windows Azure Storage (WAS). While providing similar API, Tuba extends WAS with a broad set of consistency choices, consistency-based SLAs, and a geo-replication configuration service. Compared with a system that is statically configured, our evaluation shows that Tuba increases the reads that return strongly consistent data by 63% and improves average utility up to 18%.

To my beloved wife, Niloofar,

to my kind and encouraging parents, Fatemeh and Saeid,

and to my wonderful siblings, Maryam and Mohammad

ACKNOWLEDGEMENT

I am heartily grateful to my adviser, Marc Shapiro, for giving me the opportunity of working under his supervision. This thesis would not exist without his support, patience, and invaluable guidance over the last four years.

I would like to show my sincerest gratitude to Pierre Sutra. He helped me with every bit of my research, and contributed significantly to many algorithms and theorems in the first part of this thesis. Thank you Pierre for your excellent guidance, support, and caring.

I owe many thanks to Doug Terry, my internship adviser. I learned many practical aspects of distributed systems from him. Thank you Doug for trusting me. I had a chance to work closely with Nuno Preguiça during his visit of our team in 2011. Our discussions with him resulted in some impossibility results presented in this thesis. Thank you Nuno for the Nuno's counter-example.

I would also like to thank all my thesis juries, especially Willy Zwaenepoel and Roberto Baldoni, for accepting to be in my jury without hesitation, and for their precious time, and comments on this work.

I would also like to thank all my fellow graduate students in the Regal team, specially Marek Zawirski, Pierpaolo Cincilla, Corentin Mehat, Lisong Guo, Florian David, Maxime Lorrillere, and Tyler Crain. Thank you Marek for all helpful and inspiring conversations during coffee breaks. Thank you Pierpaolo, for your friendship, and all your helps in developing early version of Jessy, and most importantly, thank you for all translations.

I am very grateful to Marcos Aguilera, Mahesh Balakrishnan, Jiaqing Du, Sameh Elnikety, Ramakrishna Kotla, Gilles Muller, Vivien Quéma, and many other researchers who spent some of their time to help me with this research.

I would like to thank my parents, my sister and my brother for encouraging me all these years. Finally, I am most grateful to my wife, Niloofar, for all her patience and understanding.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Cloud applications are accessed from many distributed end-points. In order to improve responsiveness and availability, and to tolerate disasters, cloud storage systems replicate data across several sites (data centers) located in different geographical locations (called geo-replication).

Many authors argue that geo-replicated systems should provide only eventual consistency [1, 144], because of the CAP impossibility result (in the presence of network faults, either consistency or availability must be forfeited [58]), and because of the high latency of strong consistency protocols in wide-area networks. However, eventual consistency is confusing for developers, and is too weak for implementing some applications (e.g., banking systems).

Unfortunately, classical strong consistency criteria do not scale well to high load in the wide area. Therefore, several previous works aim at designing consistency criteria that both provide meaningful guarantees to the application, and scale well [3, 22, 57, 63, 68, 90, 106, 136]. However, the performance and scalability implications of these consistency criteria, and protocols ensuring them are still not well understood.

1.1 Contributions

Our contributions in this thesis are divided into two parts. In the first part, we focus on ensuring consistencies in transactional systems. We take a systematic approach to investigate scalability limitations of current transactional consistencies (i.e., isolation levels), and to reconcile scalability with strong transactional guarantees.

In the second part, we focus on ensuring consistency in non-transactional cloud storage systems. In particular, we investigate automatically reconfiguration of storage systems while

maintaining consistencies. In the remainder of this section, we review in more details our contributions in Part I, and Part II.

1.1.1 Part I

Catalog of Transactional Protocols Supporting Partial Replication Full replication (i.e., all processes store all objects) does not scale since all replicas must execute all updates. In addition, storing and managing large data sets at all processes require tremendous storage capacity at every process. Partial replication addresses these issues by replicating subset of data at each process. Therefore, different subset of data can be accessed, and modified concurrently.

Our first contribution is to compare the transactional protocols supporting partial replication. Our comparison is based on classical metrics (failure model, synchrony assumption, or consistency criterion), and scalability properties. To this end, we first identify the following four crucial scalability properties:

(i) Wait-Free Queries: a read-only (query) transaction never waits for concurrent transactions and always commits. In workloads with a high portion of read-only transactions, this property is crucial for the scalability of the system.

(ii) Forward Freshness: a transaction may read object versions that have been committed after the transaction started. This property decreases staleness of reads, and hence the abort rate.

(iii) Genuine Partial Replication (GPR): only replicas updated by a transaction T make steps to execute T. This property ensures that non-conflicting transactions do not interfere with each other, hence the intrinsic parallelism of a workload can fully be exploited.

(iv) Minimal Commit Synchronization: two transactions synchronize with each other only if their writes conflict. Therefore, synchronization is avoided unless absolutely necessary.

We then review various transactional protocols, and show that none of the surveyed protocols is able to ensure all four properties.

Scalability of Strong Consistency Criteria Our second contribution is to study scalability limitations of various consistency criteria in more details. We focus on Snapshot Isolation (SI) since it is a popular approach in both distributed databases, and software transactional memories. We first decompose SI into a set of necessary and sufficient properties. We then investigate scalability implications of every property to see if it can be ensured in a GPR system, and show that it is impossible to guarantee some of these properties under some reasonable progress assumptions. As corollaries, we also show that other well-known consistencies (e.g., Serializability) are also subjected to our impossibility results, and hence are not scalable.
These results are published in Europar'13 [118] and WTTM'11 [114].

Non-monotonic Snapshot Isolation To sidestep our impossibility results, we introduce a new consistency criterion called Non-monotonic Snapshot Isolation (NMSI). Under NMSI, a transaction must read a committed version of an object, and always take consistent snapshots. In addition, no two concurrent write-conflicting transactions can both commit. NMSI is the strongest consistency criterion that can ensure all the four aforementioned scalability properties. We also introduce a protocol ensuring NMSI, called Jessy. Jessy uses dependence vectors, a novel data type that enables efficient computation of consistent snapshots. Finally, we perform an empirical evaluation of the scalability of NMSI, along with a careful and fair comparison against a number of classical criteria, including SER, US, SI and PSI. Our experiments show that NMSI is close to RC (i.e, the weakest criterion) with disaster-prone configurations, and up to two times faster than Parallel Snapshot Isolation PSI.
These results are published in SRDS'13 [117] and HotCDP'12 [115].

Generic Deferred Update Replication Finding one's way in the jungle of consistency criteria and transactional protocols is not easy. Although literature is abundant, papers use different vocabulary, formalisms, and perspectives. Because they assume different environments, the implementations themselves are not comparable. It thus remains difficult to understand what are the important differences, and to make an objective, scientific comparison of their real-world behavior.

To address these challenges, we propose a new approach. Our insight is that many deferred update replication (DUR) protocols share a common structure, and differ only by specific instantiations of a few generic functions (e.g., [12, 100, 103, 104, 106, 117, 125, 127, 129–131, 136]). We express this insight as a common algorithmic structure, with well-identified realization points. This generic structure is instantiated into a specific protocol by selecting appropriate plug-ins from a library. By mixing-and-matching the appropriate plug-ins, it is relatively easy to obtain a high-performance implementation of a protocol.

(1) We tailor G-DUR to implement, and compare empirically six prominent transactional protocols [106, 117, 127, 129, 131, 136].

(2) We show how a developer can use G-DUR to finely understand the limitations of a protocol. We take a recently published protocol [106], and identify its bottlenecks by methodically replacing its plugs-ins by weaker ones.

(3) The previous approach also helps a developer to enhance existing protocols. We illustrate this point by presenting a variation of P-Store [127] that leverages workload locality to perform up to 70% faster than the original protocol.

(4) In our last set of experiments, we evaluate the cost of various degrees of dependability. To that end, we take a protocol ensuring serializability and we study the price of tolerating failures by varying the replication degree and the algorithm in use during commitment.
These results are published in Middleware'14 [119].

1.1.2 Part II

A Self-Configurable Cloud Storage System Automatically reconfiguring a cloud storage system can improve its overall service. Tuba is a replicated key-value store that, like some previous systems, allows applications to select their desired consistency and dynamically selects replicas that can maximize the utility delivered to read operations. Additionally, unlike current systems, it automatically reconfigures its set of replicas while respecting application-defined constraints so that it adapts to changes in clients' locations or request rates. Tuba is built on top of Windows Azure Storage (WAS) and provides a similar API. It extends WAS with broad consistency choices, consistency-based SLAs, and a geo-replication configuration service. Compared with a system that is statically configured, our evaluation shows that Tuba increases the reads that return strongly consistent data by 63% and improves average utility up to 18%.
These results are published in OSDI'14 [113].

1.2 Outline of the thesis

This thesis is organized as follows. We begin Part I by introducing our system model, notations, and reviewing some strong consistency criteria in Chapter 2. In Chapter 3, we identify the four crucial scalability properties for partial replicated systems, and survey some recent transactional systems. Chapter 4 studies the scalability limitations of strong consistency criteria. We introduce our new consistency criterion in Chapter 5. Chapter 6 introduces the generic deferred update replication protocol as our last contribution for Part I. In Part II (Chapter 7), we introduce our self-configurable cloud storage system. We conclude this thesis in Chapter 8, and introduce future research directions.

Part I: Ensuring Consistency in Transactional Data Stores

Chapter 2

Background

Contents

In this chapter, we first describe our model, and define the notations used throughout this thesis. Then we review some well-known *strong* consistency criteria, which have received attention in both academia and industry. We also define and explain some anomalies, which are observable in each criterion. Finally, we compare consistency criteria based on these anomalies.

2.1 Model

In this section, we define elements in our model such as objects, transactions and histories. Our model is very similar to the models of Adya [3] and Bernstein et al. [24]. We also define formally the concept of a replication system used throughout this thesis.

2.1.1 Objects & transactions

Let *Objects* be a set of objects, and $\mathcal{T}$ be a set of transaction identifiers. Given an object x and a transaction identifier i, x_i denotes *version* i of x written by T_i. A *transaction* $T_{i\in\mathcal{T}}$ is a finite sequence of read and write operations followed by a *terminating* operation, commit (c_i) or abort (a_i). Throughout this thesis, a read-only transaction is specified with an alphabetic subscript, and an update transaction with a numeric subscript. We use $w_i(x_i)$ to denote transaction T_i writing version i of object x, and $r_i(x_j)$ to mean that T_i reads version j of object x. We assume that the initial transaction T_0 installs the initial versions of all objects. Without loss of generality, we assume that in a given transaction, every write is preceded by a read to the same object, and every object is read or written at most once. [1] We note $ws(T_i)$ the writeset of T_i, i.e., the set of objects written by transaction T_i. Similarly, $rs(T_i)$ denotes the readset of transaction T_i. Two transactions *conflict* when they access the same object and at least one of them modifies it (i.e., $rs(T_i) \cap ws(T_j) \neq \varnothing$); they *write-conflict* when they both write to the same object (i.e., $ws(T_i) \cap ws(T_j) \neq \varnothing$).

2.1.2 Histories

A *complete history* h is a partially ordered set of operations such that:

1. For any operation o_i appearing in h, transaction T_i terminates in h,
2. For any two operations o_i and o'_i appearing in h, if o_i precedes o'_i in T_i, then $o_i <_h o'_i$,
3. For any $r_i(x_j)$ in h, there exists a write operation $w_j(x_j)$ such that $w_j(x_j) <_h r_i(x_j)$, and
4. Any two write operations over the same objects are ordered by $<_h$.

A *history* is a prefix of a complete history. For some history h, order $<_h$ is the *real-time order* induced by h. Transaction T_i is *pending* in history h if T_i does not commit, nor abort in h. We note $\ll_h$ the version order induced by h between different versions of an object, i.e., for every object x,

[1] These restrictions ease the exposition of our results but do not change their validity.

Notation	Meaning
$x, y, \ldots$	Object
$T_a, T_b, \ldots$	Read-only transaction
$T_1, T_2, \ldots$	Update transaction
x_i	Version of x written by T_i
$w_i(x_i)$	Transaction T_i writes x
$r_i(x_j)$	Transaction T_i reads x, written by T_j
$rs(T_i)$ / $ws(T_i)$	Read-set / write-set of transaction T_i
h	Transactional history (partially ordered)
$o_i <_h o'_j$	Operation o_i appears before o'_j in h
$x_i \ll_h x_j$	Version order $w_i(x_i) <_h w_j(x_j)$ holds
$T_i \parallel T_j$	T_i and T_j are concurrent

Table 2.1: Useful Notations

and any two transactions T_i and T_j, $x_i \ll_h x_j = w_i(x_i) <_h w_j(x_j)$. Following Bernstein et al. [24], we depict a history as a graph. We illustrate this below with history h in which transaction T_a reads the initial versions of objects x and y, while transaction T_1 (respectively T_2) updates x (resp. y).

$$h = r_a(x_0) \longrightarrow r_1(x_0).w_1(x_1).c_1$$
$$\searrow\ r_a(y_0).c_a \longrightarrow r_2(y_0).w_2(y_2).c_2$$

We say transaction T_i precedes transaction T_j if the commit of T_i is before the first operation of T_j (denoted as s_j) in h: $c_i <_h s_j$. Two transactions T_i and T_j are called *concurrent* (denoted as $T_i \parallel T_j$) if neither T_i precedes T_j nor T_j precedes T_i. For example, in the above history, T_a precedes T_2, whereas T_a and T_1 are concurrent.

When order $<_h$ is total, we shall write the history as a permutation of operations, e.g., $h = r_1(x_0).r_2(y_0).w_2(y_2).c_1.c_2$. Table 2.1 summarizes our notation.

2.1.3 Distributed System

We consider a message-passing distributed system of n processes $\Pi = \{p_1, \ldots, p_n\}$. We shall define our synchrony assumptions later. Following Fischer et al. [54], an execution is a sequence of steps made by one or more processes. During an execution, processes may fail by crashing. A process that does not crash is said *correct*; otherwise it is *faulty*. We note $\mathfrak{F}$ the refinement mapping [2] from executions to histories, i.e., if ρ is an execution of the system, then $\mathfrak{F}(\rho)$ is the history produced by ρ. A history h is *acceptable* if there exists an execution ρ such that $h = \mathfrak{F}(\rho)$. We consider that given two sequences of steps U and V, if U precedes V in some execution ρ, then the operations implemented by U precedes (in the sense of $<_h$) the operations implemented by V

in the history $\mathfrak{F}(\rho)$. [2]

2.1.3.1 Failure Models

In this thesis, we consider the following failure models:

1. *Crash-stop:* in this model, a correct process never crashes. A faulty process crashes, and does not perform any computation, nor send any message to other processes.

2. *Crash-recovery:* in this model, a correct process can crash and later recover a finite number of times. A faulty process either crashes and never recovers, or crashes and recovers an infinite number of times. We note that if processes have access to a stable storage, then this model is equivalent to (asynchronous) failure-free model.

2.1.3.2 Synchrony Assumptions

Depending on communication delays and speeds of processes, a distributed system is classified as follows:

1. *Asynchronous:* no assumption is made on either communication delays or relative speeds of processes.

2. *Synchronous:* there is a known upper bound on communication delays and processing time.

3. *Partially Synchronous:* after some unknown time, there is an upper bound on communication delays and processing time.

2.1.3.3 Failure Detectors

In order to encapsulate the synchrony assumptions of some system, every process in the system is augmented with an oracle, called a failure detector [33]. The oracle maintains a list of processes that it suspected to have crashed. Failure detectors are categorized into different classes depending on accuracy and completeness. In this thesis, we are particularly interested in the following two classes:

1. *Perfect Failure Detector ($\mathcal{P}$).* A failure detector is $\mathcal{P}$ if (1) eventually, all correct processes suspect all crashed processes (strong completeness); and (2) no process is suspected before it crashes (strong accuracy).

2. *Eventually Strong Failure Detector ($\diamond\mathcal{S}$).* A failure detector is $\diamond\mathcal{S}$ if (1) eventually, all correct processes suspect all crashed processes (strong completeness); and (2) there is a time after which no correct process is suspected by another correct process (eventual strong accuracy).

2.1.4 Replication

A data store $\mathcal{D}$ is a finite set of tuples (x, v, i) where x is an object (data item), v a value, and $i \in \mathcal{T}$ a version. Initially every object x has version x_0. Given a data store $\mathcal{D}$, a system is called

[2] Notice that since steps to implement operations may interleave, $<_h$ is not necessarily a total order.

full replication when each process in Π holds the whole data store $\mathcal{D}$. It is called *partial* if some process in Π holds a proper subset of $\mathcal{D}$.

For an object x, *replicas*(x) (or group g_x) denotes the set of processes, or *replicas*, that hold a copy of x; we assume that *replicas*(x) $\neq \emptyset$. By extension, for some set of objects X, *replicas*(X) (or group g_X) denotes union of the replicas of $x \in X$. Given a transaction T_i, *replicas*(T_i) equals *replicas*($rs(T_i) \cup ws(T_i)$). Groups are called *partitions*, if they store disjoint subsets of *Objects*

The coordinator of T_i, denoted *coord*(T_i), is in charge of executing T_i on behalf of some client (not modeled). We assume that transactions are interactive: the coordinator does not know in advance the readset or the writeset of T_i. To model this, we consider that every prefix of a transaction (followed by a terminating operation) is a transaction with the same id.

In a partially-replicated system, a transaction is said *local*, if there exists some replica holding all the objects read or written by the transaction; otherwise, it is said *global*.

Wiesmann et al. [146] classify replication techniques based on the following three parameters:

1. Server Interaction: degree of communication among replicas for executing a transaction. This parameter is either constant or linear. In the former case, a constant number of messages is used to synchronize a transaction, and in the latter one, each operation in a given transaction is propagated.

2. Server Architecture: it states where transactions are submitted and executed. The following two cases exist for this parameter: primary copy, and update everywhere.

3. Transaction Termination: it expresses how transactions are terminated. Termination can either contain voting, or can be done without any voting.

In this thesis, we focus on update everywhere replication with constant interaction. In addition, we solely focus on the optimistic execution technique (or Deferred Update Replication): a transaction is executed optimistically at its coordinator, and it is only during its commit time that replicas synchronize with each other, and decide to commit or abort it. In the next section, we review various techniques to synchronize replicas, and commit a transaction.

2.1.5 Transactional Commitment

A commitment protocol allows replicas to reach an agreement on whether to commit or abort a given transaction. Commitment protocols can be classified into the following classes: (i) Atomic Commitment (Section 2.1.5.1), (ii) Total Ordering (Section 2.1.5.2), and (iii) Partial Ordering (Section 2.1.5.3). In the remainder of this section, we review some widely used transactional commitment techniques based on the above classification. In the next chapter, we study how different transactional protocols mix and use variations of the these approaches to atomically commit transactions. We also generalize these approaches into a common algorithmic structure in Chapter 6.

2.1.5.1 Atomic Commitment Approach

In this approach, given a transaction, every replica votes to commit or abort it. It commits only if all replicas vote to commit it.

Two-phase Commit (2PC) The classical two-phase commit [59] is a widely-used atomic commitment protocol. A transaction's coordinator first tries to prepare replicas: it sends a *prepare* message for a transaction T_i to the replicas that are interested in T_i. Upon receiving the *prepare* message, replicas reply either *yes* (if they can commit T_i), or *no* (if they cannot commit T_i). If the coordinator receives *yes* from all participating replicas, it sends the *commit* message to all replicas; otherwise, if it receives a *no*, it sends *abort*; The main issue with 2PC is that it blocks if the coordinator of the transaction fails.

Paxos Commit To side step the blocking problem of 2PC, Gray and Lamport [60] introduce Paxos Commit. It runs a Paxos consensus to commit or abort a transaction. Instead of relying on one coordinator (a single point of failure), Paxos Commit uses a fault-tolerant decentralized consensus algorithm. Hence, it uses $2f+1$ coordinators, and is able to progress as long as $f+1$ are correct. Paxos Commit has the same message delay as 2PC in failure-free cases, but it uses more messages to reach an agreement.

2.1.5.2 Total Ordering Approach

In Total Ordering approach, an agreement protocol is used to ensure that all transactions are delivered in the same order, and hence, a commit/abort algorithm will execute identically at all replicas. In other words, every replica will reach the same commit/abort decision deterministically, independently, and locally. This approach has been used in many Deferred Update Replication protocols with full replications [78, 87, 101, 102].

Atomic Broadcast Atomic Broadcast (or Total-Order Broadcast) delivers all messages, in the same order, to all processes. A process invokes the primitive AB-Cast to send a message m to all processes, and AB-Deliver atomically delivers the message m to every process. Since messages are delivered in the same order, and hence processed in the same order, it is an easy approach for ensuring consistency (like serializability) among replicas.

A *uniform* Atomic Broadcast guarantees the following properties [35]:

(i) *Validity*: if a correct process atomic-broadcasts message m, then it eventually atomic-delivers m.

(ii) *Uniform Integrity*: every process atomic-delivers message m at most once, and only if m was atomic-broadcast previously.

(iii) *Uniform Agreement*: if some process atomic-delivers m, then eventually all correct processes atomic-deliver m.

(iv) *Uniform Total Order*: if some process p atomic-delivers m_1 before m_2, then every process q also atomic-delivers m_1 before m_2.

2.1.5.3 Partial Ordering Approach.

Ordering transactions globally is expensive, specially in large scale systems because all replicas should be involved in the execution of some agreement protocol that orders messages globally. In addition, protocols based on atomic broadcast cannot fully leverage the benefits of partial replication since all processes are still involved in processing each transaction. Hence, an emerging alternative is to order transactions partially for partial replication protocols [117, 127, 129, 130]. In this approach, transactions are totally ordered within some replica groups (or some partitions), but they are not totally ordered globally.

Atomic Multicast Atomic Multicast sends a message m to γ groups of processes using AM-Cast primitive, and atomic-delivers m to all processes in γ using AM-Deliver primitive.

Uniform Atomic Multicast ensures the following properties [122]:

(i) *Validity*: if a correct process atomic-multicasts m, then eventually all correct processes in γ atomic-deliver m.

(ii) *Uniform Integrity*: a process p atomic-delivers message m at most once, and only if m was previously atomic-multicast.

(iii) *Uniform Agreement*: if a process atomic in γ atomic-delivers message m, then eventually all correct processes in γ atomic-delivers m.

(iv) *Uniform Prefix order*: for any two process p and q that are the recipients of m_1 and m_2, if p atomic-delivers m_1 and q atomic-delivers m_2, then either p atomic-delivers m_2 before m_1 or q atomic-delivers m_1 before m_2.

(v) *Uniform Acyclic Order*: noting $m_1 < m_2$ if and only if m_1 is delivered before m_2 by a process, the relation $<$ is acyclic.

Note that the uniform prefix order disallows holes in the sequence of messages delivered by processes. For instance, consider that three messages m_1, m_2 and m_3 are atomic-multicast to a group g containing process p and q. Process p delivers all messages as follows: $m_1 < m_2 < m_3$. Without this property, a faulty process q would be allowed to deliver only m_1 and m_3 in the same order, and to skip the delivery of m_2. Uniform prefix order precludes this.

In addition, uniform acyclic order ensures a global partial order of messages without cycles. For instance, consider messages m_1 sent to groups g_x and g_z, message m_2 that is atomic-multicast to groups g_x and g_y, and finally m_3 that targets groups g_z and g_y. Without the acyclic ordering properties, groups would be able to deliver messages in the following orders: 1. group g_x: $m_1 < m_2$; 2. group g_y: $m_2 < m_3$; 3. group g_z: $m_3 < m_1$.

All the above reviewed approaches provide the same functionality: they are used to commit or abort a transaction atomically at all replicas. Hence, we use the term *atomic commitment* to refer to any protocol providing this functionality.

2.2 Strong Consistency Criteria

In this section, we first define a consistency criterion, and a strong consistency criterion concepts. Then we review some of the main strong consistency criteria, along with the anomalies that they expose to clients. Finally, we compare the criteria reviewed in terms of their undesirable effects.

A (transactional) consistency criterion is a *safety* property that constraints how transactions interleave. Roughly speaking, a safety property ensures that nothing bad happens [79]. In the database community, this safety property is named isolation level (I in ACID) because they ensure different levels of non-interference between transactions [3], and the term consistency is used to specify the application-level consistency (C in ACID). In the first part of this thesis, we use the term consistency criterion as Adya [3] to refer to isolation levels (such as serializability).

Definition 2.1 (Consistency Criterion)**.** A consistency criterion $\mathcal{C}$ is a prefix-closed subset of $\mathcal{H}$, where $\mathcal{H}$ is the set of all histories.

Depending on how transactions are interleaved in each consistency (i.e., deviate from sequential execution), some undesirable observations called *anomalies* are observable in each consistency. Some of these anomalies are tractable: they can be precluded easily.

Tractable Anomalies In what follows, we review three tractable anomalies, and briefly explain different ways to preclude them.

Dirty Write happens when a modification to an object is overwritten with the changes made by another unfinished transaction.

Definition 2.2 (Dirty Write)**.** Dirty Write happens in a history h when transaction T_i modifies an object x, and before committing or aborting, another transaction T_j also modifies x. If either T_i or T_j aborts, then it is not clear what should be the final value of object x.

Berenson et al. [22] consider that any consistency criterion should prevent the dirty write anomaly. In practice, it is easy to prevent this anomaly either by using locks, or by using a multi-version scheme and making changes visible once the transaction commits.

The second tractable anomaly is *Dirty Read* anomaly. For instance, consider the following history: $h_{dr} = r_1(x_0).w_1(x_1).r_a(x_1).c_a.a_1$. In this history, transaction T_a reads an uncommitted value from T_1, and commits; transaction T_1 later aborts, and does not install x_1.

Definition 2.3 (Dirty Read)**.** Dirty read happens in a history h when a transaction T_j reads a value of the object modified by an uncommitted transaction T_i, and T_i may later abort or change again the value of the object.

Like Dirty Write, this anomaly can be prevented by making a transaction's changes visible only once it commits.

The third tractable anomaly is called *Non-Repeatable Read*. For instance, consider the following history $h_{nrr} = r_a(x_0).w_2(x_2).c_2.r_a(x_2)$. In this history, transaction T_a first reads version x_0, and later reads x_2 which is installed by a committed transaction T_2.

Definition 2.4 (Non-repeatable Read)**.** A Non-Repeatable Read happens in a history h when a transaction T_i reading an object twice (before and after a committed transaction) obtains different values.

It is also easy to disallow this anomaly. For instance, caching the read values, and returning the cached values upon subsequent reads to the same objects.

Strong Consistency Criteria Our focus in this thesis is solely on *strong* consistency criteria. Roughly speaking, a strong criterion ensures that replica divergence never occurs.

Definition 2.5 (Strong Consistency Criterion)**.** A consistency criterion $\mathcal{C}$ is strong if it disallows two concurrent committed transactions to modify the same object.

Thus, any strong criterion precludes the anomaly called *Lost Update*: concurrent transactions modifying the same object such that the update of one transaction is lost. For instance, in the following history $h_{lu} = r_1(x_0).r_2(x_0).w_1(x_1).c_1.w_2(x_2).c_2$, the outcome of transaction T_1 (i.e., x_1) is lost.

Definition 2.6 (Lost Update)**.** Lost update anomaly occurs in a history h when a transaction T_i reads an object x, and subsequently another transaction T_j update the object x. Transaction T_i then modifies x based on its earlier read, and commits.

We note that the above definition of strong consistency is different from the one given in the CAP theorem [58]. However, both of them imply similar results. The CAP theorem considers atomic objects, and it uses the term strong consistency to refer to linearizability [70]. We, on the other hand, use the term strong consistency to refer to any criterion $\mathcal{C}$ which precludes the Lost Update anomaly. Bailis et al. [17] demonstrate that preventing Lost Update inherently requires forgoing high availability guarantees. A system ensures high availability if a client eventually receives a response from a correct replica, even in the presence of network partitions.

We now review, and formally define some well-known consistency criteria for distributed transactional systems. We note that all the criteria reviewed in the following precludes all the three tractable anomalies along with the Lost Update.

2.2.1 Strict Serializability (SSER)

Strict serializability [98] (SSER) is the strongest consistency criterion: it ensures that every transaction appear to execute at a single point in time between its first operation, and its

commit point. Hence, clients observe no anomaly under SSER. More precisely, a transactional system ensures SSER when the following two conditions hold: (i) every concurrent execution of committed transactions is equivalent to some serial execution of the same transactions, and (ii) Non-overlapping transactions in real-time are serialized in the order in which they are executed in real-time.

The combination of the above two properties provide SSER with a unique feature called *external consistency* [86]. Under external consistency, consistency is guaranteed even when communication happens outside the system boundaries. For instance, Alice can deposit $100 into Bob's account. Upon committing the transaction, she can call Bob over phone, and give him the confirmation that $100 is transferred to his account. At this point, Bob will definitely observe $100 in his account. As we shall see later, external consistency is not easily provided by weaker consistency criteria.

2.2.2 Full Serializability (SER)

Full Serializability (SER), or Serializability for short, is the classical consistency criterion implemented by transactional systems. A transactional system ensures full serializability when every concurrent execution of committed transactions is equivalent to some serial execution of the same transactions. Unlike SSER, SER does not necessarily respect the real-time ordering between transactions. Hence, executions under SER may observe an anomaly called real-time violation. For example, consider the following history $h_{rtv} = r_1(x_0).w_1(x_1).c_1.r_2(x_0).c_2$, and assume that c_1 executes before $r_2(x_0)$ in real-time. This history is SER (but not SSER) since $T_2.T_1$ is a correct serial execution of the two transactions.

Definition 2.7 (Real-time Violation Anomaly)**.** Real-time violation happens when a transaction T_i does not observe the effect of some transaction that committed in real-time before T_i's first operation.

As long as clients do not communicate outside the system boundary, and as long as the transactional system does not behaves trivially (i.e., always returning the initial version of objects), this anomaly is not considered harmful. For instance, consider the above history h_{rtv}. If Alice issues transaction T_1, and Bob issues transaction T_2, then history h_{rtv} can be expected from a non-trivial transactional system. However, if both T_1 and T_2 are issued by the same client, then the above history is not acceptable. For instance, in a social-network like application, Alice may update her profile status in T_1, but unable to see her status in T_2. By enforcing certain session guarantees for clients namely *read-my-write* and *monotonic-read* [139], we can disallow this this anomaly for each client.

2.2.3 Update Serializability (US)

Update serializability, introduced by Garcia-Molina and Wiederhold [57], guarantees that update transactions are serialized with respect to other update transactions. Read-only transactions only need to take consistent snapshots, and always commit unilaterally. Thus, unlike SER, US does not guarantee a serial order among read-only transactions. This leads to the anomaly called long fork [136] or non-monotonic snapshot for read-only transactions. Consider the following history h_{nms}. Read-only transaction T_a takes snapshot $\{x_0, y_2\}$, and T_b takes snapshot $\{x_1, y_0\}$, where $x_0 \ll x_1$ and $y_0 \ll y_2$.

$$h_{nms} = r_a(x_0) \longrightarrow r_1(x_0).w_1(x_1).c_1 \longrightarrow r_b(x_1).c_b$$
$$r_b(y_0) \longrightarrow r_2(y_0).w_2(y_2).c_2 \longrightarrow r_a(y_2).c_a$$

Alice executing T_a observes the effect of transaction T_2 but not T_1. Bob, is executing T_b, observes the effect of T_1 but not T_2. By using session guarantees, one can easily disallow this anomaly in for a client.

The rationale behind US is that read-only transactions dominate updates in many workloads, and that ensuring consistent snapshots is usually enough for many read-only transactions.

US is also extended to aborted transactions called *Extended Update Serializability* (EUS) by Hansdah and Patnaik [68]. Therefore, aborted transactions should also observe consistent snapshots. In Section 2.2.7, we explain in detail the rationale of reading a consistent snapshot for aborted transactions.

2.2.4 Snapshot Isolation (SI)

Snapshot isolation (SI) introduced by Berenson et al. [22] is one of the most famous consistency criteria supported by many DBMSes (e.g., Oracle [110], PostgreSQL [108], and Microsoft SQL Server [95]). In Snapshot isolation (SI), a transaction reads its own consistent snapshot, and aborts only if it write-conflicts with a previously-committed concurrent transaction. A read-only transactions never conflicts with any other transaction and always commits. Moreover, unlike SSER, SER or US, update transactions abort only if they write-conflict with a concurrent transaction. This allows some transactions to commit while they would have aborted if they were serialized.

Unlike previous consistencies, SI also considers a start point for a transaction. A start point of a transaction T_i (denoted as s_i) is the first operation invoked by T_i. The start point of a transaction is then used to order the transaction with all other transactions.

A history h ensures SI iff it satisfies the following rules:

D1 (Read Rule)

$\forall r_i(x_{j\neq i}), w_{k\neq j}(x_k), c_k \in h:$

$c_j \in h$ $(D1.1)$

$\wedge\ c_j <_h s_i$ $(D1.2)$

$\wedge\ (c_k <_h c_j \vee s_i <_h c_k)$ $(D1.3)$

D2 (Write Rule)

$\forall c_i, c_j \in h:$

$ws(T_i) \cap ws(T_j) \neq \varnothing$

$\Rightarrow \left(c_i <_h s_j \vee c_j <_h s_i\right)$

Roughly speaking, the read rule ensures that a transaction T_i observes the effect of all update transactions committed before its start point. The write rule ensures that no two concurrent conflicting transactions both commit. Adya [3] also noted that: *transaction T_i's start point need not be chosen after the most recent commit when T_i started, but can be selected to be some (convenient) earlier point*.

Unlike SER, SI only disallows two concurrent write-conflicting transactions to commit. Thus, a new anomaly called *Write Skew* is observable in SI. A write skew anomaly can violate an invariant if two none write-conflicting transactions concurrently execute in a system. For example, consider two linked bank accounts for Alice and Bob. The bank allows one of Alice's or Bob's account balance to be negative, as long as the total balance is never negative. Now suppose each Alice and Bob have $100 in their account. If they both spend $150 concurrently with their credit cards, and even if each transaction correctly checks the invariant, the following history can occur. $h_{wsk} = r_1(alice = 100).r_1(bob = 100).r_2(alice = 100).r_2(bob = 100).w_1(alice = -50).w_2(bob = -50).c_1.c_2$ In the beginning, two concurrent transactions first take a consistent snapshot from their accounts. They then check whether the sum of the account's balances minus $150 is still greater than zero, and if so, the transactions proceed to subtract $150 from each account and commit. This results in -$50 balance for each account, violating the invariant.

Definition 2.8 (Write Skew Anomaly)**.** A write skew anomaly takes place when two concurrent transactions T_1 and T_2 concurrently commit such that:

1. $rs(T_i) \cap rs(T_j) \neq \varnothing$
2. $ws(T_i) \cap ws(T_j) = \varnothing$

Several proposals [50, 51, 75] ensure serializable execution under SI. The main idea behind all these work is analysis of the whole workload at design-time, and modifying it to avoid the anomaly. For example, Fekete et al. [51] propose the following techniques to prevent potential inconsistencies arisen from the write skew anomaly: (i) *Materializing the conflict:* add a conflicting object such that both transactions update it, or (ii) *Promotion:* force one of the transactions to update a read-only object. Both techniques make $ws(T_i) \cap ws(T_j)$ non-empty; hence the anomaly does not happen anymore.

Cahill et al. [29] propose to modify the concurrency control algorithm in order to automatically detect and prevent the anomaly. To this end, conflict patterns that must occur for an SI execution to be non-serializable are detected at run-time, and a sufficient number of involved transactions is aborted.

2.2.4.1 Generalized Snapshot Isolation (GSI)

Elnikety et al. [49] generalize SI, under the name *Generalized Snapshot Isolation* (GSI). GSI distinguishes between a transaction's start point (i.e., the time of the transaction's first operation), and an abstract snapshot point. The read rules are then defined with respect to the snapshot points and not start points. Therefore, the start point in Adya's definition plays the same role as snapshot point in Elnikety's one, and they are considered equivalent [43].

2.2.5 Parallel Snapshot Isolation (PSI)

With the emergence of new Internet based applications, like social networks, the need for highly scalable yet geographically replicated transactional systems has increased substantially over the past few years. Sovran et al. [136] note that SI requires snapshots to form a total order, which does not scale well. To address this problem, Sovran et al. [136] define Parallel Snapshot Isolation (PSI), a new consistency criterion suitable for geo-replicated systems. It allows the relative commit order of non-conflicting transactions to vary between replicas. Thus, PSI does not totally order transactions' snapshots. Under PSI, both read-only and update transactions might observe non-monotonic snapshots. Moreover, since PSI is weaker than SI, it also has the write-skew anomaly.

Based on the formal model given in [136], a history h guarantees PSI if the following properties hold:

(i) *Site Snapshot Read.* All read operations should read the most recent committed version at the transaction's site (i.e., a data center that the transaction starts its execution), and before the time the transaction started.

(ii) *No write-write Conflict.* No two concurrent write-conflicting transactions both commit.

(iii) *Commit Causality Across Sites.* Causality between transactions is maintained at all sites.

2.2.6 Causal Serializability (CSER)

Causal serializability (CSER) [109] is very similar to PSI. It ensures the following properties:

- Transactions from the same client are sequential.
- Transactions respect read-from dependency.
- Transactions updating the same objects are observed in the same order by all replicas of those objects.

However, unlike PSI, CSER is not a multi-version consistency criterion, and does not require reading committed values. Hence some anomalies defined previously (like non-monotonic snapshots) are not observable with CSER.

2.2.7 Consistency Criteria for Software Transactional Memory

To make concurrent programming easier, and to provide developers with a convenient abstraction, Shavit and Touitou [133] introduce a new method called Software Transactional Memory (STM). With STM, threads of an application synchronize with each other via transactions. Although some STMs ensure conventional consistency criteria (like SER or SI), Guerraoui and Kapalka [63] argue that these criteria are not suitable for STMs because these consistencies are applied only to committing transactions. Thus, they do not preclude a transaction from reading an inconsistent state as long as it later aborts. Behaviors of the transactions that abort are not considered harmful in managed environments (such as databases) since transactions can run in isolation through sandboxing techniques. However, executing a transaction in unmanaged environments is harmful, and can cause a whole application to crash [63, 137]. For instance, an STM transaction that reads an inconsistent state would cause a divide-by-zero exception.

For the sake of completeness, in the remainder of this section, we briefly review consistency criteria introduced for STMs.

Opacity: Guerraoui and Kapalka [63] introduce Opacity as the strongest consistency criterion a transactional system can provide. It strengthens SSER by considering also aborting transactions. A transaction system ensures Opacity if: (i) every concurrent execution of committed transactions along with read-prefixes of aborted transactions[3] is equivalent to some serial execution, and (ii) this serial execution respects real-time orderings among transactions.

Virtual World Consistency (VWC): Opacity requires that all read prefixes of aborted transactions observe exactly the same causal path. Imbs and Raynal [73] argue that it is more conservative than needed. They introduce a weaker consistency criterion called VWC. VWC requires that: (i) all committed transactions be SSER; (ii) every read-prefix of an aborted transaction reads values that are consistent with respect to its causal past.

2.2.8 Anomaly Comparison

Table 2.2 summarizes all the consistency criteria that we reviewed in this section, and compare them in terms of observed anomalies.

2.3 Liveness and Progress

In the previous section, we introduced various consistency criteria as safety properties for transactional systems. In this section, we specify liveness and progress properties. A *liveness* property states that eventually something useful occurs [79]. Such a property is a *progress* property when it states that, at all point in time, some action is eventually executed.

[3] a read-prefix of an aborted transaction contains all read operations of the aborted transaction until it aborts.

Anomalies	Consistency Criteria					
	SSER Opacity VWC	SER	US	SI GSI	PSI	CSER
Dirty Write	x	x	x	x	x	x
Dirty Read	x	x	x	x	x	x
Non-repeatable Reads	x	x	x	x	x	x
Lost Update	x	x	x	x	x	x
Real-time Violation	x	-	-	-	-	-
Non-monotonic Snapshot among R-O transactions	x	x	-	x	-	x
Non-monotonic Snapshot among R-O and UP transactions	x	x	x	x	-	x
Write Skew	x	x	x	-	-	-

Table 2.2: Anomaly Comparison of Strong Consistency Criteria (x:*disallowed*)

	SSER SER US	SI PSI CSER
T_i $\mathcal{C}-conflict$ T_j	$rs(T_i) \cap ws(T_j) \neq \varnothing \ \vee \ ws(T_i) \cap rs(T_j) \neq \varnothing$	$ws(T_i) \cap ws(T_j) \neq \varnothing$

Table 2.3: Conflict Table of Consistency Criteria

In the context of transactional systems, the liveness property of Schiper et al. [127], called termination, requires that if the coordinator of a transaction is correct, the system should eventually terminates the transaction, either by committing or aborting it.

Definition 2.9 (Termination)**.** For every submitted transaction T_i, if $coord(T_i)$ is correct, then T_i eventually terminates.

Unfortunately, this property is not enough, and fails to rule out a system that always abort submitted transactions. The progress properties that we introduce next constraint when the system may abort a transaction due to interleaving [65].

Our progress properties rely on the classical notion of conflict, called $\mathcal{C}-conflict$, that captures the interleaving allowed by a criterion. Table 2.3 depicts the definition of $\mathcal{C}-conflict$ for the reviewed consistency criteria. Two transactions conflict in serializability-based consistencies (i.e., SSER, SER, and US) when they exhibit read-write conflict. On the opposite, in snapshot isolation based consistencies (SI, PSI, and CSER), two transaction conflict solely when their writesets intersect.

The first progress property we consider is *wait-freedom* [65]. This property captures the maximal progress a transactional system may offer, i.e., that a transaction always commits.

Definition 2.10 (Wait-freedom)**.** A transactional system is wait-free when in every execution ρ, for every transaction T_i in $h = \mathfrak{F}(\rho)$, if T_i is not pending in h then T_i commits in h.

Wait-freedom is ensured for transactions in systems implementing weak consistency models (e.g., Causal+ [90, 91, 151]). However in general, it is not possible to ensure this progress property for update transactions, since two concurrent updates transactions cannot both commit under strong consistency. On the other hand, as we shall see shortly, wait-freedom for read-only transactions is usually attainable (e.g., in SI, PSI, or US).

The second progress property of interest we consider is *obstruction-freedom*. This property is weaker than wait-freedom, and it states that a transaction must commit when it does not encounter a conflicting transactions during its execution.

Definition 2.11 (Obstruction-freedom)**.** A transactional system implementing a consistency $\mathcal{C}$ is obstruction-free when in every execution ρ, for every transaction T_i in $h = \mathfrak{F}(\rho)$, if T_i aborts in h then T_i $\mathcal{C}$-conflicts with some concurrent transaction in h.

For example, consider a transactional system where all conflicting transactions with a transaction T_i have terminated. If at this time, a transaction T_i in h executes, it must commit. As another example, obstruction-freedom guarantees that if a transaction T_i is the only transaction being executed in the system, it must commit.

Since obstruction-freedom requires a non-conflicting transaction T_i to always commit in all possible execution, obstruction-freedom might be considered too strong. Hence, we introduce *non-triviality* as our weakest progress property. In a common sense, non-triviality necessitates that there should be at least one way to execute and commit a transaction if all other transactions have terminated. This progress property is ensured in many transactional systems (e.g., [48, 104, 106, 127]).

Definition 2.12 (Non-triviality)**.** A transactional system implementing a consistency $\mathcal{C}$ provides non-triviality if in every execution ρ such that $h = \mathfrak{F}(\rho)$ is quiescent (i.e., no transaction is pending), for every transaction $T_i \notin h$, there exists an extension ρ' of ρ such that transaction T_i commits in history $\mathfrak{F}(\rho')$.

	Wait-freedom	Obstruction-freedom	Non-triviality
Read-only Transaction	Wait-free Queries (WFQ)		
Update Transaction		Obstruction-free Updates (OFU)	Non-trivial Updates (NTU)

Table 2.4: Progress Properties

Table 2.4 summarizes the progress properties that we use throughout this thesis. For read-only transactions, we are interested in wait-free queries. We focus on obstruction-free, and non-trivial updates as progress properties of update transactions.

CHAPTER 3

CATALOG OF TRANSACTIONAL PROTOCOLS SUPPORTING PARTIAL REPLICATION

Contents

In this chapter, we first identify the four properties of interest (Section 3.1) that increase scalability of a transactional system. We then review some well-known partially-replicated transactional protocols, and compare them in terms of: (i) conventional assumptions (such as failure model, or synchrony assumptions) (ii) commitment protocol, and (iii) the four identified scalability properties.

3.1 Scalability Properties

Ensuring a consistency criterion through a commitment protocol is costly, especially in large scale or geo-replicated settings. To better understand, and minimize this cost, in this section, we identify the following essential scalability properties: *(i)* a read-only transaction never waits for concurrent transactions and always commits; *(ii)* only replicas updated by a transaction T make steps to execute T; *(iii)* a transaction may read object versions committed after it started; and *(iv)* two transactions synchronize with each other only if their writes conflict. These properties amplify scalability by increasing parallelism, and slashing abort rate.

3.1.1 Wait-Free Queries (WFQ)

Since most workloads exhibit a high proportion of read-only transactions, WFQ is a crucial scalability property as it ensures that a read-only transaction is not slowed down by synchronization, and always commits.

3.1.2 Genuine Partial Replication (GPR)

Replication improves both locality and availability. In full replication, every replica must perform all updates, therefore it does not scale. Partial replication addresses this problem, by replicating only a subset of the data at each replica. The idea is that if transactions would communicate only over the minimal number of replicas, synchronization and computation overhead would be reduced. However, in the general case, the overlap of transactions cannot be predicted; therefore, many partial replication protocols in fact perform system-wide global consensus [15, 131] or communication [136]. This approach negates the potential advantages of partial replication.

To address this issue, Schiper et al. [127] introduce *genuine* partial replication: a transaction communicates only with the replicas that store some object read or written in the transaction. GPR is also called the *minimality* property by Fritzke and Ingels [55]. We call the set of replicas storing objects read or written by transaction T_i, the replicas *concerned* by T_i.

With GPR, non-conflicting transactions do not interfere with each other, and the intrinsic parallelism of a workload can be exploited, ensuring that throughput scales linearly with the number of nodes.

Definition 3.1 (Genuine Partial Replication)**.** A transactional system ensures Genuine Partial Replication if, for any transaction T_i, only processes that replicate objects concerned by T_i make steps to execute T_i.

In addition to scalability advantages, GPR also improves *availability* of a transactional system. Failures of a replica that is not concerned by a transaction does not halt the execution of that transaction. This improves the availability of the system. For instance, consider a banking application, in which accounts of American branches are replicated in the US, and accounts of European branches are stored in the EU. Under GPR, transactions among EU accounts can execute and finish even when EU and US sites are partitioned.

We note that GPR is equivalent to the concept of strictly disjoint access parallelism in an STM. An STM is strictly disjoint-access-parallel when non-conflicting transactions never contend on the same base object, that is on any object in use at the implementation level [64].

In order to have a more fine-grained comparison among transactional protocols that are not GPR, we define two additional properties of interest in what follows:

Definition 3.2 (Committing Replicas)**.** The set of replicas that is involved in commitment of transaction T_i is called the committing replicas of T_i.

Some transactional protocols (such as Walter [136]) perform asynchronous propagation to all replicas once a transaction commits. To capture these attributes, we define affected replicas as the set of all replicas that are affected (either by sending or receiving a message) due to execution of a transaction. Note that the set of committing replicas of T_i is always subset of the set of replicas affected by T_i.

Definition 3.3 (Affected Replicas)**.** The set of replicas that send or receive a message due commitment of transaction T_i is called replicas affected by T_i.

3.1.3 Minimal Commitment Synchronization

Because of its direct cost, and the convoy effects and oscillations that it causes [27, 127], synchronization should be avoided, unless absolutely necessary. Hence, our third scalability property focuses on minimizing the synchronization between transactions (to alleviate their cost) while keeping the consensus power of transactions.

Definition 3.4 (Minimal Commitment Synchronization)**.** A consistency criterion $\mathcal{C}$ supports Minimal Commitment Synchronization if during commitment, transaction T_i waits for transaction T_j only if T_i and T_j write-conflict.

3.1.4 Forward Freshness

Some criteria (such as SI, and PSI) freeze the set of object versions that a transaction may read as soon as the transaction starts; a version that is committed afterwards cannot be used. These criteria requires *base freshness*.

However, it is desirable that a consistency criterion provides *forward freshness*: a transaction be allowed to read the most recent versions of objects available. Otherwise, abort probability increases (because the transaction reads a stale version), and even read-only transactions observe stale data.

Definition 3.5 (Base Freshness)**.** A consistency criterion $\mathcal{C}$ ensures Base Freshness if for any history $h \in \mathcal{C}$, and for any two read operations $r_i(x_j)$ and $r_i(y_l)$ in h, $r_i(x_j) \not<_h c_l$.

Definition 3.6 (Forward Freshness)**.** A consistency criterion $\mathcal{C}$ supports Forward Freshness if there exists a history $h \in \mathcal{C}$ such that $r_i(x_j) <_h c_l$ for some read operations $r_i(x_j)$ and $r_i(y_l)$.

In case of a global transaction that touches several geographical sites, forward freshness is essential to prevent stale reads, and to decrease the abort rate.

In this section, we identified four properties that can boost scalability by increasing parallelism, and decreasing the abort rate. In Chapter 4, we focus more on these properties, and investigate whether strong consistency criteria can ensure these properties or not.

3.2 Review of Transactional Protocols Supporting Partial Replication

In this section, we compare and review transactional protocols ensuring strong consistencies. To have a more coherent review, our focus in this section is solely on partial replication protocols providing general purpose transactions. Hence, we do not consider protocols with the following assumptions:

1. Transactions or workloads are known in advance [39, 41, 42, 71, 74, 83, 152].
2. Certain treatments are required for executing transactions that are not known in advance [141].
3. There is at least one replica holding all data accessed by a transaction [31, 40, 124, 125, 135].

Table 3.1 compare the protocols based on their assumptions, and Table 3.2 compare their scalability properties.

Protocol	Cons.	Opt. for Lan/Wan	Multi-master	Failure Model	Synchrony Assumption	Commitment	Failure Detector
Spanner [38]	SSER	WAN	yes	Crash-recovery	Synchronous	Intra-group Paxos [81] + Inter-group 2PC	-
P-Store [127]	SER	WAN	yes	Crash-stop	Asynchronous	AM-Cast [123] + Inter-group Voting	$\diamond\mathcal{S}$
S-DUR [129]	SER	LAN	yes	Crash-stop	Partially Synchronous	Intra-group AB-Cast [93] + Inter-group Voting	-
Sciascia13 [130]	SER	WAN	yes	Crash-stop	Partially Synchronous	Intra-group AB-Cast [81] + Inter-group Voting	-
SCORe [104]	SER	LAN	yes	Crash-stop	Asynchronous	Inter-process 2PC	-
Scalaris [121]	SER	WAN	yes	Crash-stop	Partially Synchronous	Inter-process Enhanced Paxos Commit [121]	-
Fritzke [55]	SER	LAN	yes	Crash-stop	Asynchronous	AM-Cast [56] + Inter-group Voting	$\diamond\mathcal{S}$
GMU [106]	EUS	LAN	yes	Crash-stop	Asynchronous	Inter-process 2PC	-
Serrano07 [131]	SI	LAN	yes	Crash-stop	?	AB-Cast [?]	-
Serrano08 [132]	SI	WAN	no	Crash-recovery	?	Intra-site AB-Cast [112]	?
Clock-SI [48]	SI	WAN	no	Crash-stop	Asynchronous	Inter-partition 2PC	-
SIPRe [15]	GSI	LAN	yes	Crash-recovery	Partially Synchronous	AB-Cast [112]	-
Walter [136]	PSI	WAN	yes	Crash-recovery	Asynchronous	Inter-site 2PC	$\mathcal{P}$ (conf. service)

Site: data center;
Inter-group protocol p: consider a group as a correct entity and perform protocol p among them;
Intra-site protocol p: perform protocol p among processes in one site;
Inter-process protocol p: perform protocol p among processes;
?: unknown

Table 3.1: Comparison of Assumptions of Partial Replication Protocols

	Scalability				Committing Replicas of T_i		Affected Replicas of T_i	
Protocol	WFQ	GPR	FF	MCS	Read-only Transaction	Update Transaction	Read-only Transaction	Update Transaction
Spanner	yes	yes	no	no	$\varnothing$	$replicas(rs(T_i) \cup ws(T_i))$	$\varnothing$	$replicas(rs(T_i) \cup ws(T_i))$
P-Store	no	yes	yes	no	$replicas(rs(T_i) \cup ws(T_i))$	$replicas(rs(T_i) \cup ws(T_i))$	$replicas(rs(T_i) \cup ws(T_i))$	$replicas(rs(T_i) \cup ws(T_i))$
S-DUR	yes	yes	no	no	$\varnothing$	$replicas(rs(T_i) \cup ws(T_i))$	$\varnothing$	$replicas(rs(T_i) \cup ws(T_i))$
Sciascia13	no	yes	yes	no	$replicas(rs(T_i) \cup ws(T_i))$	$replicas(rs(T_i) \cup ws(T_i))$	$replicas(rs(T_i) \cup ws(T_i))$	$replicas(rs(T_i) \cup ws(T_i))$
SCORe	yes	yes	no	no	$\varnothing$	$replicas(rs(T_i) \cup ws(T_i))$	$\varnothing$	$replicas(rs(T_i) \cup ws(T_i))$
Scalaris	no	yes	yes	no	$replicas(rs(T_i) \cup ws(T_i))$	$replicas(rs(T_i) \cup ws(T_i))$	$replicas(rs(T_i) \cup ws(T_i))$	$replicas(rs(T_i) \cup ws(T_i))$
Fritzke	no	yes	yes	no	$replicas(rs(T_i) \cup ws(T_i))$	$replicas(rs(T_i) \cup ws(T_i))$	$replicas(rs(T_i) \cup ws(T_i))$	$replicas(rs(T_i) \cup ws(T_i))$
GMU	yes	yes	yes	no	$\varnothing$	$replicas(rs(T_i) \cup ws(T_i))$	$\varnothing$	$replicas(rs(T_i) \cup ws(T_i))$
Serrano07	yes	no	no	yes	$\varnothing$	Π	$\varnothing$	Π
Serrano08	yes	no	no	yes	$\varnothing$	$replicas(certifierSite)$	$\varnothing$	$replicas(certifierSite) \cup replicas(rs(T_i) \cup ws(T_i))$
Clock-SI	yes	yes	no	yes	$\varnothing$	$replicas(ws(T_i))$	$\varnothing$	$replicas(ws(T_i))$
SIPRe	yes	no	no	yes	$\varnothing$	Π	$\varnothing$	Π
Walter	yes	no	no	yes	$\varnothing$	$replicas(primarySites(ws(T_i)))$	$\varnothing$	Π

WFQ: Wait-free Queries
GPR: Genuine Partial Replication
FF: Forward Freshness
MCS: Minimal Commitment Synchronization

Table 3.2: Comparison of Properties of Partial Replication Protocols

3.2.1 SSER

Spanner: Spanner is a large scale and temporal multi-version database ensuring SSER. Due to its large synchronization overhead, SSER was not used at large-scale until its recent implementation by Google in Spanner [38]. It assigns a globally meaningful timestamp to a transaction. Timestamps specify serialization order of transactions globally. Spanner uses a TrueTime API which provides clocks with bounded uncertainty. To this end, it uses low-latency, low-jitter network connections. Spanner's correctness relies on an upper bound on the actual message delay: if TrueTime does not guarantee bounded clock drift, external consistency is violated. Read-only transactions always commit in Spanner, hence it ensures WFQ. Spanner guarantees OFU as the progress property of update transactions. In order to commit an update transaction, and guaranteeing SSER, Spanner relies on two-phase locking, and 2PC. Update transactions commit using a 2PC among the replica groups storing objects, and Paxos inside each replica group. Thus, it also guarantees GPR.

3.2.2 SER

P-Store: P-Store [127] is a partially replicated system based on deferred update replication: a transaction is executed optimistically at the transaction's coordinator, and upon executing a commit request, it is certified and commits at concerned replicas. It is a GPR algorithm that ensures SER by leveraging genuine atomic multicast. Read operations are performed optimistically, and write operations are cached locally at the transaction's coordinator. A replica simply returns the latest committed version of an object upon receiving a read request for that object. During the termination phase, the readset and writeset of transaction T_i is multicast atomically to all the replicas holding some object in $ws(T_i) \cup rs(T_i)$. A transaction commits if all the versions read by the transaction are the latest committed versions. Since only the replica holding objects read or written by a transaction are involved in its certification and termination, P-Store is a GPR protocol. However, P-Store does not provide WFQ since a read-only transaction should also be certified, and can abort. This leads to relatively poor performance for read-only transactions compared to other similar protocols. Moreover, unlike Spanner, P-Store guarantees NTU as the progress property of update transactions.

S-DUR: S-DUR [129] is also based on deferred update replication. It ensures that every transaction reads a consistent snapshot during its optimistic execution, and commits locally without any synchronization, hence ensures WFQ. To commit an update transaction T_i, S-DUR atomically multicasts T_i (i.e., its readset, writeset, and its snapshot) to every replica group replicating an object in $ws(T_i) \cup rs(T_i)$. Thus, it is a GPR protocol. However, unlike P-Store, S-DUR only requires to ensure total ordering inside each replica group. In other words, ordering is not required across replica groups. For example, consider two update transactions T_1 and T_2, and assume that they both modify objects x and y. In S-DUR, all the processes holding objects x can deliver T_1 before

T_2, and all the processes holding object y can deliver T_2 before T_1. By only ordering transactions inside each replica group, S-DUR increases the scalability of the atomic multicast primitive, but this comes at a price: the certification test is more involved, and it needs to ensure that two global transactions T_i and T_j can be serialized in any order with respect to each other. Like P-Store, S-DUR also ensures NTU.

Sciascia13: Although S-DUR provides good performance in LAN environments, its performance is not acceptable in geo-replicated environment because a local transactions that is delivered after some global transaction must be delayed until the global transaction commits. This increases the latency of local transactions by up to 10 times. Sciascia and Pedone [130] propose a GPR protocol that is similar to S-DUR. It addresses the above issue by reordering a local transaction, and executing it before a global transaction even if it is delivered after a global transaction. Unlike S-DUR, both read-only and update transactions are certified in this protocol, hence it does not ensure WFQ.

SCORe: Peluso et al. [104] describe a GPR protocol under SER ensuring WFQ and NTU. In SCORe, when a transaction starts (i.e., upon its first read), the system assigns a scalar timestamp as the transaction snapshot. This timestamp is the maximum of last timestamps given to an update transaction at the transaction's coordinator, and the node replicating the object. A subsequent read operations must read a version committed before the transaction snapshot. This approach, disallows SCORe from ensuring Forward Freshness. An update transaction commits using a combination of 2PC and Skeen's total order multicast [26].

Scalaris: Scalaris [121] is a GPR protocol on top of a DHT. Operations are performed on the majority of replicas, and an Enhanced Paxos commit ensures both atomicity and SER. Like previous protocols, Scalaris also employs an optimistic execution approach. A read request requires a majority read quorum on the replicas to obtain the latest committed version. A write request must be preceded by a read request in the same transaction. Hence, a write request also needs to first read from majority of replicas. Both read-only and update transactions commit with Paxos Commit algorithm. Like P-Store, Scalaris does not read consistent snapshots, and does not ensure WFQ. However, unlike P-Store, it guarantees OFU.

Fritzke: Fritzke and Ingels [55] propose a GPR protocol based on atomic multicast that ensures SER. Unlike all previous protocols, a read operations for object x is atomic multicast to a group replicating object x, and delivered by all processes. Write operations are stored locally until the commit time. Like P-Store, at commit time, the transaction is atomic multicast to all groups replicating an object read or written by the transaction. To commit a transaction, the protocol uses a voting phase similar to the non-blocking two phase commit protocol. Like Scalaris, this protocol does not provide WFQ, but guarantees OFU.

3.2.3 US

GMU: The GMU transactional system of Peluso et al. [106] guarantees EUS, and ensures both GPR and WFQ. GMU relies on a particular vector clock to ensure consistent snapshots for read operations. Read-only transactions commit locally, and GMU commits update transactions with 2PC; all replicas holding an object read or written by the transaction participate in the 2PC protocol. Notice that the certification test of GMU is much more simpler than the one employed in protocols ensuring SER. GMU also ensures NTU as the progress property of update transactions.

3.2.4 SI

Serrano07: The protocol of Serrano et al. [131] offers non-genuine partial replication under SI. Read-only transactions commit locally and update transactions are atomic broadcast to all replicas. Upon delivering of an update transaction, each replica performs a certification test and decides locally to commit or abort the transaction. Therefore, and unlike the typical approach employed by partially replicated transactional systems, Serrano does not perform a distributed voting phase. Bypassing this phase comes at the cost of maintaining at each replica the last version number of each object.

In order to take a consistent snapshot, each replica creates several dummy transactions upon committing an update transaction. These dummy transactions represents different snapshots. Hence, once a new transaction starts at a replica, a dummy transaction with a particular sequence number is associated with it. The transaction uses this dummy transaction for taking consistent snapshots when contacting different sites in the system. This protocol guarantees NTU as the progress property of update transactions.

Serrano08: Serrano et al. [132] propose a partially replicated system ensuring SI for Internet-based services. All read-only transactions commit locally at their originating sites. However, unlike Serrano2007, there is only one site (called certifier) that is responsible for certifying update transactions. Therefore, this protocol does not ensure GPR. In order to guarantee the same outcome at all processes in the certifier site, all transactions are delivered in the same order in all processes of the certifier site using atomic broadcast primitives. Moreover, all processes at the certifier site needs to store keys, and last versions of all modified objects in the system to be able to certify all update transactions.

Like the previous protocol, this protocol also relies on dummy transactions for taking consistent snapshot in a partially replicated system.

The authors assume that the atomic broadcast is offered by a group communication systems, and cite [112] as the reference. Hence, the exact assumptions for synchrony and failure detector cannot be inferred.

SIPRe: Armendáriz-Iñigo et al. [15] propose a partially replicated transactional system called SIPRe ensuring GSI. At the start of a transaction T_i, SIPRe atomically broadcasts T_i to all processes. This message is used for defining a consistent snapshot for T_i. Like other SI protocols, read-only transaction commit locally without any synchronization. An update transaction is atomic broadcast to all replicas, and certify in all of them. SIPRe guarantees OFU as the progress property of update transactions. Because both committing replicas of a transaction T_i and the replicas affected by T_i are Π, SIPRe does not ensure GPR. Like [131], SIPRe also requires all replicas (whether they are concerned by the transaction or not) to store write-set of all transactions along with their commit timestamp.

Clock-SI: Du et al. [48] introduce a fully decentralized implementation of SI based on loosely synchronized clocks. Unlike previous protocols, it solely partitions data into a set of servers, and does not support replication. The synchronized clocks are used to assign snapshot and commit timestamps to transactions. To side step the problems of clocks skew, Clock-SI delays transactions until assigned timestamps to transactions become available. Like previous protocols, read-only transactions commit locally without any additional certification. Local update transactions commit at the updated partition without any global synchronization. To commit a global update transaction, a 2PC is performed among the partitions holding modified objects. Clock-SI ensures NTU as the progress property of update transactions.

3.2.5 PSI

Walter: Walter is the transactional system proposed by Sovran et al. [136] to implement PSI. This system relies on a single master replication schema per object and 2PC. Thus, each object has a primary site, called preferred site. Like SI, PSI also requires that an operation has to read the most recent versions at the time the transaction starts. Hence, when a transaction starts, the coordinator assigns a vector timestamps to the transaction. A read-only transaction commits without any synchronization, thus Walter ensures WFQ. Walter employs either a *fast* or a *slow* commit protocol to commit an update transaction. If an update transaction only modifies the objects replicated at one preferred site, it uses the fast commit. In the fast commit protocol, a transaction commits if the objects modified by it have not been modified, and are not locked. If an update transaction modify the objects replicating at different preferred sites, it uses 2PC to commit the transaction. Once a transaction is committed, Walter must propagate the transaction in the background to all the replicas in the system before it becomes visible. Hence, the replicas affected by a transaction T_i is Π, and Walter is not a GPR protocol.

CHAPTER 4

SCALABILITY OF STRONG CONSISTENCY CRITERIA

Contents

In this chapter, we study the scalability cost of different consistency criteria with respect to Genuine Partial Replication (GPR) and Wait-free Queries (WFQ). As we shall see in Chapter 5, these two properties play a crucial role in scalability of a system.

We start this chapter by focusing on SI because it is supported by many DBMSs (e.g., Microsoft SQL Server, Oracle, or PostgreSQL), and also due to the fact that SI is a popular consistency criterion in software transactional memories [25, 45, 89, 111]. Moreover, by design, SI ensures WFQ. Therefore, we investigate whether we can ensure SI and GPR together or not.

We first prove in Section 4.1 that SI is equivalent to the conjunction of the following properties: *(i)* absence of cascading aborts, *(ii)* strictly consistent snapshots, i.e., a transaction observes a snapshot that coincides with some point in (linear) time, *(iii)* two concurrent write-conflicting update transactions never both commit, and *(iv)* snapshots observed by transactions are monotonically ordered. As we explained in Section 2.2.4, previous definitions of SI [3, 49] extend histories with abstract snapshot points, or start times (when it is selected to be some earlier point in time). Our decomposition shows that in fact, like serializability, SI can be defined on plain histories comprising read, write, commit and abort operations.

In Section 4.2, and based on this decomposition, we show that a system ensuring SI cannot guarantee both GPR and obstruction-free updates (OFU). In particular, we prove that an asynchronous GPR system guaranteeing OFU, even if it is failure-free, cannot compute monotonically ordered snapshots, nor strictly consistent ones.

Finally, in Section 4.3, we extend our results to other strongly consistent criteria (such as Opacitiy, SSER, SER, and PSI), and introduce some additional corollaries.

We recall that, in our model, objects accessed by a transaction are not known in advance (i.e., transactions are interactive), and every write is preceded by a read on the same object.

4.1 Decomposing SI

Before introducing four properties whose conjunction is equivalent to SI, we define SI based on a history comprising read, write, commit and abort operations.

Let us consider a function $\mathcal{S}$ that takes as input a history h, and returns an extended history h_s by adding a *snapshot point* to h for each transaction in h. Given a transaction T_i, the snapshot point of T_i in h_s, denoted s_i, precedes every operation of transaction T_i in h_s. A history h is in SI if, and only if, there exists a function $\mathcal{S}$ such that $h_s = \mathcal{S}(h)$ and h_s satisfies the following rules:

D1 (Read Rule)

$$\forall r_i(x_{j\neq i}), w_{k\neq j}(x_k), c_k \in h_s :$$
$$c_j \in h_s \qquad (D1.1)$$
$$\wedge\ c_j <_{h_s} s_i \qquad (D1.2)$$
$$\wedge\ (c_k <_{h_s} c_j \vee s_i <_{h_s} c_k) \qquad (D1.3)$$

D2 (Write Rule)

$$\forall c_i, c_j \in h_s :$$
$$ws(T_i) \cap ws(T_j) \neq \varnothing$$
$$\Rightarrow \left(c_i <_{h_s} s_j \vee c_j <_{h_s} s_i\right)$$

In the remainder of this section, we define four properties, and prove that the conjunction

of these properties is necessary and sufficient to attain SI. We later use these properties in Section 4.2 to derive our impossibility result.

4.1.1 Absence of Cascading Aborts (ACA)

Intuitively, a read-only transaction must abort if it observes the effects of an uncommitted transaction that later aborts. Thus, criteria that require wait-free queries (WFQ) (such as SI and PSI) need to make sure that a transaction never reads an uncommitted value. In case of SI, rules D1.1 and D1.2 ensure this requirement, called *absence of cascading aborts*. We formalize this property below:

Definition 4.1 (Absence of Cascading aborts)**.** History h exhibits no without cascading aborts, if for every read $r_i(x_j)$ in h, c_j precedes $r_i(x_j)$ in h. ACA denotes the set of histories that are without cascading aborts.

4.1.2 Consistent and Strictly Consistent Snapshots (SCONS)

Consistent and strictly consistent snapshots are defined by refining causality into a dependency relation as follows:

Definition 4.2 (Dependency)**.** Consider a history h and two transactions T_i and T_j. We note $T_i \rhd T_j$ when $r_i(x_j)$ is in h. Transaction T_i depends on transaction T_j when $T_i \rhd^* T_j$ holds. [1] Transaction T_i and T_j are *independent* if neither $T_i \rhd^* T_j$, nor $T_j \rhd^* T_i$ hold.

This means that a transaction T_i depends on a transaction T_j if T_i reads an object modified by T_j, or such a relation holds by transitive closure. To illustrate this definition, consider history $h_1 = r_1(x_0).w_1(x_1).c_1.r_a(x_1).c_a.r_b(y_0).c_b$. In h_1, transaction T_a depends on T_1. Moreover, observe that T_b does not depend on T_1 in h_1 although T_1 causally precedes T_b (i.e., c_1 precedes $r_b(y_0)$ in h_1).

We now define consistent snapshots according to the above dependency relation. A transaction has a consistent snapshot iff it observes the effects of all transactions it depends on [32]. For example, consider the history $h_2 = r_1(x_0).w_1(x_1).c_1.r_2(x_1).r_2(y_0).w_2(y_2).c_2.r_a(y_2).r_a(x_0).c_a$ In this history, transaction T_a does not have a consistent snapshot: T_a depends on T_2, and T_2 also depends on T_1, but T_a does not observe the effect of T_1 (i.e., x_1). Formally, consistent snapshots are defined as follows:

Definition 4.3 (Consistent snapshot)**.** A transaction T_i in a history h sees a consistent snapshot iff, for every object x, if (i) T_i reads version x_j, (ii) T_k writes version x_k, and (iii) T_i depends on T_k, then version x_k is followed by version x_j in the version order induced by h ($x_k \ll_h x_j$). We write $h \in$ CONS when all transactions in h have a consistent snapshot.

[1] We note $\mathcal{R}^*$ the transitive closure of some binary relation $\mathcal{R}$.

SI requires that a transaction observes the committed state of database at some time in the past. This requirement is stronger than consistent snapshot. For some transaction T_i in a history h, it implies that:

1. SCONSa[2]: a transaction T_i is not allowed to read the object committed after it starts (i.e., its first operation). To illustrate this, consider the following history that SCONSa forbids: $h_3 = r_1(x_0).w_1(x_1).c_1.r_a(x_1).r_2(y_0).w_2(y_2).c_2.r_a(y_2).c_a$ where all the operations are totally ordered. In this history, transaction T_1 (resp. T_2) modifies object x (resp. y), and transaction T_a reads objects x_1 and y_2. Because $r_a(x_1)$ precedes c_2 in h_3, y_2 could have not been observed when T_a took its snapshot. As a consequence, the snapshot of transaction T_a is not strictly consistent. Moreover, observe that history h_3 is SER because transactions can be serialized as $T_1.T_2.T_a$. However, it is violating the rule D1.2 of SI.

2. SCONSb: if transaction T_i observes the effects of transaction T_j, it must also observe the effects of all transactions that precede T_j in history h. For instance, consider the following history that SCONSb forbids: $h_4 = r_1(x_0).w_1(x_1).c_1.r_2(y_0).w_2(y_2).c_2.r_a(x_0).r_a(y_2).c_a$ where all operations are totally ordered. Like the previous example, transactions T_1 and T_2 modify objects x and y, and transaction T_a wants to read objects x and y. Since c_1 precedes c_2 in h_4 and transaction T_a observes the effect of T_2 (i.e., y_2), it should also observe the effect of T_1 (i.e., x_1). Note that like history h_3, history h_4 is also serializable: $T_2.T_a.T_1$. However, it is not in SI because: (i) if we consider the snapshot of T_a before the commit of T_1 or between the commit of T_1, and T_2, then it violates the rule D1.2; and (ii) if we consider the snapshot of T_a after the commit of T_2, then it violates the rule D1.3.

A history is called strictly consistent if both SCONSa and SCONSb hold. Formally:

Definition 4.4 (Strictly consistent snapshot)**.** Snapshots in history h are strictly consistent, when for any committed transactions T_i, T_j, $T_{k\neq j}$ and T_l, the following two properties hold:

- $\forall r_i(x_j), r_i(y_l) \in h : r_i(x_j) \not<_h c_l$ (SCONSa)

- $\forall r_i(x_j), r_i(y_l), w_k(x_k) \in h : c_k <_h c_l \Rightarrow c_k <_h c_j$ (SCONSb)

We note SCONS the set of strictly consistent histories.

4.1.3 Snapshot Monotonicity (MON)

In addition, SI requires what we call *monotonic snapshots*, or partially ordered snapshots. For instance, although history h_5 below satisfies SCONS, this history does not belong to SI: since T_a reads $\{x_0, y_2\}$, and T_b reads $\{x_1, y_0\}$, there is no extended history that would guarantee the read rule of SI.

[2] SCONSa is equivalent to the base freshness. Because of historical reasons, we use the term SCONSa in this chapter.

$$h_5 = r_a(x_0) \longrightarrow r_1(x_0).w_1(x_1).c_1 \longrightarrow r_b(x_1).c_b$$
$$r_b(y_0) \longrightarrow r_2(y_0).w_2(y_2).c_2 \longrightarrow r_a(y_2).c_a$$

While SI requires monotonic snapshots, the underlying reason is intricate enough that some previous works [25, for instance] do not ensure this property, while claiming to be SI. Below, we introduce an ordering relation between snapshots to formalize snapshot monotonicity.

Definition 4.5 (Snapshot precedence)**.** Consider a history h and two distinct transactions T_i and T_j. The snapshot read by T_i precedes the snapshot read by T_j in history h, written $T_i \rightarrow T_j$, when $r_i(x_k)$ and $r_j(y_l)$ belong to h and either (i) $r_i(x_k) <_h c_l$ holds, or (ii) transaction T_l writes x and $c_k <_h c_l$ holds.

For more illustration, consider histories $h_6 = r_1(x_0).w_1(x_1).c_1.r_2(y_0).w_2(y_2).r_a(x_1).c_2.r_b(y_2).c_a.c_b$ and $h_7 = r_1(x_0).w_1(x_1).c_1.r_a(x_1).c_a.r_2(x_1).r_2(y_0).w_2(x_2).w_2(y_2).c_2.r_b(y_2).c_b$. In history h_6, $T_a \rightarrow T_b$ holds because $r_a(x_1)$ precedes c_2 and T_b reads y_2. In h_7, c_1 precedes c_2 and both T_1 and T_2 modify object x. Thus, $T_a \rightarrow T_b$ also holds. We define snapshot monotonicity using snapshot precedence as follows:

Definition 4.6 (Snapshot monotonicity)**.** Given some history h, if the relation $\rightarrow^*$ induced by h is a partial order, the snapshots in h are *monotonic*. We note MON the set of histories that satisfy this property.

According to this definition, both $T_a \rightarrow T_b$ and $T_b \rightarrow T_a$ hold in history h_7. Thus, history h_7 does not belong to MON.

As we saw in Chapter 2, Non-monotonic snapshots are also observed under US, and PSI.

4.1.4 Write-Conflict Freedom

Rule D2 of SI forbids two concurrent write-conflicting transactions from both committing. Since we assume that every write is preceded by a read on the same object, every update transaction depends on a previous update transaction (or on the initial transaction T_0). Therefore, under SI, concurrent conflicting transactions must be independent:

Definition 4.7 (Write-Conflict Freedom (WCF))**.** A history h is write-conflict free if two independent transactions never write to the same object. We denote by WCF the histories that satisfy this property.

4.1.5 The Decomposition

Theorem 4.1 below proves that the conjunction of the above four properties is necessary and sufficient to attain SI. In other words, a history h is in SI iff (1) every transaction in h sees a

committed state, (2) every transaction in h observes a strictly consistent snapshot, (3) snapshots are monotonic, and (4) h is write-conflict free. A detailed proof appears in Appendix A.

Theorem 4.1. $SI = ACA \cap SCONS \cap MON \cap WCF$

To the best of our knowledge, this is the first result showing that SI can be split into simpler properties. Theorem 4.1 also establishes that SI is definable on plain histories that consist of read, write, commit, and abort operations. This has two interesting consequences: (i) a transactional system does not have to explicitly implement snapshots to support SI, and (ii) one can compare SI to other consistency criterion without relying on Adya's phenomena based characterization [3, 4].

4.2 The impossibility of SI with GPR

This section leverages our previous decomposition result to show that SI is inherently non-scalable. In more detail, we show that none of MON, SCONSa or SCONSb is obtainable in an asynchronous failure-free GPR system Π when updates are obstruction-free (OFU) and queries are wait-free (WFQ).

We first characterize in Lemmata 4.1 and 4.2 histories acceptable by Π.

Lemma 4.1 (Positive-freshness Acceptance)**.** *Consider an acceptable history h and a transaction T_i pending in h such that the next operation invoked by T_i is a read on some object x. Note x_j the latest committed version of x prior to the first operation of T_i in h. Let ρ be an execution satisfying $\mathfrak{F}(\rho) = h$. If $h.r_i(x_j)$ belongs to SI and there is no concurrent transaction write-conflicting with T_i, then there exists an execution ρ' extending ρ such that in history $\mathfrak{F}(\rho')$, transaction T_i reads at least (in the sense of $\ll_h$) version x_j of x.*

Proof. By contradiction. Assume that in every execution extending ρ, transaction T_i reads a version $x_k \ll_h x_j$. Let ρ' be such an extension in which (i) no other transaction than T_i makes steps, (ii) we extend T_i after its read upon x by a write on x, then (iii) $coord(T_i)$ tries committing T_i. Since T_i reads version x_k in $\mathfrak{F}(\rho')$, transaction T_i should abort. However in history $\mathfrak{F}(\rho')$ there is no concurrent write-conflicting transaction with T_i. Hence, this execution contradicts the assumption that updates are obstruction-free. ■

Lemma 4.2 (Genuine Acceptance)**.** *Let $h = \mathfrak{F}(\rho)$ be an acceptable history by a GPR system Π such that a transaction T_i is pending in h. Note X the set of objects accessed by T_i in h. Only processes in $replicas(X)$ make steps to execute T_i in ρ.*

Proof. (By contradiction.) Consider that a process $p \notin replicas(X)$ makes steps to execute T_i in ρ. Since the prefix of a transaction is a transaction with the same id, we can consider an extension ρ' of ρ such that T_i does not execute any additional operation in ρ' and $coord(T_i)$ is correct in ρ'. The progress requirements satisfied by Π imply that T_i terminates in ρ'. However, process $p \notin replicas(X)$ makes steps to execute T_i in ρ'. A contradiction to the fact that Π is GPR. ■

We now state that monotonic snapshots are not constructable by Π. Our proof holds because objects accessed by a transaction are not known in advance.

Theorem 4.2. *No asynchronous failure-free GPR system implements MON*

Proof. (By contradiction.) Let us consider (i) four objects x, y, z and u such that for any two objects in $\{x,y,z,u\}$, their replica sets do not intersect; (ii) four queries T_a, T_b, T_c and T_d accessing respectively $\{x,y\}$, $\{y,z\}$, $\{z,u\}$ and $\{u,x\}$; and (iii) four updates T_1, T_2, T_3 and T_4 modifying respectively x, y, z and u.

Obviously, history $r_b(y_0)$ is acceptable, and since updates are obstruction-free, there is a run such that $r_b(y_0).r_2(y_0).w_2(y_2).c_2$ is also acceptable. Applying that Lemma 4.1, we obtain that history $r_b(y_0).r_2(y_0).w_2(y_2).c_2.r_a(x_0).r_a(y_2)$ is acceptable. Since T_a is wait-free, $h = r_b(y_0).r_2(y_0).w_2(y_2).c_2.r_a(x_0).r_a(y_2).c_a$ is acceptable as well. Using a similar reasoning, $h' = r_d(u_0).r_4(u_0).w_4(u_4).c_4.r_c(z_0).r_c(u_4).c_c$ is also acceptable. We note ρ and ρ' respectively two sequences of steps such that $\mathfrak{F}(\rho) = h$ and $\mathfrak{F}(\rho') = h'$.

The system Π is GPR. Therefore, Lemma 4.2 tells us that only processes in *replicas*(x,y) make steps in ρ. Similarly, only processes in *replicas*(u,z) make steps in ρ'. By hypothesis, *replicas*(x,y) and *replicas*(u,z) are disjoint. Applying a classical indistinguishably argument [54, Lemma 1], both $\rho'.\rho$ and $\rho.\rho'$ are admissible by Π. Thus, histories $h'.h = \mathfrak{F}(\rho'.\rho)$ and $h.h' = \mathfrak{F}(\rho.\rho')$ are acceptable.

Since updates are obstruction-free, there is a run such that history $h'.h.r_3(z_0).w_3(z_3).c_3$ is acceptable. Note U the sequence of steps following $\rho'.\rho$ with $\mathfrak{F}(U) = r_3(z_0).w_3(z_3).c_3$. Observe that by Lemma 4.2 $\rho'.\rho.U$ is indistinguishable from $\rho'.U.\rho$. Then consider history $\mathfrak{F}(\rho'.U.\rho)$. In this history, T_b is pending and the latest version of object z is z_3, As a consequence, by applying Lemma 4.1, there exists an extension of $\rho'.U.\rho$ in which transaction T_b reads z_3. From the fact that queries are wait-free and since $\rho'.\rho.U$ is indistinguishable from $\rho'.U.\rho$, we obtain that history $h_1 = h'.h.r_3(z_0).w_3(z_3).c_3.r_b(z_3).c_b$ is acceptable.

We note U_1 the sequence of steps following $\rho'.\rho$ such that $\mathfrak{F}(U_1)$ equals $r_3(z_0).w_3(z_3).c_3.r_b(z_3).c_b$.

With a similar reasoning, history $h_2 = h'.h.r_1(x_0).w_1(x_1).c_1.r_d(x_1).c_d$ is acceptable. Note U_2 the sequence satisfying $\mathfrak{F}(U_2) = r_1(x_0).w_1(x_1).c_1.r_d(x_1).c_d$.

Executions $\rho'.\rho.U_1$ and $\rho'.\rho.U_2$ are both admissible. Because Π is GPR, only processes in *replicas*(y,z) (resp. *replicas*(x,u)) make steps in U_1 (resp. U_2). By hypothesis, these two replica sets are disjoint. Applying again an indistinguishably argument, $\rho'.\rho.U_1.U_2$ is an execution of Π. Therefore, the history $\hat{h} = \mathfrak{F}(\rho'.\rho.U_1.U_2)$ is acceptable. In this history, relation $T_a \rightarrow T_b \rightarrow T_c \rightarrow T_d \rightarrow T_a$ holds. Thus, $\hat{h}$ does not belong to MON. Contradiction. ■

Our next theorem states that SCONSb is not feasable with GPR. Similarly to Attiya et al. [16], our proof builds an infinite execution in which a query T_a on two objects never terminates. We first define a finite execution during which we interleave, between any two consecutive steps to execute T_a, a transaction updating one of the objects read by T_a. We show that during such an

execution, transaction T_a does not terminate successfully. Then, we prove that asynchrony allows us to continuously extend such an execution, contradicting the fact that queries are wait-free.

Definition 4.8 (Flippable execution)**.** Consider two distinct objects x and y, a query T_a over both objects, and a set of updates $T_{j\in[\![1,m]\!]}$ accessing x if j is odd, and y otherwise. An execution $\rho = U_1V_2U_2 \dots V_mU_m$ is called flippable if:

- transaction T_a reads in history $h = \mathfrak{F}(\rho)$ at least version x_1 of x,
- for any j in $[\![1,m]\!]$, U_j is the execution of transaction T_j by processes Q_j,
- for any j in $[\![2,m]\!]$, V_j are steps to execute T_a by processes P_j, and
- both $(Q_j \cap P_j = \varnothing) \oplus (P_j \cap Q_{j+1} = \varnothing)$ and $Q_j \cap Q_{j+1} = \varnothing$ hold,

Lemma 4.3. *Let ρ be an execution admissible by Π. If ρ is flippable and histories accepted by Π satisfy SCONSb, query T_a does not terminate.*

Proof. Let h be the history $\mathfrak{F}(\rho)$. In history h transaction T_j precedes transaction T_{j+1}, it follows that h is of the form $h = w_1(x_1).c_1.*.w_2(y_2).c_2.* \dots$, where each symbol $*$ corresponds to either no operation, or to some read operation by T_a on object x or y.

Because ρ is flippable, transaction T_a reads at least version x_1 of object x in h. For some odd natural $j \geq 1$, let x_j denote the version of object x read by T_a. Similarly, for some even natural l, let y_l be the version of y read by T_a. Assume that $j < l$ holds. Therefore, h is of the form $h = \dots w_j(x_j) \dots w_l(y_l) \dots$.

Note k the value $l+1$, and consider the sequence of steps V_k made by P_k right after U_l to execute T_a. Applying the definition of a flippable execution, we know that (F1) $(Q_l \cap P_k = \varnothing) \oplus (P_k \cap Q_k = \varnothing)$, and (F2) $Q_l \cap Q_k = \varnothing$. Consider now the following cases:

(Case $Q_l \cap P_k = \varnothing$.) It follows that ρ is indistinguishable from the execution $\rho'' = \dots U_j \dots V_k U_l U_k \dots$. Then from fact F2, ρ is indistinguishable from execution $\rho' = \dots U_j \dots V_k U_k U_l \dots$.

(Case $P_k \cap Q_k = \varnothing$) With a similar reasoning, we obtain that ρ is indistinguishable from $\rho' = \dots U_j \dots U_k U_l V_k \dots$.

(Case $P_k \cap (Q_l \cup Q_k) = \varnothing$.) This case reduces to any of the two above cases.

Note h' the history $\mathfrak{F}(\rho')$. Observe that since ρ' is indistinguishable from ρ, history h' is acceptable. In history h', $c_k <_{h'} c_l$ holds. Moreover, $c_j <_{h'} c_k$ holds by the assumption $j < l$ and the fact that k equals $l+1$. Besides, operations $r_i(x_j)$, $r_i(y_l)$ and $w_k(x_k)$ all belong to h'. According to the definition of SCONSb, transaction T_a does not commit in h'. (The case $j > l$ follows a symmetrical reasoning to the case $l > j$ we considered previously.) ■

Theorem 4.3. *No asynchronous failure-free GPR system implements SCONSb.*

Proof. (By contradiction.) Consider two objects x and y such that *replicas*(x) and *replicas*(y) are disjoint. Assume a read-only transaction T_a that reads successively x then y. Below, we exhibit an execution admissible by Π during which transaction T_a never terminates. We build this execution as follows:

[Construction.] Consider some empty execution ρ. Repeat for all $i >= 1$: Let T_i be an update of x, if i is odd, and y otherwise. Start the execution of transaction T_i. Since no concurrent transaction is write-conflicting with T_i in ρ and updates are obstruction-free, there must exist an extension $\rho.U_i$ of ρ during which T_i commits. Assign to ρ the value of $\rho.U_i$. Execution ρ is flippable. Hence, Lemma 4.3 tells us that transaction T_a does not terminate in this execution. Consider the two following cases: (Case $i = 1$) Because of Lemma 4.1, there exists an extension ρ' of ρ in which transaction T_a reads at least version x_1 of object x. Notice that execution ρ' is of the form $U_1.V_2.s.\ldots$ where *(i)* all steps in V_2 are made by processes in *replicas*(x), and *(ii)* s is the first step such that $\mathfrak{F}(U_1.V_2.s.) = r_1(x_0).w_1(x_1).c_1.r_a(x_1)$. Assign $U_1.V_2$ to ρ . (Case $i > 2$) Consider any step V_{i+1} to terminate T_a and append it to ρ.

Execution ρ is admissible by Π. Hence $\mathfrak{F}(\rho)$ is acceptable. However, in this history transaction T_a does not terminate. This contradicts the fact that queries are wait-free. ■

Our last theorem shows that SCONSa cannot be maintained under GPR.

Theorem 4.4. *No asynchronous failure-free GPR system implements SCONSa.*

Proof. (By contradiction.) Consider two distinct objects x and y such that *replicas*(x) and *replicas*(y) are disjoint. Let T_1 be an update accessing y, and T_a be a query reading both objects.

Obviously, history $h = r_a(x_0)$ is acceptable. Note U_a a sequence of steps satisfying $U_a = \mathfrak{F}(r_a(x_0))$. Because Π supports obstruction-free updates, we know the existence of an extension $U_a.U_1$ of U_a such that $\mathfrak{F}(U_1) = r_1(y_0).w_1(y_1).c_1$. By Lemma 4.2, we observe that $U_a.U_1$ is indistinguishable from $U_1.U_a$. Then by Lemma 4.1, there must exist an extension $U_1.U_a.V_a$ of $U_1.U_a$ admissible by Π and such that $\mathfrak{F}(V_a) = r_a(y_1).c_a$. Finally, since $U_a.U_1$ is indistinguishable from $U_1.U_a$ and $U_1.U_a.V_a$ is admissible, $U_a.U_1.V_a$ is admissible too. The history $\mathfrak{F}(U_a.U_1.V_a)$ is not in SCONSa. Contradiction. ■

Each of the above three theorems independently shows that no asynchronous system, even if it is failure-free, can support both GPR and SI. In particular, even if the system is augmented with failure detectors [33], a common approach to model partial synchrony, SI cannot be implemented under GPR. This fact strongly hinders the usage of SI at large scale. In the following sections, we further discuss implications of this impossibility result. In Chapter 5, we introduce a novel consistency criterion to overcome it.

4.3 Discussion

In this section, we discuss the consequences of our impossibility results, with an emphasis on other consistency criteria than SI.

4.3.1 SSER and Opacity

Observe that Theorem 4.4 also holds if we consider read-write and write-write conflicts (as showed in Table 2.3) for $\mathcal{C}-$*conflict* in the definition of obstruction-free updates.

As a consequence, neither SSER, nor Opacity (which is stronger than SSER) is attainable under GPR. In the case of opacity, this answers negatively to a problem posed by Peluso et al. [105].

4.3.2 SER

Permissiveness A transactional system Π is *permissive* with respect to a consistency criterion $\mathcal{C}$ when every history $h \in \mathcal{C}$ is acceptable by Π. Permissiveness [67] measures the optimal amount of concurrency a system allows. If we consider again histories h_1 and h_2 in the proof of Theorem 4.2, we observe that both histories are serializable. Hence, every system permissive with respect to SER accepts both histories. By relying on the very same argument as the one we exhibit to close the proof of Theorem 4.2, we conclude that no transactional system is both GPR and permissive with respect to SER. For instance, P-Store [127], a GPR protocol that ensures SER, does not accept history $h_8 = r_1(x_0).w_1(x_1).c_1.r_2(x_0).r_2(y_0).w_2(y_2).c_2$.

Wait-free Queries. Because of WFQ progress property of SI, a query never forces an update transaction to abort. This key feature of SI greatly improves performance. Most recent transactional systems that support SER (e.g., [104, 129]) also offer such a progress property. In the case of SER, this property is a feature of the input acceptance of the protocol.

Our impossibility result also applies to SER as follows. We note that Lemma 4.1 proves positive-freshness acceptance for SI under standard assumptions (i.e., OFU and WFQ)). If we assume that Lemma 4.1 holds for conflicting transactions, then Theorem 4.2 applies to transactional systems ensuring SER. This implies a choice between WFQ, OFU, and GPR.

4.3.3 PSI

In Section 2.2.5, we reviewed PSI as a weaker consistency criterion than SI. PSI allows snapshots to be non-monotonic, but still requires them to ensure SCONSa. Our impossibility result establishes that, in order to scale by having a GPR system, a transactional system needs supporting both non-monotonic *and* non-strictly consistent snapshots. Thus, while being more scalable than SI, PSI yet cannot be implemented in a GPR system that also ensures OFU.

4.3.4 Circumventing The Impossibility Result

We can sidestep the impossibility result in a transactional system while still ensuring the desired scalability properties (i.e., GPR and WFQ) as follows:

1. *Assuming a synchronous system.* The first way of circumventing our impossibility result is to assume a synchronous system: there exists an upper bound on message delays or speeds of processes. For example, Pacitti et al. [97] introduce a preventive replication protocol ensuring GPR and SER. They assume an upper bound on the time required to multicast a message, and an upper bound on the clock drift between any two processes in the system.

2. *Declaring readsets in advance.* When a transaction declares objects it accesses as it starts (before executing a read), a GPR system can install a strictly consistent and monotonic snapshot just at the start of the transaction. Therefore, such an assumption sidesteps our impossibility result. This is the approach employed for example in SIPRe [15]. Although the published SIPRe protocol makes use of atomic broadcast to install a snapshot, we obtain a GPR system that supports SI by replacing atomic broadcast by a genuine atomic multicast.

3. *Weakening the progress property of update transactions.* To allow update transactions to abort even if no other conflicting and pending transaction exists in the system. Clock-SI [48], and SCORe [104] employ this technique. For instance, authors of SCORe have proposed a GPR algorithm that supports both SER and WFQ in the failure-free case.
SCORe sidesteps the impossibility result by dropping obstruction-freedom for updates in certain scenarios. In more detail, this algorithm numbers every version with a scalar. If a transaction T_i first reads an object x then updates an object y, in case the version of x is smaller than the latest version of y, say y_k, T_i will not be able to read y_k, and it will thus abort.
In case of Clock-SI, if clocks of processes are not synchronized, an update transaction T_i can abort several times even if it is the only transaction being executed in the system. In more detail, consider transaction T_i that first reads an object x from a process p with a slow clock, and assigns a time t_p as its snapshot timestamp. If it later tries to read an object y, and modify y that is stored on a process q with an advanced clock, it cannot read versions of objects committed after t_p at process q. Hence, it needs to abort T_i and retries. Transaction T_i can commit only when the commit timestamp of the latest version of object y becomes smaller than the clock of process p.

4. *Weakening consistency.* Another technique is to sidestep the impossibility results by weakening consistency guarantees. In the next chapter, we investigate this approach, and introduce a new consistency criterion that is weakened just enough to circumvent the impossibility results.

4.4 Conclusion

In this chapter, we showed that ensuring snapshot isolation (SI) in a genuine partial replication system is impossible. To state this impossibility result, we proved that SI is decomposable into a set of simpler properties, and proved that two of these properties, namely snapshot monotonicity

and strictly consistent snapshots cannot be ensured. As a corollary, we also showed that a GPR system with obstruction-free updates cannot support SSER, nor Opacity. Moreover, a GPR system ensuring OFU and WFQ cannot ensure PSI.

CHAPTER 5

NMSI: NON-MONOTONIC SNAPSHOT ISOLATION

Contents

As we studied in Chapter 3, and Chapter 4, none of the previously published strong consistency criteria is able to ensure the following four scalability properties: (i) WFQ, (ii) GPR, (iii) forward freshness, and (iv) minimal commitment synchronization. In this chapter, we introduce a new consistency criterion, named Non-Monotonic Snapshot Isolation (NMSI), that both satisfies strong safety properties and ensures the above scalability properties.

We also introduce Jessy, a key-value store ensuring NMSI. Jessy uses dependence vectors, a novel data type that enables the efficient computation of consistent snapshots.

At the end of this chapter, we present our empirical evaluation results regarding the scalability of NMSI, along with a careful and fair comparison against a number of classical criteria, including SER, US, SI and PSI .

5.1 Definition of NMSI

NMSI retains the most important properties of SI and PSI, namely snapshots are consistent, a read-only transaction can commit locally without synchronization, and two concurrent conflicting updates do not both commit. But, NMSI does not need to respect strictness for consistent snapshots (i.e., SCONSa or SCONSb), or to ensure monotonicity of snapshots (i.e., MON). More formally:

Definition 5.1 (Non-monotonic Snapshot Isolation (NMSI))**.** A history h is in NMSI iff h belongs to $\text{ACA} \cap \text{CONS} \cap \text{WCF}$.

Applicative Anomalies Table 5.1(a) compares NMSI to other criteria based on the anomalies that an application might observe. We also include Read-Committed consistency criterion [22] (RC): a weak criterion (unlike the rest) that is widely used in industrial databases as a default criterion (e.g., SAP HANA [120], Oracle 11g [110], Microsoft SQL Server 2008 through 2014 [95], Azure SQL Database [95], and PostgreSQL 8.4 through 9.3 [108]). RC only disallows Dirty Read, and Dirty Write anomalies.

Write skew, the classical anomaly of SI, is observable under all criteria that ensure minimal commitment synchronization; thus histories in NMSI also observe this anomaly. Real-time violation happens when a transaction T_i observes the effect of some transaction T_j, but does not observe the effect of all the transactions that precede T_j in real-time. This issue occurs under serializability as well; this argues that it is not considered a problem in practice. Non-monotonic snapshots also occur in US (among read-only transactions) and in PSI; following Garcia-Molina and Wiederhold [57], Sovran et al. [136], we believe that this is a small price to pay for improved performance.

Scalability Properties With Table 5.1(b), we turn our attention to the scalability properties of each criterion in a transactional system. To make our comparison fair, we consider that the

Anomalies	Consistency Criteria						
	SSER	SER	US	SI	PSI	NMSI	RC
Dirty Reads	x	x	x	x	x	x	x
Non-Repeat. Reads	x	x	x	x	x	x	-
Read Skew	x	x	x	x	x	x	-
Dirty Writes	x	x	x	x	x	x	x
Lost Updates	x	x	x	x	x	x	-
Write Skew	x	x	x	-	-	-	-
Non-monotonic Snapshot among R-O txns	x	x	x	-	-	-	-
Non-monotonic Snapshot among R-O and UP txns	x	x	-	-	-	-	-
Real-time Violation	x	-	-	-	-	-	-

(a) *Disallowed Anomalies (x*:disallowed*)*

Scalability Properties	SSER	SER	US	SI	PSI	NMSI	RC
Genuine Partial Replication	†	†	-	†	†	-	-
Forward Freshness Snapshot	-	-	-	‡	‡	-	-
Min. Commitment Synchronization	‡	‡	‡	-	-	-	-

(b) *Scalability Properties (*†:Not Possible with OFU and WFQ, ‡:Not Possible*)*

Table 5.1: Comparing Consistency Criteria

system ensuring each criterion also ensures OFU, and WFQ.

In the previous chapter, we showed that none of SSER, SER, SI and PSI are implementable under GPR when queries are wait-free and update transactions are obstruction-free.

Peluso et al. [106] show that EUS, and consequently US, can combine GPR and WFQ. NMSI can also conjointly satisfy these two properties.

As pointed out in Chapter 4, both PSI and SI requires SCONSa (base freshness), thus disallowing forward freshness. However, NMSI does not have this requirement, hence it allows forward freshness. Observe that NMSI is weaker than PSI in term of acceptable histories, and not in term of anomalies.

To avoid the write-skew anomaly, SSER, SER, and US need to certify update transactions with respect to read-write and write-write conflicts. Hence, they do not provide minimal commitment synchronization.

In the next section, we introduce Jessy with the NTU progress property (called Jessy). In Section 5.3, we extend Jessy such that it ensure obstruction-free update.

5.2 Jessy: a Protocol for NMSI

We now describe Jessy, a scalable transactional protocol that ensures NMSI and guarantees the four scalability properties above. Because distributed locking policies do not scale [61, 145], Jessy employs deferred update replication: transactions are executed optimistically, then certified by a

termination protocol. Jessy uses a novel clock mechanism to ensure that snapshots are both fresh and consistent, while preserving wait-freedom of queries and genuineness. We describe it in the next section. To help the readability, we defer some proofs to Appendix B.

5.2.1 Taking Consistent Snapshots

Constructing a shared snapshot object, and database checkpointing (i.e., taking consistent snapshots) are classical problems in distributed system literature [5, 19, 20, 53, 85]. Nevertheless, in our context, two difficulties arise: (i) multiple updates might be related to the same transaction, and (ii) the construction should be both genuine and wait-free. This problem has been addressed in several protocols [106, 129] previously. In this section, we propose another solution. To achieve the above properties, Jessy uses a novel data type called *dependence vectors*. Each version of an object is assigned its own dependence vector. Therefore, the dependence vector of some version x_i reflects all the versions read by T_i, or read by the transactions on which T_i depends, as well as the writes of T_i itself:

Definition 5.2 (Dependence Vector)**.** A dependence vector is a function V that maps every read (or write) operation $o(x)$ in a history h to a vector $V(o(x)) \in \mathbb{N}^{|Objects|}$ such that:

$$V(r_i(x_0)) = 0^{|Objects|}$$
$$V(r_i(x_j)) = V(w_j(x_j))$$
$$V(w_i(x_i)) = max\ \{V(r_i(y_j)) \mid y_j \in rs(T_i)\}\ +\ \Sigma_{z_i \in ws(T_i)}\ 1_z$$

where $max\ \mathcal{V}$ is the vector containing for each dimension z, the maximal z component in the set $\mathcal{V}$, and 1_z is the vector that equals 1 on dimension z, and 0 elsewhere.

To illustrate this definition, consider history h_5 below. In this history, transactions T_1 and T_2 update objects x and y respectively, and transaction T_3 reads x then updates y. The dependence vector of $w_1(x_1)$ equals $\langle 1,0\rangle$, and it equals $\langle 0,1\rangle$ for $w_2(y_2)$. Since transaction T_3 reads x_1 then updates y after reading version y_2, the dependence vector of $w_3(y_3)$ equals $\langle 1,2\rangle$.

$$h = \quad \begin{array}{l} r_1(x_0).w_1(x_1).c_1 \searrow \\ \qquad\qquad\qquad\quad r_3(x_1).r_3(y_2).w_3(y_3).c_3 \\ r_2(y_0).w_2(y_2).c_2 \nearrow \end{array}$$

Consider a transaction T_i and two versions x_j and y_l read by T_i. We shall say that x_j and y_l are *compatible* for T_i, written $compat(T_i, x_j, y_l)$, when both $V(r_i(x_j))[x] \geq V(r_i(y_l))[x]$ and $V(r_i(y_l))[y] \geq V(r_i(x_j))[y]$ hold. Using the compatibility relation, we can prove that dependence vectors fully characterize consistent snapshots. To this end, we first prove a technical lemma, then we state our main theorem.

Lemma 5.1. *Consider a history h in WCF, and two transactions T_i and T_j in h. Then,*

$$T_i \rhd^* T_j \Leftrightarrow \forall x, y \in Objects : \forall w(x), w(y) \in T_i \times T_j : V(w_i(x_i)) > V(w_j(y_j))$$

Proof. The proof goes as follows:

- ($\Rightarrow$) First consider that $T_i \rhd T_j$ holds. By definition of relation $\rhd$, we know that for some object z, operations $r_i(z_j)$ and $w_j(z_j)$ are in h. According to definition of function V we have: $V(w_i(x_i)) \geq V(r_i(z_j)) + 1_x$. Besides, always according to the definition of V, it is true that the following equalities hold: $V(r_i(z_j)) = V(w_j(z_j)) = V(w_j(y_j))$. Thus, we have: $V(w_i(x_i)) > V(w_j(y_j))$. The general case $T_i \rhd^* T_j$ is obtained by applying inductively the previous reasoning.
- ($\Leftarrow$) From the definition of function V, it must be the case that $r_i(y'_{j'})$ is in h with $j' \neq 0$. We then consider the following two cases: (Case $j' = j$) By definition of relation $\rhd$, $T_i \rhd T_j$ holds. (Case $j' \neq j$) By construction, we have that: $T_i \rhd T_{j'}$. By definition of function V, we have that $V(r_{j'}(y_{j'})) = V(w_{j'}(y_{j'}))$. Since $V(w_i(x_i)) > V(w_j(y_j))$ holds, $V(w_{j'}(y_{j'}))[y] \geq V(w_j(x_j))[y]$ is true. Both transactions T_j and $T_{j'}$ write y. Since h belongs to WCF, it must be the case that either $T_j \rhd^* T_{j'}$ or that $T_{j'} \rhd^* T_j$ holds. If $T_j \rhd^* T_{j'}$ holds, then we just proved that $V(w_j(y_j)) > V(w_{j'}(y_{j'}))$ is true. A contradiction. Hence necessarily $T_{j'} \rhd^* T_j$ holds. From which we conclude that $T_i \rhd^* T_j$ is true.

■

Theorem 5.1. *Consider a history h in WCF and a transaction T_i in h. Transaction T_i sees a consistent snapshot in h iff every pair of versions x_l and y_j read by T_i is compatible (i.e., $V(r_i(x_l))[x] \geq V(r_i(y_j))[x]$ holds).*

Proof. The proof goes as follows:

- ($\Rightarrow$) By contradiction. Assume the existence of two versions x_l and y_j in the snapshot of T_i such that $V(r_i(x_l))[x] < V(r_i(y_j))[x]$ holds. By definition of function V, we have $V(r_i(x_l)) = V(w_l(x_l))$ and $V(r_i(y_j)) = V(w_j(y_j))$. Hence, $V(w_l(x_l))[x] < V(w_j(y_j))[x]$ holds. Again from the definition of function V, there exists a transaction $T_{k\neq 0}$ writing on x such that (i) $V(w_j(y_j)) \geq V(w_k(x_k))$ and (ii) $V(w_j(y_j))[x] = V(w_k(x_k))[x]$. Applying Lemma 5.1 to (i), we obtain $T_j \rhd^* T_k$. From which we deduce that $T_i \rhd^* T_k$. Now since both transactions T_l and T_k write x and h belongs to WCF, $T_l \rhd^* T_k$ or $T_k \rhd^* T_l$ holds. From (ii) and $V(w_l(x_l))[x] < V(w_j(y_j))[x]$, we deduce that $V(w_l(x_l))[x] < V(w_k(x_k))[x]$. As a consequence of Lemma 5.1, $T_k \rhd^* T_l$ holds. Hence $x_l \ll_h x_k$. But $T_i \rhd^* T_k$ and $r_i(x_l)$ is in h. It follows that T_i does not read a consistent snapshot. Contradiction.
- ($\Leftarrow$) By contradiction. Assume that there exists an object x and a transaction T_k on which T_i depends such that T_i reads version x_j, T_k writes version x_k, and $x_j \ll_h x_k$. First of all, since h is in WCF, one can easily show that $T_k \rhd^* T_j$. Since $T_k \rhd^* T_j$, Lemma 5.1 tells us that $V(w_k(x_k)) > V(w_j(x_j))$ holds. Since $T_i \rhd^* T_k$ holds, a short induction on the definition of function V tells us that $V(r_i(x_j))[x] \geq V(w_k(x_k))|x]$ is true. Hence, $V(r_i(x_j))[x] \geq V(w_k(x_k))[x] > V(w_j(x_j))[x] = V(r_i(x_j))[x]$. Contradiction.

■

Despite that in the common case dependence vectors are sparse, they might be large for certain workloads. For instance, if transactions execute random accesses, the size of each vector tends asymptotically to the number of objects in the system. To address the above problem, Jessy employs a mechanism to approximate dependencies safely, by coarsening the granularity, grouping objects into disjoint partitions and serializing updates in a group as if it was a single larger object. We cover this mechanism in what follows.

Consider some partition $\mathcal{P}$ of *Objects*. For some object x, note $P(x)$ the partition x belongs to, and by extension, for some $S \subseteq Objects$, note $P(S)$ the set $\{\mathcal{P}(x) \mid x \in S\}$. A partition is *proper* for a history h when updates inside the same partition are serialized in h, that is, for any two writes $w_i(x_i)$, $w_j(y_j)$ with $\mathcal{P}(x) = \mathcal{P}(y)$, either $w_i(x_i) <_h w_j(y_j)$ or the converse holds.

Now, consider some history h, and for every object x replace every operation $o_i(x)$ in h by $o_i(\mathcal{P}(x))$. We obtain a history that we note $h^{\mathcal{P}}$. The following result links the consistency of h to the consistency of $h^{\mathcal{P}}$:

Proposition 5.1. *Consider some history h. If $\mathcal{P}$ is a proper partition of Objects for h and history $h^{\mathcal{P}}$ belongs to CONS, then h is in CONS.*

Proof. We observe that for any two transactions T_i and T_j:

- If $T_i \rhd^* T_j$ holds in h then $T_i \rhd^* T_j$ holds in $h^{\mathcal{P}}$.
  Proof. If $T_i \rhd T_j$ holds in h, then $r_i(x_j)$ is in h. Thus $r_i(\mathcal{P}(x_j))$ is in $h^{\mathcal{P}}$. It follows that $T_i \rhd T_j$ holds in $h^{\mathcal{P}}$. If $T_i \rhd^* T_j$ in h then there exist a set of transactions $\{T_1, \ldots, T_m\}$ such that: $T_i \rhd T_1 \ldots \rhd T_m \rhd T_j$ hold in h. From the result above, we deduce that $T_i \rhd T_1 \ldots \rhd T_m \rhd T_j$ hold in $h^{\mathcal{P}}$. Hence, $T_i \rhd^* T_j$ holds in $h^{\mathcal{P}}$. □

- If $x_i \ll x_j$ holds in h then $\mathcal{P}(x_i) \ll \mathcal{P}(x_j)$ holds in $h^{\mathcal{P}(x)}$.

□

For the sake of contradiction, assume that $h^{\mathcal{P}}$ is in CONS while h is not in CONS. It follows that there exist a transaction T_i, some object x and a transaction T_k on which T_i depends such that in h, T_i reads version x_j, T_k writes version x_k, and $x_j \ll_h x_k$. From the two observations above, we obtain that $T_i \rhd T_j$, $T_i \rhd^* T_k$ and $\mathcal{P}(x_j) \ll_h \mathcal{P}(x_k)$ hold in $h^{\mathcal{P}}$. Hence, $h^{\mathcal{P}}$ is not consistent. Contradiction.

■

Given two operations $o_i(x_j)$ and $o_k(y_l)$, let us introduce relation $o_i(x_j) \leq_h^{\mathcal{P}} o_k(y_l)$ when $o_i(x_j) = o_k(y_l)$, or $o_i(x_j) <_h o_k(y_l) \wedge \mathcal{P}(x) = \mathcal{P}(y)$ holds. Based on Proposition 5.1, we define below a function that approximates dependencies safely:

Definition 5.3 (Partitioned Dependence Vector)**.** A function PV is a partitioned dependence vector when PV maps every read (or write) operation $o(x)$ in a history h to a vector $PV(o(x)) \in \mathbb{N}^{|\mathcal{P}|}$

such that:

$$PV(r_i(x_0)) = 0^{|\mathcal{P}|}$$
$$PV(r_i(x_j)) = max\ \{PV(w_l(y_l)) \mid w_l(y_l) \leq_h^{\mathcal{P}} r_i(x_j) \wedge \left(\forall k : x_j \ll_h x_k \Rightarrow w_l(y_l) \leq_h^{\mathcal{P}} w_k(x_k)\right)\}$$
$$PV(w_i(x_i)) = max\ \{PV(r_i(y_j)) \mid y_j \in rs(T_i)\} \cup \{PV(w_k(z_k)) : w_k(z_k) \leq_h^{\mathcal{P}} w_i(x_i)\} + \Sigma_{X \in \mathcal{P}(ws(T_i))}\ 1_X$$

The first two rules of function PV are identical to the ones that would give us function V on history $h^{\mathcal{P}}$. The second part of the third rule serializes objects in the same partition

When Jessy uses partitioned dependence vectors and $\mathcal{P}$ is a proper partition for h, Theorem 5.1 holds for the following definition of $compat(T_i, x_j, y_l)$:

Case $\mathcal{P}(x) \neq \mathcal{P}(y)$**.** This case is identical to the definition we gave for function V. In other words, both $PV(r_i(x_j))[\mathcal{P}(x)] \geq PV(r_i(y_l))[\mathcal{P}(x)]$ and $PV(r_i(y_l))[\mathcal{P}(y)] \geq PV(r_i(x_j))[\mathcal{P}(y)]$ must hold.

Case $\mathcal{P}(x) = \mathcal{P}(y)$**.** This case deals with the fact that inside a partition writes are serialized. We have (i) if $PV(r_i(x_j))[\mathcal{P}(y)] > PV(r_i(y_l))[\mathcal{P}(y)]$ holds then $y_l = max\ \{y_k \mid w_k(y_k) \leq_h^{\mathcal{P}} w_j(x_j)\}$, or symmetrically (ii) if $PV(r_i(y_l))[\mathcal{P}(x)] > PV(r_i(x_j))[\mathcal{P}(x)]$ holds then $x_j = max\ \{x_k \mid w_k(x_k) \leq_h^{\mathcal{P}} w_l(y_l)\}$, or otherwise (iii) the predicate equals *true*.

We prove next that the "if" part of Theorem 5.1 holds for the above definition of compatibility:

Proposition 5.2. *Consider a history h in NMSI and a transaction T_i in h. If every pair of versions x_j and y_l read by T_i is compatible, then transaction T_i sees a consistent snapshot in h*

Proof. Using a reasoning identical to the one we depicted in the proof of Theorem 5.1, we can prove that $h^{\mathcal{P}}$ belongs to CONS. Then, from Proposition 5.1, we know that if $h^{\mathcal{P}}$ belongs to CONS, then h belong to CONS. ■

5.2.2 Transaction Lifetime in Jessy

Jessy is a distributed system of processes which communicate by message passing. Each process executing Jessy holds a local data store denoted as ds. A data store contains a finite set of tuples (x, v, i), where x is an object (data item), v a value, and i a version. Jessy supports GPR, and consequently two processes may store different set of objects. We recall that for an object x, we note $replicas(x)$ the processes that store a copy of x, and by extension, $replicas(X)$ the processes that store one of the objects in X.

When a client (not modeled) executes a transaction T_i with Jessy, T_i is handled by a coordinator, denoted $coord(T_i)$. The coordinator of a transaction can be any process in the system. In what follows, $replicas(T_i)$ denotes the replica set of T_i, that is $replicas(rs(T_i) \cup ws(T_i))$.

A transaction T_i can be in one of the following four states at some process:

- *Executing*: Each non-termination operation $o_i(x)$ in T_i is executed optimistically (i.e., without synchronization with other replicas) at the transaction coordinator $coord(T_i)$. If $o_i(x)$ is a read, $coord(T_i)$ returns the corresponding value, fetched either from the local

Algorithm 1 Execution Protocol of Jessy

```
1: Variables:
2:     ds, submitted, committed, aborted
3:
4: execute(READ, x, T_i)
5:     eff: if ∃(x,v,i) ∈ up(T_i) then return v
6:          else
7:              send ⟨REQ, T_i, x⟩ to replicas(x)
8:              wait until received ⟨REPLY, T_i, x, v⟩
9:              return v
10:
11: execute(WRITE, x, v, T_i)
12:     eff: up(T_i) ← up(T_i) ∪ {(x,v,i)}
13:
14: remoteRead(x, T_i)
15:     pre: received ⟨REQ, T_i, x⟩ from q
16:          ∃(x,v,j) ∈ ds : ∀y_l ∈ rs(T_i) : compat(T_i, x_j, y_l)
17:     eff: send ⟨REPLY, T_i, x, v⟩ to q
18:
19: execute(TERM, T_i)
20:     eff: submitted ← submitted ∪ {T_i}
21:          wait until T_i ∈ decided
22:          if T_i ∈ committed then return COMMIT
23:          return ABORT
24:
```

replica or a remote one. If $o_i(x)$ is a write, $coord(T_i)$ stores the corresponding update value in a local buffer, enabling *(i)* subsequent reads to observe the modification, and *(ii)* a subsequent commit to send the write-set to remote replicas.

- *Submitted*: Once all the read and write operations of T_i have executed, T_i terminates, and the coordinator submits it to the termination protocol. The protocol applies a certification test on T_i to enforce NMSI. This test ensures that if two concurrent conflicting update transactions terminate, one of them aborts.
- *Committed/Aborted*: When T_i enters the *Committed* state at $r \in replicas(T_i)$, its updates (if any) are applied to the local data store. If T_i aborts, T_i enters the *Aborted* state.

5.2.3 Execution Protocol

Algorithm 1 describes the execution protocol in pseudocode. These protocols are explained as a set of atomic actions guarded by pre-conditions. An action is executed when its precondition becomes true. Logically, the protocol can be divided into two parts: action *remoteRead*(), executed at some process, reads an object replicated at that process in a consistent snapshot; and the coordinator $coord(T_i)$ performs actions *execute*() to execute T_i and to buffer the updates in $up(T_i)$.

The variables of the execution protocol are: *ds*, the local data store; *submitted* contains locally-submitted transactions; and *committed* (respectively *aborted*) stores committed (respectively aborted) transactions. We use the shorthand *decided* for $committed \cup aborted$.

Upon a read request for x, $coord(T_i)$ checks against $up(T_i)$ if x has been previously updated by the same transaction; if so, it returns the corresponding value (line 5) [1]. Otherwise, $coord(T_i)$ sends a read request to the processes that replicate x (lines 8 to 9). Conversely, when a process

[1] This mechanism is to ensure that every object is read at most once for every transaction in a history h.

Algorithm 2 Termination Protocol of Jessy

```
 1: Variables:
 2:     ds, submitted, committed, aborted, Q
 3:
 4: submit(T_i)
 5:     pre:  T_i ∈ submitted
 6:           ws(T_i) ≠ ∅
 7:     eff:  AM-Cast(T_i) to replicas(ws(T_i))
 8:
 9: deliver(T_i)
10:     pre:  T_i = AM-Deliver()
11:     eff:  Q ← Q ∘ ⟨T_i⟩
12:
13: vote(T_i)
14:     pre:  T_i ∈ Q \ decided
15:           ∀T_j ∈ Q, T_j <_Q T_i ⇒ T_j ∈ decided
16:     eff:  v ← certify(T_i)
17:           send ⟨VOTE, T_i, v⟩ to replicas(ws(T_i)) ∪ {coord(T_i)}
18:
19: commit(T_i)
20:     pre:  outcome(T_i) ≠ ⊥
21:     eff:  foreach (x, v, i) in up(T_i) do
22:               if x ∈ ds then ds ← ds ∪ {(x, v, i)}
23:           committed ← committed ∪ {T_i}
24:
25: abort(T_i)
26:     pre:  ¬outcome(T_i)
27:     eff:  aborted ← aborted ∪ {T_i}
28:
```

receives a read request for object x that it replicates, it returns a version of x which complies with Theorem 5.1 (lines 15 to 17).

Upon a write request of T_i, the process buffers the update value in $up(T_i)$ (line 12). During commitment, the updates of T_i will be sent to all replicas holding an object that is modified by T_i.

When transaction T_i terminates, it is submitted to the termination protocol (line 20). The execution protocol then waits until T_i either commits or aborts, and returns the outcome.

5.2.4 Termination Protocol

Algorithm 2 depicts the termination protocol of Jessy. It accesses the same four variables *ds*, *submitted* and *committed*, along with a FIFO queue named Q.

In order to satisfy GPR, the termination protocol uses a genuine atomic multicast primitive [66, 126]. This requires that either (i) we form non-intersecting groups of replicas, and an eventual leader oracle is available in each group, or (ii) that a system-wide *reliable* failure detector is available. The latter setting allows Jessy to tolerate a disaster [122].

To terminate an update transaction T_i, $coord(T_i)$ atomic-multicasts it to every process that holds an object written by T_i. Every such process p certifies T_i by calling function $certify(T_i)$ (line 16). This function returns *true* at process p, iff for every transaction T_j committed prior to T_i at p, if T_j write-conflicts with T_i, then T_i depends on T_j. Formally:

$$certify(T_i) \triangleq \forall T_j \in committed : ws(T_i) \cap ws(T_j) \neq \varnothing \Rightarrow T_i \rhd^* T_j$$

Under partial replication, a process p might store only a subset of the objects written by T_i, in which case p does not have enough information to decide on the outcome of T_i. Therefore, we introduce a voting phase where replicas of the objects written by T_i send the result of their certification test in a VOTE message to every process in $replicas(ws(T_i)) \cup \{coord(T_i)\}$ (line 17).

A process can safely decide on the outcome of T_i when it has received votes from a *voting quorum* for T_i. A voting quorum Q for T_i is a set of replicas such that for every object $x \in ws(T_i)$, the set Q contains at least one of the processes replicating x. Formally, a set of processes is a voting quorum for T_i iff it belongs to $vquorum(T_i)$, defined as follows:

$$vquorum(T_i) \triangleq \{Q \subseteq \Pi \mid \forall x \in ws(T_i) : \exists j \in Q \cap replicas(x)\}$$

A process p makes use of the following (three-values) predicate $outcome(T_i)$ to determine whether some transaction T_i commits, or not:

$outcome(T_i) \triangleq$
- **if** $ws(T_i) = \varnothing$
  - **then** *true*
- **else if** $\forall Q \in vquorum(T_i), \exists q \in Q,$ $\neg$*received* $\langle \text{VOTE}, T, \_ \rangle$ *from* q
  - **then** $\bot$
- **else if** $\exists Q \in vquorum(T_i), \forall q \in Q,$ *received* $\langle \text{VOTE}, T, true \rangle$ *from* q
  - **then** *true*
- **else** *false*

Observe that as long as the predicate $outcome(T_i)$ equals $\bot$, actions of $commit(T_i)$ cannot be executed. To commit transaction T_i, process p first applies T_i's updates to its local data store, then p adds T_i to variable *committed* (lines 20 to 23). If instead T_i aborts, p adds T_i to *aborted* (lines 26 to 27).

5.2.5 Sketch of Proof

This section shows that every history accepted by Jessy is in NMSI. Then, it proves that Jessy satisfies the four scalability properties we listed in Section 3.1. Both explanations are given in the broad outline, and a complete treatment is deferred to Appendix B.

5.2.5.1 Safety Properties

Since transactions in Jessy always read committed versions of the shared objects, Jessy ensures ACA. Theorem 5.1 states that transactions observe consistent snapshots, hence CONS is also satisfied. It remains to show that all the histories accepted by Jessy are write-conflict free (WCF).

Assume by contradiction that two concurrent write conflicting transactions T_i and T_j both commit. Note p_i (resp. p_j) the coordinator of T_i (resp. T_j), and let x be the object on which the conflict occurs (i.e., $x \in ws(T_i) \cap ws(T_j)$). According to the definition of function *outcome*, p_i (resp. p_j) has received a yes vote from some process q_i (resp. q_j). Hence, T_i (resp. T_j) is in variable $\mathcal{Q}$ at process q_i (resp. q_j) before it sends its vote message. One can show that either once q_i sends its vote, $T_j <_{\mathcal{Q}} T_i$ holds, or once q_j sends its vote, $T_i <_{\mathcal{Q}} T_j$ holds. Assume the former case holds (the proof for the latter is symmetrical). Because of line 15 in Algorithm 2, process q_i waits until T_j is decided before sending a vote for T_i. Due to the properties of atomic multicast, and the fact that $\mathcal{Q}$ is FIFO, T_j should be committed at q_i. Thus, *certify*(T_i) returns false at process q_i; contradiction.

5.2.5.2 Scalability Properties

We observe that in the case of a read-only transaction Jessy does not execute line 7 of Algorithm 2, and that the function *outcome* always returns true. Hence, such a transaction is wait-free. As previously mentioned, a transaction is atomic-multicast only to the replicas holding an object written by the transaction. Hence, the system ensures GPR. Forward freshness is reached by the *compat*() function, and the fact that we can read the most recent committed version of an object as long as it is consistent with previous reads. Finally, since only replicas holding objects modified by a transaction participate in its commitment, Jessy has minimum commitment synchronization.

5.3 Ensuring Obstruction-Freedom

In Jessy, a transaction T_i commits once its coordinator receives a voting quorum from all participating groups (i.e., replicas of $ws(T_i)$). However, some replica p inside a group may be slow, and commit T_i later. Therefore, some transaction T_j can read from p an older version of an object x that is modified by T_i. If transaction T_j later tries to update x, it must abort. Hence, Jessy cannot ensure OFU.

In this section, we introduce a variant of Jessy that also ensures OFU, called Jessy^{ofu}. The idea is that we modify Jessy such that once a transaction T_i commits at its coordinator, its effects become visible to every other transaction in the system.

We can transform Jessy to Jessy^{ofu} with the following changes:

1. *coord*(T_i) should read from a quorum. In Jessy, a coordinator returns as long as the first replica holding an object replies back (see line 8). In Jessy^{ofu}, it must wait to receive read replies from a quorums q_r of replicas holding an object.

2. *coord*(T_i) should write to a quorum. In Jessy, a coordinator returns as soon as it receives a vote from a replica in each replica group (see definition of *outcome*(T_i)). In Jessy^{ofu}, a coordinator must wait to receive votes from a quorums q_w of replicas holding an object such that $q_r \cap q_w \neq \emptyset$.

3. instead of also sending votes to $coord(T_i)$ once T_i is certified (line 17), replicas should send their votes to the coordinator when the outcome of T_i is known (i.e., after line 23).

5.4 Empirical study

5.4.1 Implementation

Jessy is written in Java inside our framework called (G-DUR), which we will explain in details in Chapter 6. We implemented Jessy as a middleware based on Algorithms 3 and 4. To minimize noise, and to focus on the scalability and synchronization costs in our experiments, the database is an in-memory concurrent hashmap, even though Jessy normally uses BerkeleyDB.

We also implemented a number of replication protocols that are representative of different consistency criteria (SER, SI, US and PSI). The protocols all support partial replication; furthermore the US and SER implementations ensure GPR. The following table summarizes the criteria and the corresponding protocols:

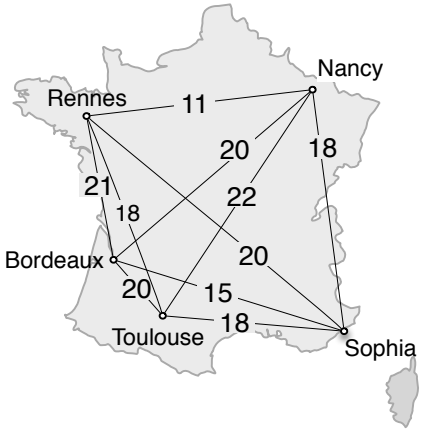
Criterion	Protocol	Difference with the original protocol
SER	P-Store [127]	-
US	GMU [106]	AM-Cast instead of 2PC
SI	Serrano07 [131]	-
PSI	Walter [136]	AM-Cast instead of 2PC

Our implementations closely follow the published specification of each protocol and are highly optimized. As they are all based on deferred update, their structure is very similar, and we were able to use the Jessy framework with relatively small variations. All our implementations use genuine atomic multicast [66, 126], even when the original used 2PC. The common structure, the use of the same multicast, and careful optimization ensure that the comparison is fair.

The protocols all support wait-free queries, except for SER, which trades it for GPR. Since the performance of US represents an upper bound on the performance of SER with wait-free queries, this decision allows us to isolate the cost of not ensuring the property. We also implemented a (weakly-consistent) deferred-update RC, to show the maximum achievable performance.

5.4.2 Setup and Benchmark

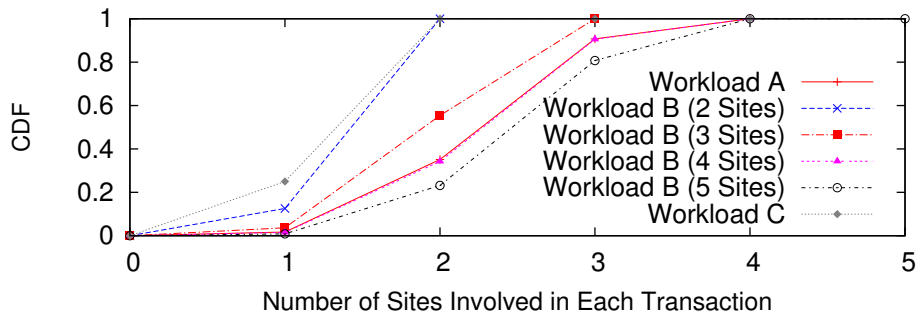
Figure 5.1 sums-up our experimental settings. All experiments are run on different sites of the French Grid'5000 experimental testbed [62], as illustrated in Figure 5.1(a). We always use four cores of machines with 2.2 GHz to 2.6 GHz processors, and a maximum heap size of 4 GB. For each server machine, two additional client machines generate the workload. Thus, there is no shared memory between clients and servers.



(a) *Sites and Latencies (in ms)*

	Key Selection Distribution	Operations	
		Read-Only Transaction	Update Transaction
A	Zipfian	4 Reads	2 Reads, 2 Updates
B	Uniform	4 Reads	3 Reads, 1 Update
C	Uniform	2 Reads	1 Read, 1 Update

(b) *Workload Types*



(c) *CDF of Number of Sites Involved in Each Transaction*

Figure 5.1: Experimental Settings

Every object is replicated across a multicast group of three replicas. We assume that each group as a whole is correct, i.e., it contains a majority of correct replicas. Every group contains 10^5 objects, replicated at each replica in the group, and each object has a payload size of 1 KB. Every group is replicated at a single site (no disaster tolerance). To study the scalability effects of consistency criteria in geo-replication, all our experiments are performed with global transactions. Clients are simply distributed uniformly way between the sites.

We use the Yahoo! Cloud Serving Benchmark (YCSB) [37], modified to generate transactional workload. This benchmark is used for evaluations in several other transactions protocols (such as

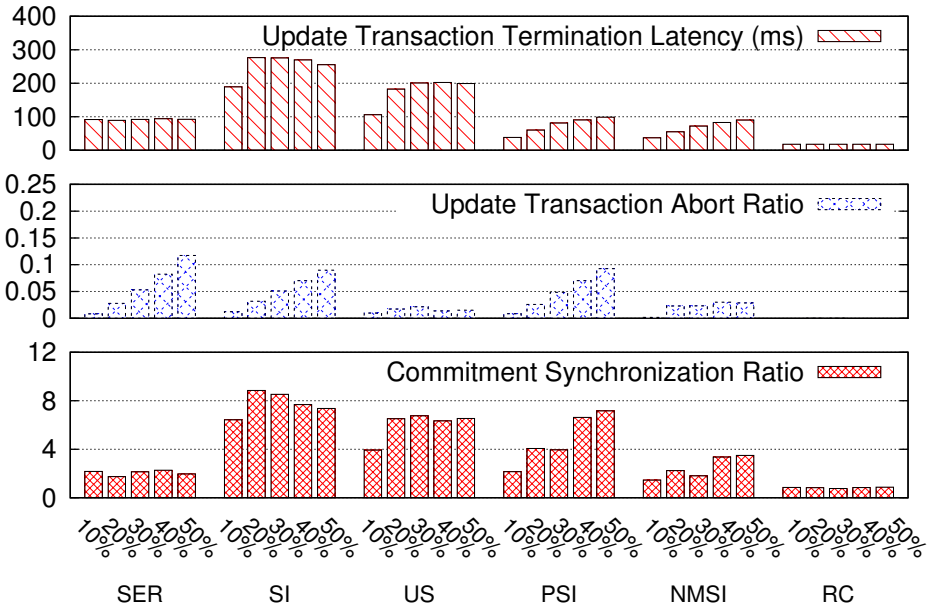


Figure 5.2: Update Transaction Termination Latency (on 4 sites)

[9, 48, 104, 106, 150]). Figure 5.1(b) describes the workloads used. For each workload, one client machine emulates multiple client threads in parallel, each being executed in closed loop. In all our experiments, a client machine executes at least 10^6 transactions. Figure 5.1(c) plots the CDF of the number of sites involved in each transaction.

The code of all the protocols, benchmarks, and scripts we used in the experiments are publicly available [116].

5.4.3 Experimental Results

We first study the impact of freshness and commitment synchronization on the latency of update transactions. Figure 5.2 depicts our results for workload A. The experiment is performed by varying the proportion of update/read-only transactions from 10%/90% (left) to 50%/50% (right). The load is limited so that the CPU of each replica is never saturated. The zipfian distribution is scrambled in order to scatter popular keys across different sites.

Forward Freshness: The abort ratio of update transactions, in the second graph of Figure 5.2, shows the effect of forward freshness. As expected, NMSI and US have the smallest abort rate, thanks to their fresher snapshots. The abort rate of US is one or two percent better than NMSI. This is mainly because NMSI is faster than US, and therefore it processes more transactions. In contrast, PSI and SI both take snapshots at the start of a transaction (SCONSa), resulting in an almost identical abort ratio, higher than NMSI and US. SER has the highest abort rate because

in our implementation only the certification test ensures that a transaction reads a consistent snapshot.

Minimal Commitment Synchronization: The third graph of Figure 5.2 studies the effect of commitment synchronization. We measure here the ratio of termination latency over solo termination latency, i.e., the time to terminate a transaction in the experiment divided by the time to terminate a transaction without contention. The ratio for RC equals 1, the optimum. This means that increasing concurrency does not increase the latency of update transactions. NMSI also has a small commitment synchronization cost. It is slightly higher for PSI, because PSI is non-genuine, and propagates when committing. This, along with its higher abort ratio, results in a termination latency increase of approximately 10 ms. SI has the highest termination latency, due to a high commitment convoy effect (because it is non-genuine), and a high abort ratio. A convoy effect occurs when the certification of a global transaction T_i is delayed by another transaction, even though T_i can be certified [127]. SER low termination latency is explained by the fact that SER synchronizes both read-only and update transactions, resulting in lower thread contention than the criteria that support wait-free queries.

We now turn our attention to the impact of wait-free queries and genuine partial replication on performance. To this goal, we measure the maximal throughput of each criterion as the number of sites increases. Figure 5.3 depicts our results for workload B with 90% read-only transactions, and 10% update transactions.

Wait-Free Queries: According to Figure 5.3, ensuring WFQ have a great performance impact. Recall that our implementation of SER favors GPR over WFQ. Observe that the maximum throughput of SER is at least two times lower than the other criteria. This emphasizes how crucial this property is for scalability.

Genuine Partial Replication: To see the effects of GPR on system performance, we first compare PSI and NMSI. If the system consists in a single site, their throughput is almost identical. However, PSI does not scale as well as NMSI as the number of sites increases: NMSI scales as linearly as RC; with five sites, its throughput is almost double of PSI. Although SI outperforms US up to three sites, it falls behind with four sites or more, and with five sites, its throughput drops substantially due to non-genuineness. Under four sites the effect is small, but with four or more sites, genuineness pays off, and US outperforms SI. Since US does not minimize commitment synchronization, its synchronization cost becomes high at 5 sites, decreasing its throughput.

We close this empirical evaluation by a detailed comparison of the scalability performance of NMSI with respect to other criteria.

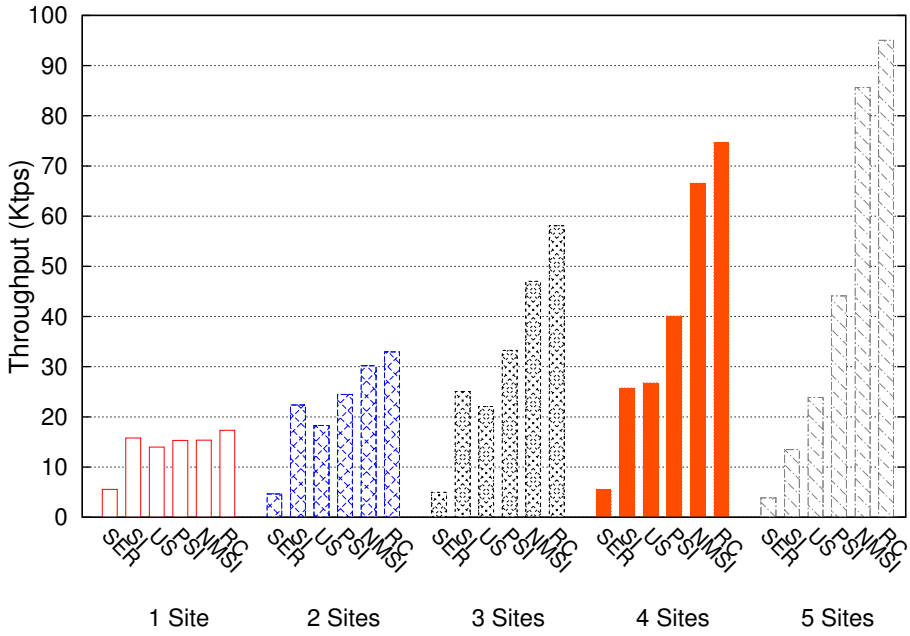


Figure 5.3: Maximum Throughput of Consistency Criteria

Overall Scalability: Figure 5.3 shows that performance of NMSI is comparable to RC, and between two to fourteen times faster than well-known strong consistency criteria. Our last experiment addresses the scalability of NMSI when the number of sites is constant. To this goal, we use workload C and four sites. Figure 5.4 shows our results. The load increases from left to right. We also vary the proportion of update/read-only transactions, between 10%/90% to 30%/70% (bottom to top). Since workload C has few reads, SER and US do not suffer much from non-minimal commitment synchronization. For a given criterion, termination latency varies from a low end, with 10% of update transactions, to a high end with 30% of update transactions. The throughput of NMSI is similar to RC, with excellent termination latency, thanks to the combination of GPR and forward freshness. Similarly, these same properties help US to deliver better performance than PSI with a lower termination latency, when the proportion of updates reaches 30%.

5.5 Conclusion

This chapter introduces Non-Monotonic Snapshot Isolation (NMSI). NMSI is the strong consistency criterion gathering the following four properties: Genuine Partial Replication, Wait-Free Queries, Forward Freshness Snapshot, and Minimal Commitment Synchronization. The conjunction of the above properties ensures that NMSI completely leverages the intrinsic parallelism of the workload and reduces the impact of concurrent transactions on each others. We also assess empirically these benefits by comparing our NMSI implementation with the implementation of

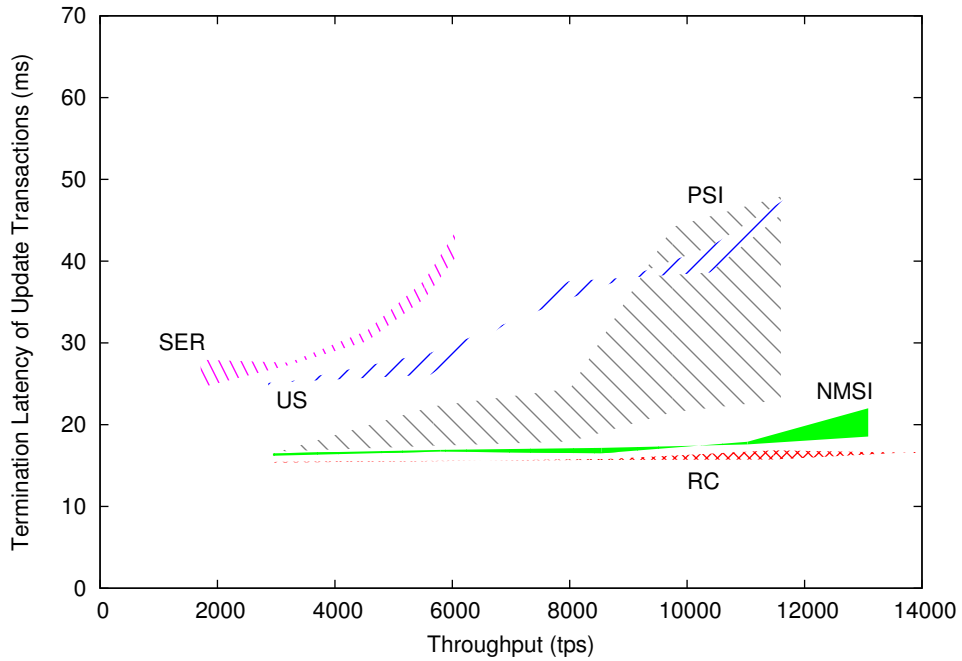


Figure 5.4: Comparing the throughput and termination latency of update transactions for different protocols

several replication protocols representative of well-known criteria. Our experiments show that NMSI is close to RC (i.e, the weakest criterion) and up to two times faster than PSI.

CHAPTER 6

G-DUR: GENERIC DEFERRED UPDATE REPLICATION

Contents

In the previous chapters, we introduced several consistency criteria, along with the protocols ensuring them, and performed an anomalistic comparison (i.e., comparison in terms of anomalies) between them. It still remains difficult to understand what are the important differences between protocols, and to make an objective, scientific comparison of their real-world behavior.

In this chapter, we propose an approach to this issue between empirical and formal: we compare them by using implementation variants. Our insight is that many deferred update replication (DUR) protocols (such as [12, 100, 103, 104, 106, 117, 125, 127, 129–131, 136]) share a common structure, and differ only by specific instantiations of a few generic functions. For instance, they all have a read phase, differing by their choice of which specific object version to read; and a termination phase, differing only by how they detect and resolve concurrency conflicts.

We express this insight as a common algorithmic structure, with well-identified realization points. This generic structure is instantiated into a specific protocol by selecting appropriate plug-ins from a library. For instance, for a serializable protocol, the read plug-in will select the most recent committed version of an object, and the termination plug-in will abort any transaction if it is concurrent with an already-committed conflicting transaction.

We implement this structure as a generic transactional middleware, called Generic DUR (G-DUR). G-DUR implements the generic structure, and offers a library of highly-optimized plug-ins. In particular, G-DUR supports the following customizations: *(i)* Different optimistic read protocols, differing by their versioning mechanisms and their freshness guarantees. *(ii)* Various certification procedures, differing by the amount of transactions they handle in parallel and the way they manage concurrency conflicts. *(iii)* Different commitment protocols, based on the group communication (atomic broadcast or multicast), Two-Phase commit, or Paxos Commit.

By mixing-and-matching the appropriate plug-ins, it is relatively easy to obtain a high-performance implementation of a protocol. We leverage this capability in an extensive experimental evaluation that we conduct in a geo-replicated environment.

(1) We tailor G-DUR to implement six prominent transactional protocols [106, 117, 127, 129, 131, 136]. Implementation of these protocols in G-DUR require only 200 to 600 lines of code. We also evaluate them empirically in an apples-to-apples comparison, which clearly brings out the differences between the protocols, and between the consistency criteria they implement. Our study shows that they have well-separated performance domains and, it enables us to precisely identify their respective limitations.

(2) We show how a developer can use G-DUR to understand finely the limitations of a protocol. We take a recently published protocol [106], and identify its bottlenecks by methodically replacing its plugs-ins by weaker ones.

(3) The previous approach also helps a developer to enhance existing protocols. We illustrate this point by presenting a variation of P-Store [127] that leverages workload locality to perform

Notation	Meaning	Notation	Meaning
x, y	object	T_i, $i \in \mathbb{N}$	transaction
x_i	version of x written by T_i	$o_i(x)$	T_i reads or writes object x
$coord(T_i)$	coordinator of T_i	$rs(T_i)$, $ws(T_i)$	read and write set of T_i
$T_i \parallel T_j$	T_i and T_j are concurrent	Π	set of all replicas

Table 6.1: Notations

up to 70% faster than the original protocol.

(4) In our last set of experiments, we evaluate the cost of various degrees of dependability. Based on a protocol ensuring serializability, we study the price of tolerating failures by varying the replication degree and the commitment algorithm.

The remainder of this chapter is organized as follows: We give an overview of the transactional middleware at core of G-DUR in Section 6.1. Section 6.2 details the execution phase of G-DUR, and we explain its termination, and atomic commitment protocols in Section 6.3. In Section 6.4, we show how to realize various protocols in G-DUR. The implementation of G-DUR is covered in Section 6.5. In Section 6.6, we conduct our experimental evaluation. We review related work in Section 6.7, and we conclude in Section 6.8.

6.1 Overview

Deferred Update Replication (DUR) is a widely used approach to build a fault-tolerant transactional datastore. Under the hood, this store is distributed and replicated across multiple replicas. Replicas synchronize each other to offer clients a consistent and live access to the datastore.

G-DUR is designed as a generic, tailorable, implementation of DUR. Figure 6.1 presents the global architecture of the middleware. Clients submit their transactions to G-DUR instances. A client may execute a transaction interactively, i.e., G-DUR does not require all the transactional code to be submitted at once. Certain optimizations are nevertheless possible if such an assumption holds. A transaction starts by a *begin* operation, followed by one or more CRUD operations (i.e., Create, Read, Update, or Delete) , and ends with *commit* or *abort*. Create, update, and delete operations are called write operations in what follows.

A G-DUR instance *coordinates* the transactional requests it receives from a client. To that end, an instance (i.e., a replica) holds a local datastore containing a subset of the globally available data, and it executes two customizable *execution* and *termination* protocols (see bottom of Figure 6.1). The execution protocol is responsible for reading data and for buffering after-values. The termination protocol handles the propagation of the transaction side effects, its commitment and the persistence of after-values.

At some G-DUR instance, a transaction T_i, can be in four distinct states: *Executing*, *Submitted*, *Committed* or *Aborted*. We comment on each of these states below. Table 6.1 recalls a summary of the notations used in the remainder of this chapter.

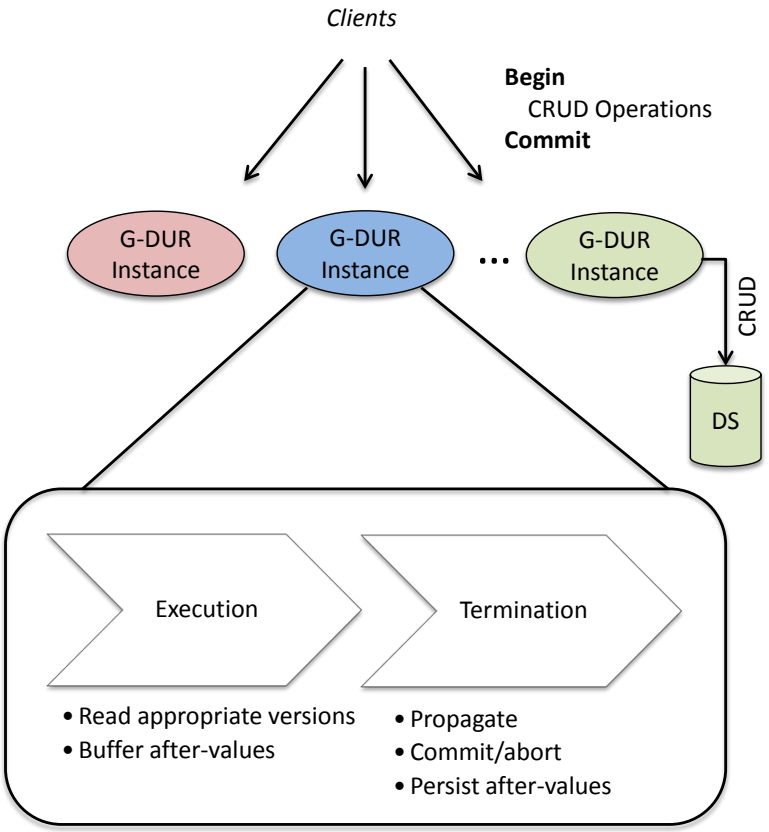


Figure 6.1: G-DUR Architecture

- *Executing*: Each operation $o_i(x)$ in T_i is executed speculatively at the coordinator, i.e., at the replica that receives the transaction from the client. If $o_i(x)$ is a read, the coordinator returns the corresponding value, fetched either from the local replica or a remote one. If $o_i(x)$ is a write, the coordinator stores the corresponding update value in a local buffer, enabling *(i)* subsequent reads to observe the modified value, and *(ii)* the subsequent commit to send the after-values to remote replicas.

- *Submitted*: Once all the read and write operations of T_i have executed, the coordinator submits it for termination. This includes synchronizing with the concerned replicas, and a *certification* check to satisfy the safety conditions of the implemented consistency criterion.

- *Committed*/*Aborted*: If certification is successful, T_i enters the *Committed* state, and every process $q \in replicas(T_i)$ applies the transaction's after-values (if any) to its copy of the datastore. Otherwise, T_i aborts, and enters the *Aborted* state. Consequently, its after-values are discarded.

Building upon the work of Wiesmann et al. [147], our key insight in the design of G-DUR is that *all* the DUR protocols satisfy the above description. In the next sections, we give additional detail on our generic execution and termination protocols. These protocols are explained as a set

Algorithm 3 Execution protocol - code at process p

```
1:  Variables:
2:      ds, committed, aborted, executing, submitted
3:
4:  execute(BEGIN, T_i)
5:      pre:  T_i ∉ executing ∪ executed
6:      eff:  executing ← executing ∪ {T_i}
7:            init(T_i)
8:
9:  execute(READ, T_i, x)
10:     pre:  T_i ∈ executing
11:     eff:  if ∃x_i ∈ ws(T_i) then return x_i
12:           else if isLocal(x) then
13:              return localRead(x)
14:           else
15:              send ⟨REQ, T_i, x⟩ to  q ∈ replicas(x)
16:              wait until
17:                 received ⟨REPLY, T_i, x_j⟩ from q
18:              return x_j
19:
20: execute(WRITE, T_i, x)
21:     pre:  T_i ∈ executing
22:     eff:  ws(T_i) ← ws(T_i) ∪ {x_i}
23:
24: execute(COMMIT, T_i)
25:     pre:  T_i ∈ executing
26:     eff:  submit(T_i)
27:
28: localRead(T_i, x)
29:     pre:  choose(x, T_i) ≠ ∅
30:     eff:  let x_j ∈ choose(x, T_i)
31:           return x_j
32:
33: remoteRead(T_i, x, q)
34:     pre:  received ⟨REQ, T_i, x⟩ from q
35:           choose(x, T_i) ≠ ∅
36:     eff:  let x_j ∈ choose(x, T_i)
37:           send ⟨REPLY, T_i, x_j⟩ to q
38:
```

of atomic actions guarded by pre-conditions. The customizable points (called *realization points*) appear as functions whose names are underlined and in blue in the algorithms, e.g., *choose*(). Concretely, a *realized* protocol will define the set of plug-ins to be called in lieu of the realization points.

6.2 Execution

Algorithm 3 shows the pseudo-code of the execution protocol from the perspective of a G-DUR instance (replica p). The description refers to the following variables: We note ds the local (partial) copy of the datastore. Variable *committed*, *aborted*, *executing* and *submitted* refer to four sets that serve to log the transactions that the replica executes.

A transaction T_i starts when action *execute*(BEGIN, T_i) is invoked at a replica p. In such a case, we say that p is the coordinator of T_i, denoted *coord*(T_i). Action *execute*(READ, T_i, x) describes how T_i reads some object x. First, *coord*(T_i) checks against the buffer $ws(T_i)$ in case T_i previously updated x. Otherwise, if the local datastore contains a copy of x, *coord*(T_i) reads it (lines 12 to 13). If none of the previous cases hold, *coord*(T_i) sends an (asynchronous) read request to some replica holding x (line 15). Such a request is re-iterated to another replica, in case no answer is returned after some time (not covered in Algorithm 3).

Local (i.e., the coordinator is p) and remote reads (when the coordinator has requested a read

from p) are handled by the actions *localRead* and *remoteRead* respectively. In both cases, the plug-in for *choose* selects a version that complies with the consistency criterion's versioning rules. We shall detail shortly how.

Action *execute*(WRITE, T_i, x) describes the processing of a write request by T_i at the coordinator *coord*(T_i). The middleware buffers the update value in $ws(T_i)$ (line 22).

When the execution reaches the end of the transaction, action *execute*(COMMIT, T_i) submits T_i to the termination protocol line 26). The execution algorithm then waits until T_i either commits or aborts, and returns the outcome.

When the termination protocol commits a transaction, it stores its modifications in the datastore. Depending on the protocol realized, one or more versions of an object may exist simultaneously in *ds*. The realization point *choose* (line 29 in Algorithm 3) abstracts which version is selected when the replica resolves a read request. G-DUR provides a convenient and generic support for tracking and choosing versions. We detail it in the next sections.

6.2.1 Version Tracking

G-DUR supports a generic mechanism for numbering, and selecting different versions of objects. The choice of a version depends on the *versioning mechanism* of a datastore. Recall that when a transaction T_i writes to object x, we say that it creates *version* x_i. Given some history h, we abstract a versioning mechanism Θ as a mapping that associates to each version $x_i \in h$, a *version number* $\Theta(x_i)$ taken from some partially ordered set $(\mathcal{V}, <)$.

Several versioning mechanisms have been proposed in the past. Their implementations often rely on timestamps (TS) [80], vector clocks (VC) [94] or version vectors (VV) [8, 99]. More recently, Sovran et al. [136] and Sciascia and Pedone [129, Section E] discovered independently the concept of vector timestamps (VTS) that allows the computation of partially consistent snapshots at the cost of communicating in the background with all replicas. The GMU vectors (GMV) of Peluso et al. [106] do not have this drawback, but they do not guarantee the monotonicity of snapshots. In Chapter 5, we also introduced Partitionable dependence vectors (PDV) that offers a mechanism close to GMV. In the current state of the implementation, G-DUR supports the VV, VTS, GMV and PDV versioning mechanisms, but more can be added.

Workload contention, liveness of read-only transactions and storage cost, all influence the choice of a versioning mechanism. For instance, some particular DUR protocol may use a central sequencer that assigns timestamps for its simplicity even though the sequencer can become a potential bottleneck. The dimension of $\mathcal{V}$ usually varies from one to the size of the data set or the number of storage nodes. Recently, Peluso et al. [107] prove an $\Omega(min(m,n))$ lower bound on the dimension of $\mathcal{V}$ with $n = |\Pi|$ when the datastore is strictly disjoint access parallel.[1]

[1] We recall that a datastore is strictly disjoint access parallel when two non-conflicting transactions never contend on the same base object, that is on any object in use at the implementation level [64]. This definition generalizes the concept of genuine partial replication.

6.2.2 Picking a Version

The realization of function *choose* by a DUR protocol fits in two the following categories:

(i) $choose_{last}$: it returns the latest version of the object in the sense of $<$.

(ii) $choose_{cons}$: it returns a version consistent with the previous reads.

The first mechanism is straightforward, but requires to abort transactions that did not read a consistent snapshot. In the second case, G-DUR abstracts the dynamic construction of consistent snapshots, on the course of an execution, with a *version compatibility test*. This test takes as input two version numbers $\Theta(x_i)$ and $\Theta(y_j)$, and it outputs *true* iff $\{x_i, y_j\}$ forms a consistent snapshot according to the versioning mechanism Θ. Upon executing a read request from transaction T_i on some object x, $choose_{cons}$ returns the latest version of x that is compatible with all the versions read previously by T_i.

6.3 Termination

Algorithm 4 depicts the pseudo-code of the termination protocol in G-DUR. It accesses the same variables as the execution protocol, along with a FIFO queue named $\mathcal{Q}$ and a variable called $\mathcal{AC}$. This latter variable specifies a particular atomic commitment algorithm; we shall detail its role shortly. When a transaction T_i is submitted for termination, the coordinator first computes the set of objects required to *certify* T_i. Depending on the protocol that is realized, *certifying_obj*(T_i) at line 13 returns one of the following sets of objects;

- $\emptyset$: an empty set allows a transaction T_i to commit without synchronization in the protocol under construction. A typical use case is to ensure that a read-only transaction is wait-free.
- $ws(T_i)$: in this case, the objects modified by transaction T_i are certified.
- $ws(T_i) \cup rs(T_i)$: both the readset and the writeset of T_i are involved in certification. Protocols ensuring serializability (or above) usually return this value.
- *Objects*: this last case represents a scenario where all replicas must participate in certification.

In the case where *certifying_obj*(T_i) returns an empty set, T_i commits locally (line 14). Otherwise, T_i is sent to all replicas *concerned* by T_i: holding some objects in the set *certifying_obj*(T_i).

This is up to the atomic commitment of the protocol under construction to choose an appropriate primitive for *xcast* (line 17). For example, some protocols [103, 131] employ atomic broadcast, while others [117, 127] use atomic multicast (see [44] for complete specifications, and explanations). When T_i is delivered for termination (action *xdeliver*(T_i)), T_i is added to queue $\mathcal{Q}$. The atomic commitment algorithm (variable $\mathcal{AC}$) then *decides* upon T_i, and eventually commits or abort its.

Once the atomic commitment algorithm has taken its decision regarding T_i, the transaction is flagged either COMMIT or ABORT. If it is COMMIT, the transaction is removed from $\mathcal{Q}$, added to *committed*, and its updates are applied to the local datastore. Otherwise, it is added to *aborted*,

Algorithm 4 Termination protocol - code at process p

```
1: Variables:
2:     ds, committed, aborted, executing, submitted,
   Q, AC
3:
4: initialize()
5:     eff:  if AC = GC then start Algorithm 5
6:           else if AC = 2PC then
7:                start Algorithm 6
8:
9: submit(T_i)
10:    pre:  T_i ∈ executing
11:    eff:  executing ← executing \ {T_i}
12:          submitted ← submitted ∪ {T_i}
13:          obj ← certifying_obj(T_i)
14:          if obj = ∅ then
15:              committed ← committed ∪ {T_i}
16:          else
17:              xcast(TERM, T_i) to replicas(obj)
18:
19: xdeliver(T_i)
20:    pre:  received ⟨TERM, T_i⟩
21:    eff:  Q ← Q ∘ T_i
22:
23: commit(T_i)
24:    pre:  decide(T_i) = COMMIT
25:    eff:  Q ← Q \ {T_i}
26:          committed ← committed ∪ {T_i}
27:          ds ← ds ∪ {x_i ∈ ws(T_i) : x_0 ∈ ds}
28:          post_commit(T_i)
29:
30: abort(T_i)
31:    pre:  decide(T_i) = ABORT
32:    eff:  Q ← Q \ {T_i}
33:          aborted ← aborted ∪ {T_i}
34:          post_abort(T_i)
35:
```

and its updates are discarded. In both cases, and once a transaction is terminated, an event (*post_commit*() or *post_abort*()) is triggered. These events can be used to perform operations outside the critical path (e.g., garbage collection).

G-DUR contains several atomic commitment plugins, and supports all the three approaches that we explained in Chapter 2 (i.e., 2PC, total ordering using atomic broadcast, and partial ordering using atomic multicast). In the remainder of this section, we explain in details these plugins.

6.3.1 Group Communication

Figure 6.2 presents an overview of atomic commitment with group communication (GC). Conceptually, this approach is divided into the following steps: Transaction T_i is first sent to a set of voting replicas, and are delivered in total or partial order. These replicas certify T_i, then send the results of their certification tests to another group of replicas. The latter contains at least the coordinator and the replicas of $ws(T_i)$, but it might be larger in some cases. After receiving enough votes, these processes decide locally upon the outcome of T_i.

Algorithm 5 details the internals of the approach. This realization requires that function *xcast* ensures a partial order property over the set of submitted conflicting transactions. Hence, depending on the protocol under construction, *xcast* can be replaced with atomic broadcast or multicast. A transaction T_i is certified at a replica once it is added to $\mathcal{Q}$, and it commutes with all

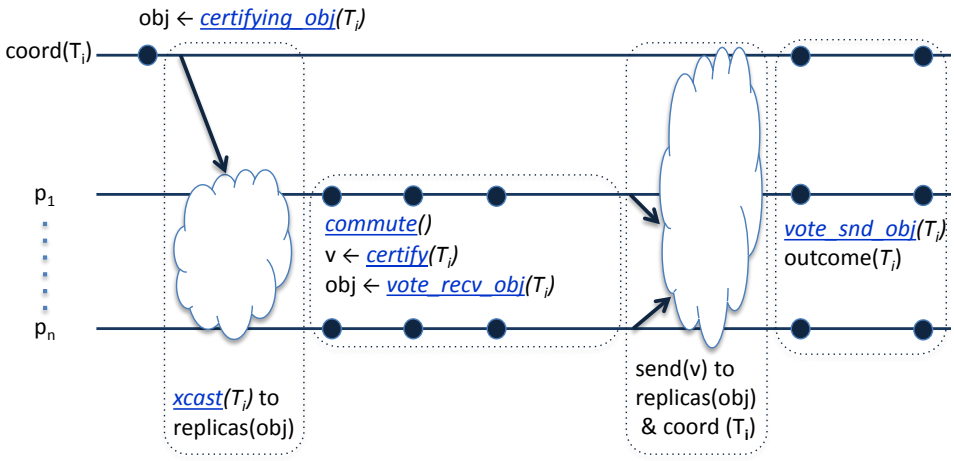


Figure 6.2: Timeline of Atomic Commitment with Group Communication

Algorithm 5 Atomic Commitment with GC - code at p

```
1: vote(T_i)
2:     pre: T_i ∈ Ω
3:          ∀T_j ≤_Ω T_i : commute(T_i, T_j)
4:     eff: v ← certify(T_i)
5:          send ⟨VOTE, T_i, v⟩ to
6:               replicas(vote_recv_obj(T_i)) ∪
7:               {coord(T_i)}
8:
9: decide(T_i)
10:    pre: outcome(T_i) ≠ ⊥
11:    eff: if outcome(T_i) then
12:             wait until T_i = head(Ω)
13:             return COMMIT
14:         else
15:             return ABORT
16:
```

the transactions that precede it in Ω (lines 2 to 3), Commutativity is crucial to minimize convoy effect during certification [28, 127], that is when the certification of one transaction slows down the certification of another one. The definition of commutativity is a function of the consistency model implemented by the realized protocol.

Once a replica certifies T_i locally (line 4), it sends the result of its vote to the coordinator and to the processes in $replicas(vote\_recv\_obj(T_i))$. In most cases, $vote\_recv\_obj(T_i)$ equals $ws(T_i)$; i.e., the set of objects updated by the transaction. However, in some protocols, all replicas receive the certification votes.

A process can safely decide upon the outcome of T_i once it has received votes from a *voting quorum* for T_i. A voting quorum Q for T_i is a set of replicas such that for every object $x \in vote\_snd\_obj()$, the set Q contains at least one replica of x. Formally:

$$vquorum(T_i) \triangleq \{Q \subseteq \Pi \mid \forall x \in vote\_snd\_obj() : \exists j \in Q \cap replicas(x)\}$$

A process p uses the following (three-values) predicate $outcome(T_i)$ to determine whether some transaction T_i commits, or not:

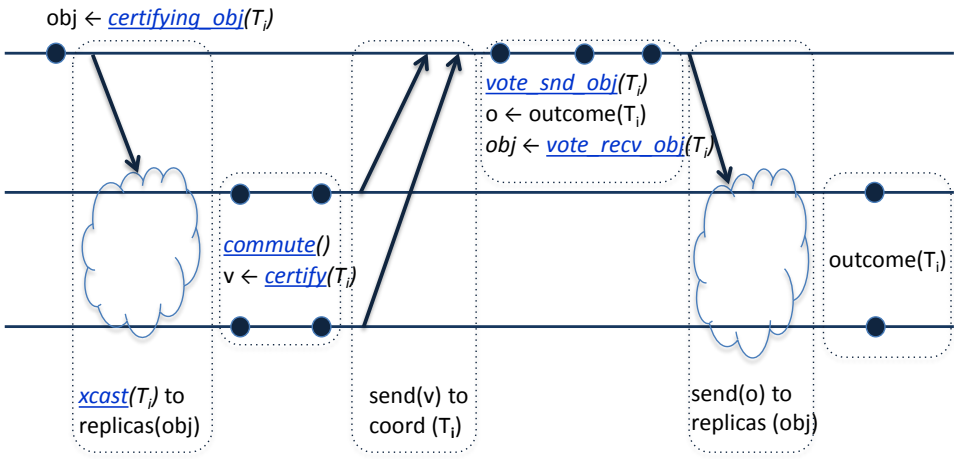


Figure 6.3: Timeline of Atomic Commitment with Two-phase Commit

$$
\begin{aligned}
&\mathit{outcome}(T_i) \triangleq \\
&\qquad \textbf{if } \mathit{vote\_snd\_obj}(T_i) = \varnothing \\
&\qquad\quad \textbf{then } \mathit{true} \\
&\qquad \textbf{else if } \forall Q \in \mathit{vquorum}(T_i), \exists q \in Q, \\
&\qquad\quad \neg \mathit{received}\ \langle \text{VOTE}, T, \_ \rangle\ \mathit{from}\ q\ \textbf{then}\ \bot \\
&\qquad \textbf{else if } \exists Q \in \mathit{vquorum}(T_i), \forall q \in Q, \\
&\qquad\quad \mathit{received}\ \langle \text{VOTE}, T, \mathit{true} \rangle\ \mathit{from}\ q\ \textbf{then}\ \mathit{true} \\
&\qquad \textbf{else}\ \mathit{false}
\end{aligned}
$$

Once the result of $\mathit{outcome}(T_i)$ equals *true*, and T_i is at the beginning of Ω, T_i is flagged COMMIT (lines 11 to 13). Otherwise, if $\mathit{outcome}(T_i)$ equals *false*, it is flagged ABORT (line 15). Algorithm 5 waits that T_i reaches the head of the queue before committing it to ensure replicas apply updates in the same order. This property is mandatory for consistency criteria equal or stronger than Serializability [128]. For weaker consistency models (e.g., Read Committed), we can suppress this constraint. However, in our experience, such a modification has a small impact on performance.

6.3.2 Two-Phase Commit

Figure 6.3 plots the timeline of termination with two-phase commit (2PC) for some transaction T_i. The core difference between GC and 2PC is in the way 2PC uses the coordinator. Whereas with GC, all processes receive the certification votes and decide locally to whether commit or abort the transaction, in 2PC, the coordinator receives all the votes, decides on the outcome of the transaction, and notifies other participants about its decision.

We give more details in Algorithm 6. Our 2PC plugin overrides function *xcast* with a multicast primitive. When a replica delivers a transaction T_i, it aborts T_i in case a concurrent conflicting

Algorithm 6 Atomic Commitment with 2PC - code at p

```
1: vote(T_i)
2:     pre: T_i ∈ 𝒬
3:     eff: if ∃T_j ∈ 𝒬 : ¬commute(T_i, T_j) then
4:              send ⟨VOTE, T_i, false⟩ to coord(T_i)
5:          else
6:              v ← certify(T_i)
7:              send ⟨VOTE, T_i, v⟩ to coord(T_i)
8:
9: vote_coordinator(T_i)
10:    pre: p = coord(T_i)
11:    eff: send ⟨VOTE, T_i, outcome(T_i)⟩ to
12:             vote_recv_obj(T_i)
13:
14: decide(T_i)
15:    pre: outcome(T_i) ≠ ⊥
16:    eff: if outcome(T_i) then return COMMIT
17:         else return ABORT
18:
```

transactions precedes it in $\mathcal{Q}$. Otherwise, the transaction is certified. In both cases, the outcome is sent to $coord(T_i)$, which will decide upon the outcome; other replicas only receive the final vote from $coord(T_i)$. We reflect this by modifying the definition of a voting quorum. More precisely:

$$\begin{aligned} vquorum(T_i)\ at\ coord(T_i) &\triangleq \{Q \subseteq \Pi \mid \forall x \in vote\_snd\_obj() : replicas(x) \subseteq Q\} \\ vquorum(T_i)\ at\ other\ replicas &\triangleq \{coord(T_i)\} \end{aligned}$$

6.3.3 Fault-Tolerance

The approach based on two-phase commit works either when perfect failure detectors are available [21], or in a crash-recovery model. In the first case, the coordinator preemptively aborts the transaction when a replicas fails in the middle of the termination phase. In the later, *(i)* every time the state of Algorithm 6 changes, the modification must be logged, and *(ii)* when a replica crashes, Algorithm 6 has to wait that it comes back online to pursue the execution.

A commitment protocol based on group communication can cope more easily with failures if it internally relies on a dependable consensus protocol. In more details, if Algorithm 5 employs atomic broadcast to order transactions, it needs inaccurate failure detection and tolerates up to $f < n/2$ replica crashes, where n is the total number of replicas. Now, if only replicas concerned by the transaction make steps to commit it, Algorithm 5 should use a genuine atomic multicast primitive [66]. In the general case, this requires perfect failure detection [122]. Nevertheless, with an appropriate replicas placement, e.g., in a geo-replicated scenario, inaccurate failure detectors can implement a non disaster-tolerant atomic multicast (see Schiper et al. [126] for more detail).

The difference in terms of dependability between the two approaches translates into a difference in time and message complexity. Let us note r the average cardinality of $replicas(ws(T) \cup rs(T))$. Algorithm 6 requires $\Omega(r)$ messages and its message delay is 2. On the other hand, an optimal atomic broadcast protocol [47, 82] costs 3 message delay with $\Omega(n)$ messages, and the best genuine

Algorithm 7 P-Store [127]

1: $\Theta \triangleq TS$
2: $choose \triangleq choose_{last}$
3: $\mathcal{AC} \triangleq$ GC
4: $xcast \triangleq$ AM-Cast
5: $certifying\_obj(T_i) \triangleq ws(T_i) \cup rs(T_i)$
6: $commute(T_i, T_j) \triangleq rs(T_i) \cap ws(T_j) = \varnothing \wedge\ rs(T_j) \cap ws(T_i) = \varnothing$
7: $certify(T_i) \triangleq \forall x_i, x_j \in rs(T_i) \times db : \Theta(x_j) \leq \Theta(x_i)$

fault-tolerant atomic multicast known to date needs 6 message delays with $\Omega(r^2)$ messages [122]. This makes fault-tolerance expensive at first glance. In Section 6.6.5, we further investigates this cost in the context of geo-replicated data.

6.4 Realizing Protocols

This section illustrates how to implement a protocol in the G-DUR middleware. We consider five different consistency criteria: Serializability, Snapshot Isolation, Update Serializability, Parallel Snapshot Isolation and Non-Monotonic Snapshot Isolation. For each criterion, we pick at least one state-of-the-art protocol, and explain how to implement it with G-DUR. As we shall see, we can express the core aspects of a protocol in fewer than 10 lines of pseudo-code.

In all the implementations we cover next, unless specified otherwise, *vote_recv_obj*(T_i) returns $ws(T_i)$, and the realization of *vote_snd_obj*() is the same as the realization of *certifying_obj*().

6.4.1 P-Store

Algorithm 7 depicts our realization of P-Store [127]. This protocol relies on a timestamping mechanism to version objects (line 1). Every read operation retrieves asynchronously the latest version of the corresponding object (line 2). A transaction commits iff no new versions of the objects it read were created concurrently (line 7); such a certification test is typical of DUR protocols that ensure SER. An alternative classical approach [138] is to rely on cycle detection in the serializability graph; however, as pointed by Guerraoui et al. [67], this is expensive.

6.4.2 S-DUR

Unlike P-Store, S-DUR [129] (Algorithm 8) ensures that every read operates on a consistent snapshot (line 2). This implies that queries are wait-free (line 5). Upon the termination of an update transaction T_i, S-DUR atomic multicasts T_i to the replicas holding an object in $ws(T_i) \cup rs(T_i)$. In that case, however, the group communication primitive only ensures a pairwise ordering of the transactions [44], i.e., two processes only deliver transactions that they have in common in the same order (line 4). This design tends to increase the scalability of the multicast

Algorithm 8 S-DUR [129]

1: $\Theta \triangleq VTS$
2: $choose \triangleq choose_{cons}$
3: $\mathcal{AC} \triangleq$ GC
4: $xcast \triangleq$ AM$_{pw}$-Cast
5: $certifying\_obj(T_i) \triangleq$ **if** $|ws(T_i)| = 0$ **then** $\varnothing$
else $ws(T_i) \cup rs(T_i)$
6: $commute(T_i, T_j) \triangleq rs(T_i) \cap ws(T_j) = \varnothing \wedge\ rs(T_j) \cap ws(T_i) = \varnothing$
7: $certify(T_i) \triangleq \forall T_j \parallel T_i \in committed :\ ws(T_i) \cap rs(T_j) = \varnothing \wedge rs(T_i) \cap ws(T_j) = \varnothing$
8: $vote\_recv\_obj(T_i) \triangleq ws(T_i)$
9: $post\_commit(T_i) \triangleq$ M-Cast($\Theta(T_i)$) to $(\Pi \setminus replicas(certifying\_obj(T_i)))$

Algorithm 9 GMU [106]

1: $\Theta \triangleq GMV$
2: $choose \triangleq choose_{cons}$
3: $\mathcal{AC} \triangleq$ 2PC
4: $certifying\_obj(T_i) \triangleq$ **if** $|ws(T_i)| = 0$ **then** $\varnothing$
else $rs(T_i) \cup ws(T_i)$
5: $commute(T_i, T_j) \triangleq rs(T_i) \cap ws(T_j) = \varnothing \wedge\ rs(T_j) \cap ws(T_i) = \varnothing$
6: $certify(T_i) \triangleq \forall x_i, x_j \in rs(T_i) \times db : \Theta(x_j) \leq \Theta(x_i)$

primitive, but comes at the price of the following drawbacks: *(i)* More aborts because an update transaction commits only if there is no concurrent conflicting committed transaction (line 7), and *(ii)* As we showed in Chapter 4, no GPR system under SER can ensure WFQ. Thus, S-DUR needs to perform some background propagation to all replicas (line 9) in order to ensure NTU. This propagation consists of sending $\Theta(T_i) = max\ \{\Theta(x_i) : x_i \in rs(T_i) \cup ws(T_j)\}$ to all replicas in order to advance the vector clock maintained at each replica.

6.4.3 GMU

The GMU transactional system of Peluso et al. [106] (see Algorithm 9) ensures EUS. Hence, it commits queries locally (line 4), and provides WFQ. In the case of update transactions, GMU makes use of 2PC (line 3). All replicas holding an object read or written by the transaction participate in the 2PC. We note here that the certification test of GMU (line 6) is similar to P-Store, but unlike P-Store, it ensures both GPR and WFQ. Moreover, since GMU ensures EUS, non-monotonic snapshots are observable for read-only transactions.

6.4.4 Serrano07

Serrano et al. [131] introduce a non-genuine partial replication protocol under SI. We depict its pseudo-code in Algorithm 10. Queries commit locally and update transactions are atomic-

Algorithm 10 Serrano07 [131]

1: $choose \triangleq choose_{cons}$
2: $\Theta \triangleq TS$
3: $\mathcal{AC} \triangleq$ GC
4: $xcast \triangleq$ AB-Cast
5: $certifying\_obj(T_i) \triangleq$ **if** $|ws(T_i)| = 0$ **then** $\varnothing$
else $Objects$
6: $commute(T_i, T_j) \triangleq ws(T_i) \cap ws(T_j)$
7: $certify(T_i) \triangleq \forall x_i, x_j \in ws(T_i) \times db : \Theta(x_j) \leq \Theta(x_i)$
8: $vote\_snd\_obj(T_i) = vote\_recv\_obj(T_i) \triangleq LocalObjects$

Algorithm 11 Walter [136]

1: $choose \triangleq choose_{cons}$
2: $\Theta \triangleq VTS$
3: $\mathcal{AC} \triangleq$ 2PC
4: $certifying\_obj(T_i) \triangleq ws(T_i)$
5: $commute(T_i, T_j) \triangleq ws(T_i) \cap ws(T_j) = \varnothing$
6: $certify(T_i) \triangleq \forall x_i, x_j \in ws(T_i) \times db : \Theta(x_j) \leq \Theta(x_i)$
7: $post\_commit(T_i) \triangleq$ M-Cast$(\Theta(T_i))$ to $(\Pi \setminus replicas(certifying\_obj(T_i)))$

broadcast to all replicas (lines 4 and 5). When a replica delivers an update transaction, it performs a certification test to check for concurrent updates (line 7). This protocol need to maintain the latest version number of every object at each replica. Hence, every replica decides independently on the outcome of a transaction, and a distributed voting phase is not required. We capture this behavior by the fact that both *vote_snd_obj*() and *vote_recv_obj*() equal the local objects (line 8).

6.4.5 Walter

Walter (shown in Algorithm 11) is the transactional system proposed by Sovran et al. [136] to implement PSI. This protocol relies on a two-phase commit among the replicas that hold an object written by the transaction (lines 3 and 4). To satisfy PSI, the certification of Walter ensures that no two concurrent write-conflicting transactions both commit (line 6). Once a transaction is committed, Walter propagates $\Theta(T_i)$ in the background to all the replicas in the system (line 7). As in the case of S-DUR, this background propagation is crucial to ensure progress.

6.4.6 Jessy$_{2pc}$

The Jessy protocol presented in the previous chapter guarantees NMSI. In this chapter, we shall be considering a 2PC-based variation of Jessy. This protocol, denoted Jessy$_{2pc}$, is presented in Algorithm 12. Jessy$_{2pc}$ relies on the PDV versioning mechanism to compute consistent snapshots (line 2), and it uses two-phase commit during termination (line 3). Like Serrano and Walter,

Algorithm 12 Jessy$_{2pc}$

1: $choose \triangleq choose_{cons}$
2: $\Theta \triangleq PDV$
3: $\mathcal{AC} \triangleq$ 2PC
4: $certifying\_obj(T_i) \triangleq ws(T_i)$
5: $commute(T_i, T_j) \triangleq ws(T_i) \cap ws(T_j) = \varnothing$
6: $certify(T_i) \triangleq \forall x_i, x_j \in ws(T_i) \times db : \Theta(x_j) \leq \Theta(x_i)$

Jessy$_{2pc}$ checks for concurrent write-conflicting transactions (line 6) during the certification. Note that, because Jessy$_{2pc}$ is GPR, no background propagation is needed by this protocol after the commitment of a transaction.

6.5 Implementation

We implemented G-DUR, and the realized transactional protocols in Java. Our implementations closely follow the published specification of each protocol. We also implemented a DUR protocol ensuring read-committed consistency criterion (RC). As we explained in the previous chapter, RC is a weak consistency criterion ensuring that a transaction reads a committed version of an object without any additional guarantee. RC plays as a baseline, and shows the maximum achievable performance in our experiments.

G-DUR can work either with a data persistence layer (i.e., BerkeleyDB), or without (i.e., an in-memory concurrent hashmap). To minimize noise, and to focus on scalability and synchronization, our experiments in this paper are done using the latter case. Should the user decide to use the data persistence layer, she can easily implement an interface, and attach any other data store.

The implementation of G-DUR and the six protocols takes approximately 10^4 source lines of code (SLOC). The communication layer also takes an additional 10^4 source lines of code. Table 6.2 details the number of SLOC for each protocol. Observe that the G-DUR implementations take an order of magnitude fewer LOC than the monolithic originals. This code, the benchmarks, as well as the scripts we used in our experiments are publicly available [116].

6.6 Case Study

This section shows some practical uses of G-DUR: *(i)* a comparison of the protocols realized in Section 6.4, *(ii)* an analysis of the bottlenecks of the GMU protocol, *(iii)* a demonstration of the pluggability capabilities of G-DUR, and *(iv)* an assessment of the cost of dependability.

Protocol	Source Lines of Code			
			Total	
	Exec.†	*Term.*†	G-DUR†	Original
P-Store‡	45	134	179	6000
S-DUR	199	288	397	N/A
GMU‡	184	292	476	6000
Serrano	104	247	351	N/A
Walter	322	277	599	30000
Jessy_{2pc} ‡	155	197	352	6000

Table 6.2: Source lines of code

†: excluding comments ‡: open source

Workload	Key Selection Distribution	Operations	
		Read-only Transaction .	Update Transaction
A	Uniform	2 Read	1 Read, 1 Update
B	Uniform	4 Read	2 Read, 2 Update
C	Zipfian	2 Read	1 Read, 1 Update

Table 6.3: Experimental Settings

6.6.1 Setup and Benchmark

As in Chapter 5, all our experiments are run on sites of the French Grid'5000 experimental testbed [62]. Latencies between sites are between 10 to 20 ms (see Figure 5.1.a). We use 4-core machines running between 2.2 and 2.6 GHz with a maximum heap size of 4 GB.

We performed our experiments under different configurations and using various numbers of sites. Although G-DUR can take care of intra-site replication as well, our experiments are considering a single replica per site (similarly to Sovran et al. [136]). For each replica, two additional client machines generate the workload. We perform our experiments in either a *Disaster Prone* (DP), or *Disaster Tolerant* (DT) configuration. In case of DP, every object is stored in a single site, whereas DT replicates every object in two sites.

Every replica contains 10^5 objects, and each object has a payload size of 1 KB. We employ the Yahoo Cloud Serving Benchmark (YCSB) [37], modified to generate transactions. Table 6.3 describes the workloads we use during our experiments, which were already used by previous papers [117, 129]. A client machine emulates multiple client threads in parallel, each running in closed loop. During an experiment, a client machine executes at least 10^6 transactions. Every transactions is (i) *interactive*: none of the objects it accesses is known in advance, and (ii) *global*: no replica holds all the objects read or written by the transaction. We chose this last setting to emphasize the geo-replicated performance of the protocols.

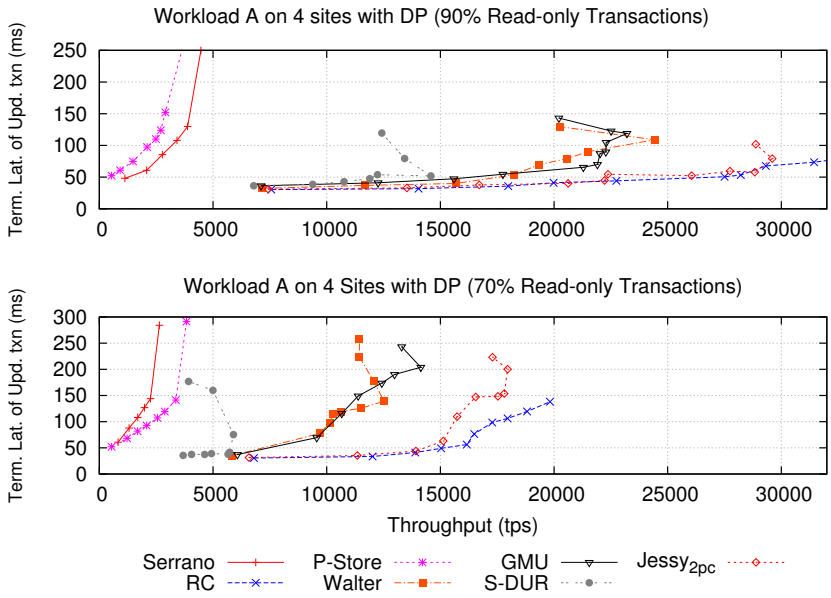


Figure 6.4: Performance Comparison with Disaster Prone Configuration

6.6.2 Comparing Transactional Protocols

This section compare the realizations of Section 6.4 in order to bring out differences between the protocols, as well as between the consistency criteria they implement.

In Figure 6.4, we depict the performance of each protocol under Workload A, when using four sites in a DP configurations. We use either 90% (top) or 70% (bottom) of read-only transactions. Each point plots the *termination latency of update transactions*, that is, the average time between the termination request of an update transaction and the reception of the response by the client, as a function of the throughput. The termination latency of the update transactions is the most meaningful metric to observe differences between the realized protocols since *(i)* all protocols (except P-Store) implement wait-free queries, and *(ii)* they all follow the DUR approach with the same execution phase (except Serrano).

Jessy$_{2pc}$ is the fastest protocol by being a genuine protocol, and requiring minimal synchronization: only replicas holding modified objects are included in transaction certification. Although Walter also enjoys minimal synchronization, being non-genuine results in smaller throughput compared to Jessy$_{2pc}$. In GMU, the replicas of both read and modified objects are involved in the certification. Yet the performance of GMU and Walter are the same with 90% read-only transactions. With 70% of read-only transactions, Walter requires more global propagation (due to its non-genuineness), and starts degrading before GMU.

In P-Store, queries are not wait-free and they have to go through AM-Cast. This design choice greatly impacts performance and explains that the throughput of P-Store is the worst among the studied protocols with 90% of read-only transactions. On the other hand, P-Store compensates

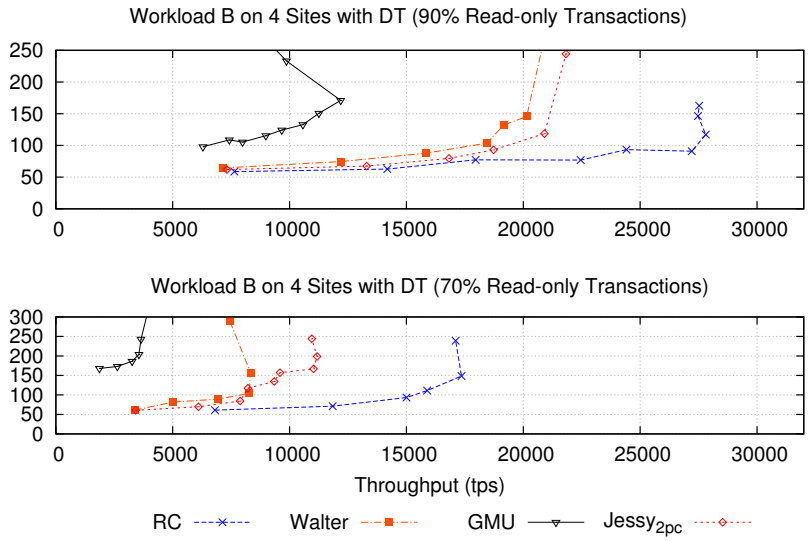


Figure 6.5: Performance Comparison with Disaster Tolerant Configuration

this gap with 70% of read-only transaction, and overtakes Serrano. This observation underlines again the importance of supporting GPR for a transactional system. This also shows that as pointed out by Lin et al. [88], the protocol of Serrano et al. is more oriented for LAN environments.

S-DUR always delivers a better throughput than Serrano. This difference points out that, unlike the general credence and for certain workloads, SER can be faster than SI, provided the protocol implementing SER ensures wait-free queries. Finally, the performance gap between Serrano and Walter clearly motivates the use of PSI over SI in a geo-replicated setting. This assesses empirically the original argument of Sovran et al. [136].

We note that while PSI is weaker than SI [136] and US is weaker than SER [3], neither US and PSI nor SI and SER are mutually comparable (see Section 5.1). Thus, an anomalistic comparison between the two criteria would not explain their performance differences. This is the realization of the protocols inside G-DUR which allows us to fairly compare them in terms of throughput and latency.

To better understand the differences among the protocols ensuring weaker consistency criteria, we evaluate them in Figure 6.5 using Workload B and in a disaster tolerant configuration. Under 90% of read-only transactions, the performance of Walter, and Jessy$_{2pc}$ are similar. This is due to the fact that transactions contain more operations in Workload B, hence the non-genuineness of Walter does not impact performance in comparison to Jessy$_{2pc}$. The performance of GMU also degrades with Workload B. This is mainly due to the abort rate. With 1024 client threads, and 90% read-only transactions, the abort rate of GMU reaches 12% while the abort rates of Walter and Jessy$_{2pc}$ stay below 0.1%. With 30% update transactions, the abort rate of GMU deteriorates

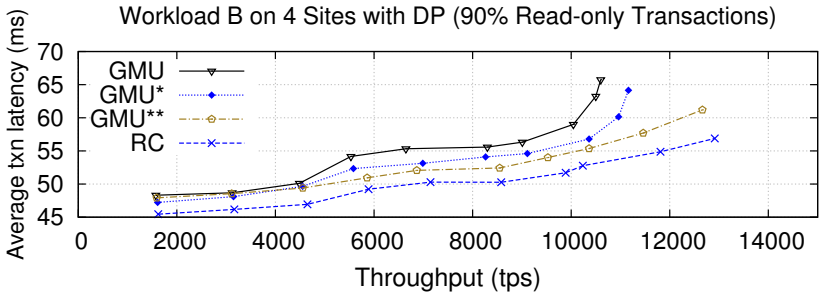


Figure 6.6: Study of Bottlenecks in GMU -

GMU: Consistent Snapshot & Certification, GMU: Trivial Snapshot & Certification, GMU**: Trivial Snapshot & Trivial Certification*

to 48%, and the abort rate of Jessy$_{2pc}$ and Walter reach around 1%.

6.6.3 Understanding Bottlenecks

In this section, we explain how a developer can study the costs of the different components implementing a transactional protocol in order to locate its bottlenecks. Our approach consists in a careful substitution of the versioning and certification plug-ins with trivial ones. We plot our results for the GMU protocol in Figure 6.6.

As depicted in Algorithm 9, GMU takes consistent and fresh snapshots during the execution phase. In GMU*, we turned off this versioning component, replacing it with $choose_{last}$. However, metadata required for taking consistent snapshots is still sent during the execution phase. We observe in Figure 6.6 that both GMU and GMU* follow the same trend, and that the overhead of taking consistent snapshots in GMU is around 5%. With GMU**, we turn off the certification test, and all the transactions now pass the certification test. The resulting protocol follows the trend of RC, while still exhibiting a small performance gap. This difference is explained by the overhead of marshaling and sending metadata related to snapshots that GMU** inherits from the original protocol. Thus, at the light of the results depicted in Figure 6.6, we can conclude that the certification test is the main bottleneck of the algorithm of Peluso et al. [106].

6.6.4 Pluggability Capabilities

G-DUR allows a developer to replace the plug-ins that compose a transactional protocol. Such a feature helps to finely understand a protocol and in turn paves the way to improve it. In this section, we demonstrate its usage with P-Store [127].

In P-Store, read-only transactions have to go through a certification test. Following an analysis similar to the one we conducted in the previous section for GMU, we can show that this mechanism is an important bottleneck of the protocol. In general, as explained in Section 6.4, this

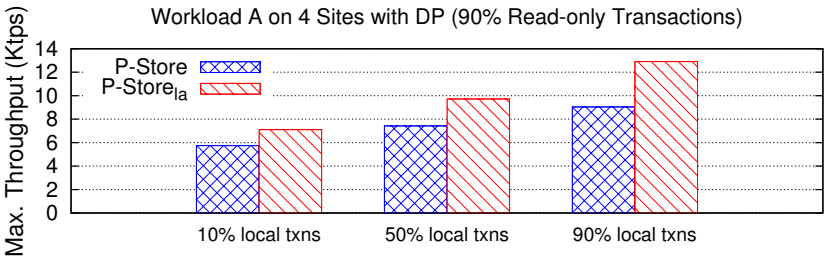


Figure 6.7: Throughput improvement of P-Store

bottleneck cannot be overcome since P-Store is a genuine algorithm. However, P-Store can safely commit a read-only transaction without certifying it when the transaction accesses a single data partition (typically a single site). We implement this feature as follows: (i) Instead of reading the latest committed value during the execution phase, we take a consistent snapshot. We achieve this in G-DUR by using the $choose_{cons}$ component implemented with partitioned dependency vectors (PDV). (ii) We change the realization of $certifying\_obj(T_i)$ such that it returns $\varnothing$ in the case where T_i is a query accessing a single partition. Figure 6.7 plots the throughput of our locality aware P-Store, denoted $P\text{-}Store_{la}$, in comparison to the original algorithm of Schiper et al. [127]. We can observe that, depending on the ratio of local read-only transactions, $P\text{-}Store_{la}$ is 20 to 70% faster than the original protocol.

6.6.5 Dependability

As pointed out in Section 6.3, termination based on group communication primitives orders a priori conflicting transactions, whereas the use of 2PC relies on a spontaneous ordering of the network. The two approaches also differ in terms of fault-tolerance. The former requires either a crash-recovery model or perfect failure detection to ensure liveness, whereas the later can accommodate with faults. In this section, we compare them empirically in our geo-replicated environment.

To this goal, we picked P-Store and changed its atomic commitment protocol from AM-Cast to 2PC. The rationale of this choice is that the versioning mechanism of P-Store has the smallest overhead compared to other protocols, and that this protocol certifies both read-only and updates transactions. Both features reduce noise during our measurements since we limit the amount metadata used by the system, and all the transactions go through the termination phase.

6.6.5.1 Disaster Prone

In a disaster-prone scenario, every object is replicated at a single site. Hence, when a site goes down, the system has to wait that it becomes available again. Figure 6.8 compares the 2PC and AM-Cast variations of P-Store in this configuration. With Workload A, the abort ratio of both

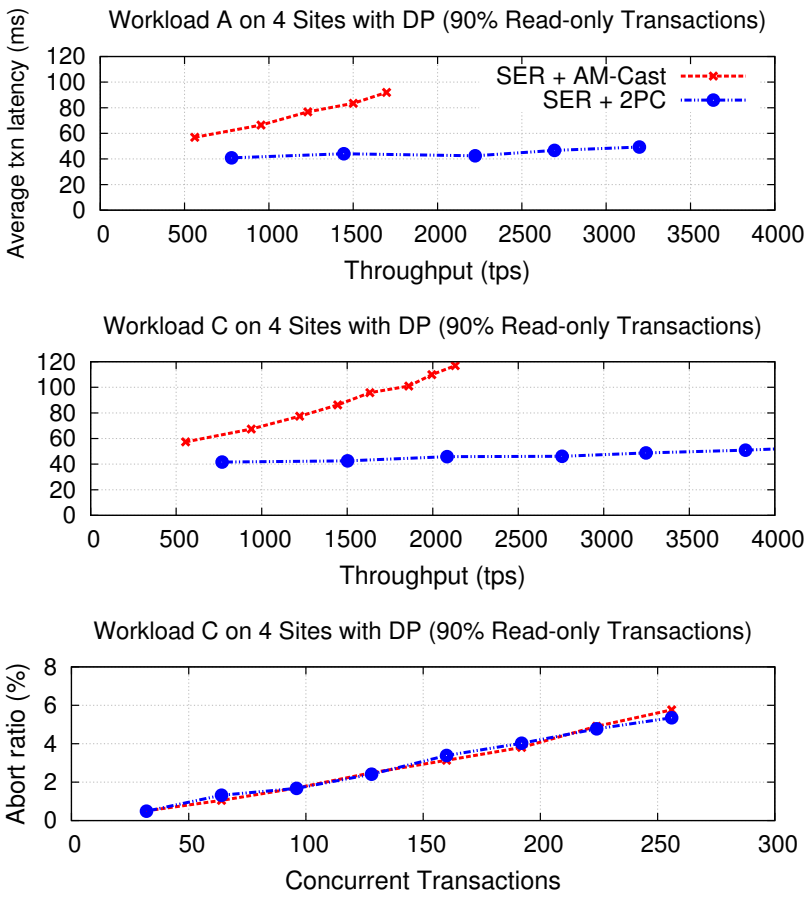


Figure 6.8: 2PC vs. AM-Cast with Disaster Prone Configuration

protocols is almost null and 2PC outperforms AM-Cast by a factor of at least two. Under a highly contended workload (Workload C), the abort ratio of both protocols increases similarly, and 2PC still outperforms AM-Cast. As a consequence, in such a scenario, ordering transactions a priori has a limited positive effect on the abort ratio. and it does not pay off.

6.6.5.2 Disaster Tolerant

In a disaster-tolerant setting, every object is replicated at two sites. Therefore, the system can tolerate a complete site failure. The results of this experiment are shown in Figure 6.9. Like the previous scenario, 2PC still outperforms AM-Cast with Workload A. However, under Workload C, once the sites become saturated, the abort ratio of 2PC increases drastically, due to the preemptive aborts (line 4 in Algorithm 6). Thus, in such a situation, pre-ordering the transactions in the commitment phase pays off.

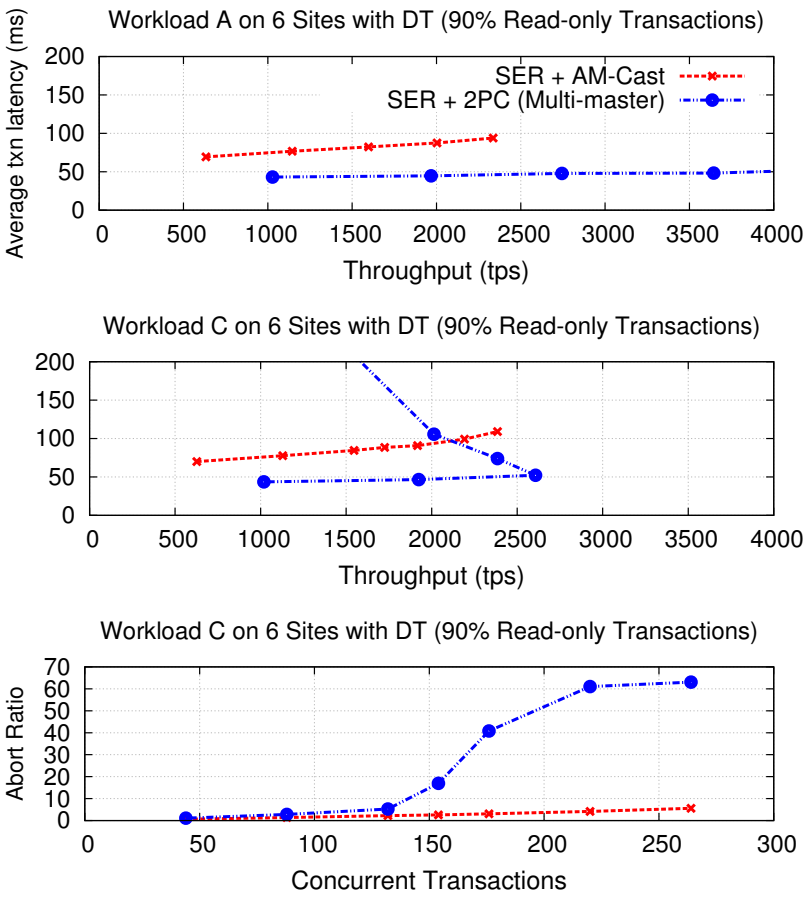


Figure 6.9: 2PC vs. AM-Cast with Disaster Tolerant Configuration

6.7 Related Work

In general, application workloads exhibit a large portion of non-conflicting transactions. Under such an assumption, the interest of the DUR approach, i.e., an optimistic phase followed by a termination phase, was underlined by Alonso [10]. Wiesmann and Schiper [145] compare several replication protocols and confirm that DUR is better than distributed locking and primary copy under full replication. The work of Schmidt and Pedone [128] provides a formal analysis of DUR, focusing on serializability and full replication. The DUR approach is also a de facto standard for software transactional memories (e.g., [25, 52]).

Several works aim at understanding and classifying full replication techniques. Wiesmann et al. [147] provide a classification of different replication mechanisms, for both transactional and non-transactional systems. Subsequently, the authors propose a three parameters classification of transactional protocols [146]. They classify protocols according to the server architecture (primary copy versus update-everywhere), the server interaction (constant versus linear), and the transaction termination (voting versus non-voting). The present work continues their study,

focusing on DUR protocols under *partial replication*. According to their terminology, this means that we shall be interested in passive replication with either update-everywhere or primary-copy, and both voting and linear interaction.

An abundant literature (see [96] for a detailed survey) provides analytical models and simulations of distributed database systems. These works focus on evaluating throughput or latency as a function of workload, replication factor and network characteristics, and provide a useful insight on how transactional protocols behave under low conflict rate. However, by oversimplifying the management of conflicting transactions, they do not give a completely accurate figure. In particular, the impacts of *(i)* versioning mechanism, *(ii)* consistency criteria, *(iii)* convoy effects during certification, and *(iv)* genuineness[2] on a protocol are largely under-evaluated. Our experiments show that these parameters strongly influence the performance of a transactional protocol.

Commercial database products [95, 110] usually allow the client to pick a consistency criterion when executing transactions. Several researchers have studied, and compared multiple criteria. Berenson et al. [22] show that the phenomena-based definition of ANSI SQL-92 does not properly characterize the differences between SI and SER. They propose new anomalies and compare most well-known criteria based on this characterization. Adya et al. in [3, 4] present the first implementation-independent specification of ANSI levels. This specification is not limited to pessimistic implementations, that is based on an *a priori* ordering of the transactions (e.g., [101], or more recently [39]), but is also applicable to optimistic and multi-version schemes. To achieve this, the authors define different read/write conflicts, and specify different graphs based on these conflicts. Subsequently, most of the well-known criteria are formally defined, and compared. The above specifications are yet hard to understand, especially in the context of distributed transactional systems.

A few works study and compare different family of transactional protocols. Bernabé-Gisbert et al. [23] introduce a middleware that ensures different levels of consistency. They use an update everywhere approach, and certify transactions with the help of atomic broadcast. Each transaction can declare separately a consistency criterion of its choice. It is then the job of the underlying database to certify transactions, and to execute the concurrency control. No comparison or evaluation is given in this paper. Kemme and Alonso [77] introduce a family of eager replication protocols using group communication primitives. They compare various consistency criteria (SER, SI and Cursor Stability) with simulations, but do not provide a unified protocol.

[2] A protocol is genuine when only the replicas of the objects accessed by a transaction make computational steps to execute it.

6.8 Conclusion

Deferred update replication (DUR) is a classical technique to construct transactional datastores. Protocols that follow the DUR approach share a common algorithmic structure consisting in a speculative execution phase followed by a termination phase, and at core, they only differ by instantiating a few generic functions in each phase. This chapter presents G-DUR, a generic deferred update replication middleware built upon this insight. G-DUR brings several benefits to practitioners and researchers in the field of transactional storage:

- It allows to easily fast prototype a transactional protocol following the DUR approach. In Section 6.4, we presented the implementation of six state-of-the-art replication protocols published in the past few years [106, 117, 127, 129, 131, 136]. Each protocol in our middleware requires less than 600 lines of code.
- G-DUR fosters apples-to-apples comparison of transactional protocols. We illustrated this in Section 6.6.2 by presenting an empirical evaluation in a geo-replicated environment. To the best of our knowledge, such a fair comparison never appears elsewhere in literature. The key reason is that it is either hard (or impossible) to be performed with the original implementations as source codes are generally not comparable, nor always publicly available. In addition, mastering each protocol requires a large amount of time.
- With G-DUR, a developer can study in details the limitations and overheads of her protocol. In Section 6.6.3, we illustrated this point with the protocols of Peluso et al. [106]. Then, we presented in Section 6.6.4 a variation of P-Store [127] that leverages workload locality. Our variation performs up to 70% faster than the original protocol.
- Finally, G-DUR allows us to study the cost of various degrees of dependability. In Section 6.6.5, we evaluated in practice the difference between commitment based on group communication primitives and 2PC in a disaster-prone and a disaster-tolerant setting.

Part II: Ensuring Consistency in Non-Transactional Data Stores

CHAPTER 7

TUBA: A SELF-CONFIGURABLE CLOUD STORAGE SYSTEM

Contents

In Part I, we focused on ensuring consistency criteria in transactional data stores. In the second part, we shift our attention, and study the problem of ensuring consistency criteria for data stores with read and write operations. In particular, we investigate how to automatically reconfigure a storage system while maintaining consistency guarantees, and respecting user defined constraints so that it adapts to changes in users locations or request rates.

7.1 Introduction

Cloud storage systems can meet the demanding needs of their applications by dynamically selecting when and where data is replicated. An emerging model is to utilize a mix of strongly consistent primary replicas and eventually consistent secondary replicas. Applications either explicitly choose which replicas to access or let the storage system select replicas at run-time based on an application's consistency and performance requirements [140]. In either case, the configuration of the system significantly impacts the delivered level of service.

Configuration issues that must be addressed by cloud storage systems include: (i) where to put primary and secondary replicas, (ii) how many secondary replicas to deploy, and (iii) how frequently secondary replicas should synchronize with the primary replica. These choices are complicated by the fact that Internet users are located in different geographical locations with different time zones and access patterns. Moreover, systems must consider the growing legal, security, and cost constraints about replicating data in certain countries or avoiding replication in others.

For a stable user community, static configuration choices made by a system administrator may be acceptable. But many modern applications, like shopping, social networking, news, and gaming, not only have evolving world-wide users but also observe time-varying access patterns, either on a daily or seasonal basis. Thus, it is advantageous for the storage system to *automatically* adapt its configuration subject to application-specific and geo-political constraints.

Tuba is a geo-replicated key-value store based on Pileus [140]. It addresses the above challenges by configuring its replicas automatically and periodically. While clients try to maximize the utility of individual read operations, Tuba improves the overall utility of the storage system by automatically adapting to changes in access patterns and constraints. To this end, Tuba includes a configuration service that periodically receives from clients their consistency-based service level agreements (SLAs) along with their hit and miss ratios. This service then changes the locations of primary and secondary replicas to improve the overall delivered utility. A key property of Tuba is that both read and write operations can be executed in parallel with reconfiguration operations.

We have implemented Tuba as middleware on top of Microsoft Azure Storage (MAS) [30]. It extends MAS with broad consistency choices as in Bayou [139], and provides consistency-based SLAs like Pileus. Moreover, it leverages geo-replication for increased locality and availability. Our

API is a minor extension to the MAS Blob Store API, thereby allowing existing Azure applications to use Tuba with little effort while experiencing the benefits of dynamic reconfiguration.

An experiment with clients distributed in datacenters (sites) around the world shows that reconfiguration every two hours increases the fraction of reads guaranteeing strong consistency from 33% to 54%. This confirms that automatic reconfiguration can yield substantial benefits which are realizable in practice.

The outline of this chapter is as follows. We review Pileus and Tuba in Section 7.2. We look under the hood of Tuba's configuration service in Section 7.3. Section 7.4 describes execution modes of clients in Tuba. In Section 7.5, we explain implementation details of the system. Our evaluation results are presented in Section 7.6. We review related work in Section 7.7 and conclude the chapter in Section 7.8.

7.2 System Overview

In this section, we first briefly explain features that Tuba inherits from Pileus. Since we do not cover all technical issues of Pileus, we encourage readers to read the original paper [140] for more detail. Then, we overview Tuba and its fundamental components, and how it extends the features of the Pileus system.

7.2.1 Tuba Features from Pileus

Storage systems cannot always provide rapid access to strongly consistent data because of the high network latency between geographical sites and diverse operational conditions. Clients are forced to select less ideal consistency/latency combinations in many cases. Pileus addresses this problem by allowing clients to declare their consistency and latency priorities via SLAs. Each SLA comprises several subSLAs, and each subSLA contains a desired consistency, latency and utility.

The utility of a subSLA indicates the value of the associated consistency/latency combination to the application and its users. Inside a SLA, higher-ranked subSLAs have higher utility than lower-ranked subSLAs. For example, consider the SLA shown in Figure 7.1. Read operations with strong consistency are assigned utility 1 as long as they complete in less than 50 ms. Otherwise, the application tolerates eventually consistent data and longer response times though the rewarded utility is very small (0.01). Pileus, when performing a read operation with a given SLA, attempts to maximize the delivered utility by meeting the highest-ranked subSLA possible.

The replication scheme in Pileus resembles that of other cloud storage systems. Like BigTable [34], each key-value store is horizontally partitioned by key-ranges into *tablets*, which serve as the granularity of replication. Tablets are replicated at an arbitrary collection of storage sites. Storage sites are either primary or secondary. All write operations are performed at the primary sites. Secondary sites periodically synchronize with the primary sites in order to receive updates.

Rank	Consistency	Latency(ms)	Utility
1	Strong	50	1
2	Eventual	1000	0.01

Figure 7.1: SLA Example

Depending on the desired consistency and latency as specified in an SLA, the network delays between clients and various replication sites, and the synchronization period between primary and secondary sites, the Pileus client library decides on the site to which a read operation is issued. Pileus provides six consistency choices that can be included in SLAs:

- Strong: a read operation on an object returns the value of the last preceding write operation that is performed on the object by any client.
- Eventual: a read operation on an object returns the value of some write operation performed on the object.
- Read-my-writes (RMW): a read operation on an object returns the value of the last preceding write operation that is performed on the object by the same client, or it returns some later versions.
- Monotonic reads: a read operation on an object returns the same or later versions of the object compared to the previous read operation issued by the same client.
- Bounded(t): a read operation on an object returns a value that is stale by at most t seconds.
- Causal: a read on an object returns a value of the latest write operation that causally precedes it, or it returns some later versions.

Consider again the SLA shown in Figure 7.1. A Pileus client reads the most recent data and *hits* the first subSLA as long as the round trip latency between that client and a primary site is less than 50ms. But, the first subSLA *misses* for clients with a round trip latency of more than 50ms to primary sites. For these clients, Pileus reads data from any replica site and hits the second subSLA.

Pileus helps developers find a suitable consistency/latency combination given a fixed configuration of tablets. Specifically, the locations of primary and secondary replication sites, the number of required secondary sites, and the synchronization period between secondary and primary sites need to be specified by system administrators manually. However, a world-wide distribution of users makes it extremely hard to find an optimal configuration where the overall utility of the system is maximized with a minimum cost. Tuba extends Pileus to specifically address this issue.

7.2.2 Tuba's New Features

The main goal of Tuba is to periodically improve the overall utility of the system while respecting replication and cost constraints. To this end, it extends Pileus with a configuration service (CS) delivering the following capabilities:

1. performing a reconfiguration periodically for different tablets, and

2. informing clients of the current configuration for different tablets.

We note that the above capabilities do not necessarily need to be collocated at the same service. Yet, we assume they are provided by the same service for the sake of simplicity.

In order for the CS to configure a tablet's replicas such that the overall utility increases, it must be aware of the way the tablet is being accessed globally. Therefore, all clients in the system periodically send their *observed latency* and the *hit and miss ratios* of their SLAs to the CS.

The observed latency is a set comprising the latency between a client (e.g., an application server) and different datacenters. The original Pileus system also requires clients to maintain this set. Since the observed latency between datacenters does not change very often, this set is only sent every couple of hours, or when it changes by more than a certain threshold.

Tuba clients also send their SLAs' hit and miss ratios periodically. It has been previously observed that placement algorithms with client workload information (such as the request rate) perform two to five times better than workload oblivious random algorithms [84]. Thus, every client records aggregate ratios of all hit and missed subSLAs for a sliding window of time, and sends them to the CS periodically. The CS then periodically (or upon receiving an explicit request) computes a new configuration such that the overall utility of the system is improved, all constraints are respected, and the cost of the migrating to and maintaining the new configuration remains below some threshold.

Once a new configuration is decided, one or more of the following operations are performed as the system changes to the new configuration: (i) changing the primary replica, (ii) adding or removing secondary replicas, and (iii) changing the synchronization periods between primary and secondary replicas. In the next section, we explain in more detail how the above operations are performed with minimal disruption to active clients.

7.3 Configuration Service (CS)

The CS is responsible for periodically improving the overall utility of the system by computing and applying new configurations. The CS selects a new configuration by first generating all reasonable replication scenarios that satisfy a list of defined constraints.

For each configuration possibility, it then computes the expected gained utility and the cost of reconfiguration. The new chosen configuration is the one that offers the highest utility-to-cost ratio. Once a new configuration is chosen, the CS executes the reconfiguration operations required for making a transition from the old configuration to the new one.

In the remaining of this section, we first explain the different types of constraints and the cost model used by the CS. Then, we introduce the algorithm behind the CS to compute a new configuration. Finally, we describe how the CS executes different reconfiguration operations to install the new configuration.

7.3.1 Constraints

Given the simple goal of maximizing utility, the CS would have a *greedy* nature: it would generally decide to add replicas. Hence, without constraints, the CS could ultimately replicate data in all available datacenters. To address this issue, a system administrator is able to define constraints for the system that the CS respects.

Through an abstract constraint class, Tuba allows constraints to be defined on any attribute of the system. For example, a constraint might disallow creating more than three secondary replicas or disallow a reconfiguration to happen if the total number of online users is greater than 1 million. Tuba abides by all defined constraints during every reconfiguration.

Several important constraints are currently implemented and ready for use including: (i) Geo-replication factor, (ii) Location, (iii) Synchronization period, and (iv) Cost.

With geo-replication constraints, the minimum and maximum number of replicas can be defined. For example, consider an online music store. Developers may set the maximum geo-replication factor of tablets containing less popular songs to one, and set the minimum geo-replication factor of a tablet containing top-ten best selling songs to three. Even if the storage cost is relatively small, limiting the replication factor may still be desirable due to the cost of communication between sites for replica synchronization.

Location constraints are able to explicitly force replication in certain sites or disallow them in others. For example, an online social network application can respond to security concerns of European citizens by allowing replication of their data only in Europe datacenters.

With the synchronization period constraint, application developers can impose bounds on how often a secondary replica synchronizes with a primary replica.

The last and perhaps most important constraint in Tuba is the cost constraint. As mentioned before, the CS picks a configuration with the greatest ratio of gained utility over cost. With a cost constraint, application developers can indicate how much they are willing to pay (in terms of dollars) to switch to a new configuration. For instance, one possible configuration is to put secondary replicas in all available datacenters. While the gained utility for this configuration likely dominants all other possible configurations, the cost of this configuration may be unacceptably large. In the next section, we explain in more detail how these costs are computed in Tuba.

Should the system administrator neglect to impose any constraint, Tuba has two default constraints in order to avoid aggressive replication and to avoid frequent synchronization between replicas: (1) a lower bound for the synchronization period, and (2) an upper bound on the recurring cost of a configuration.

7.3.2 Cost Model

The CS considers the following costs for computing a new configuration:

- Storage: the cost of storing a tablet in a particular site.
- Read/Write Operation: the cost of performing read/write operations.

- Synchronization: the cost of synchronizing a secondary replica with a primary one.

The first two costs are computed precisely for a certain period of time, and the third cost is estimated based on read and write ratios.

Given the above categories, the cost of a primary replica is the sum of its storage and read and write operation costs, and the cost of a secondary replica is the sum of storage, synchronization, and read operation costs. Since Tuba uses batching for synchronization to a secondary replica and only sends the last write operation on an object in every synchronization cycle, the cost of a primary replica is usually greater than that of secondary replicas.

In addition to the above costs, the CS also considers the cost of creating a new replica; this cost is computed as one-time synchronization cost.

7.3.3 Selection

Potential new configurations are computed by the CS in the following three steps:

Ratios aggregation. Clients from the same geographical region usually have similar observed access latencies. Therefore, as long as they use the same SLAs, their hit and miss ratios can be aggregated to reduce the computation. We note that this phase does not necessarily need to be in the critical path, and aggregations can be done once clients send their ratios to the CS.

Configuration computation. In this phase, possible configurations that can improve the overall utility of the system are generated and sorted. For each missed subSLA, and depending on its consistency, the CS may produce several potential configurations along with their corresponding reconfiguration operations. For instance, for a missed subSLA with strong consistency, two scenarios would be: (i) creating a new replica near the client and making it the solo primary replica, or (ii) adding a new primary replica near the client and making the system run in multi-primary mode.

Each new configuration has an associated cost of applying and maintaining it for a certain period of time. The CS also computes the overall gained utility of every new configuration that it considers. Finally, the CS sorts all potential configurations based on their gained utility over their cost.

Constraints satisfaction. Configurations that cannot satisfy all specified constraints are eliminated from consideration. Constraint classes also have the ability to add configurations being considered. For instance, the minimum geo-replication constraint might remove low-replica configurations and create several new ones with additional secondary replicas at different locations.

Rank	Consistency	Latency(ms)	Utility
1	Strong	100	1
2	RMW	100	0.9
3	Eventual	1000	0.01

Figure 7.2: SLA of a Social Network Application

7.3.4 Operations

Once a new configuration is selected, the CS executes a set of reconfiguration operations to transform the system from the current configuration. In this section, we explain various reconfiguration operations and how they are executed abstractly by the CS, leaving the implementation specifics to Section 7.5.

7.3.4.1 Adjust the Synchronization Period

When a secondary replica is added to the system for a particular tablet, a default synchronization period is set, which defines how often a secondary replica synchronizes with (i.e., receives updates from) the primary replica. Although this value does not affect the latency of read operations with strong or eventual consistency, the average latency of reads with intermediary consistencies (i.e., RMW, monotonic reads, bounded, and causal) can depend heavily on the frequency of synchronization. Typically, the cost of adjusting the synchronization period is smaller than the cost of adding a secondary replica or of changing the locations of primary/secondary replicas. Hence, it is likely that the CS will decide to decrease this period to increase the hit ratios of subSLAs with intermediary consistencies.

For example, consider a social network application with the majority of users located in Brazil and India accessing a storage system with a primary replica located in Brazil, initially, and a secondary replica placed in South Asia with the synchronization period set to 10 seconds. Assume that the SLA shown in Figure 7.2 is set for all read operations. Given the fact that the round trip latency between India and Brazil is more than 350 ms, the first subSLA will never hit for Indian users. Yet, depending on the synchronization period and frequency of write operations performed by Indian users, the second subSLA might hit. Thus, if the CS detects low utility for Indian users, a possible new configuration would be similar to the old one but with a reduced synchronization period.

In this case, the chosen operation to apply the new configuration is *adjust_sync_period*. Executing this operation is very simple since the value of the synchronization period need only be changed in the secondary replica. Clients do not directly observe any difference between the new configuration and the old one, but they benefit from a more up-to-date secondary replica.

Rank	Consistency	Latency(ms)	Utility
1	RMW	50	1
2	Monotonic Read	50	0.5
3	Eventual	500	0

Figure 7.3: SLA of an online multiplayer game

7.3.4.2 Add/Remove Secondary Replica

In certain cases, the CS might decide to add a secondary replica to the system. For example, consider an online multiplayer game with the SLA shown in Figure 7.3 and where the primary replica is located in the East US region. In order to deliver a better user experience to gamers around the globe, the CS may add a secondary replica near users during their peak times. Once the peak time has passed, in order to reduce costs, the CS may decide to remove the added, but now lightly used, secondary replica.

Executing *add_secondary*($site_i$) is straight-forward. A dedicated thread is dispatched to copy objects from the primary replica to the secondary one. Once the whole tablet is copied to the secondary replica, the new configuration becomes available to clients. Clients with the old configuration may continue submitting read operations to previously known replicas, and they eventually will become aware of the newly added secondary replica at $site_i$.

Executing *remove_secondary*($site_i$) is also simple. The CS removes the secondary replica from the current configuration. In addition, a thread is dispatched to physically delete the secondary replica.

7.3.4.3 Change Primary Replica

In cases where the system maintains a single primary site, the CS may decide to change the location of the primary replica. For instance, consider the example given in Section 7.3.4.1. The CS may detect that adjusting the synchronization period between the primary and secondary replicas cannot improve the utility. In this case, the CS may decide to swap the primary and secondary replica roles. During peak times in India, the secondary replica in South Asia becomes the primary replica. Likewise, during peak times in Brazil, the replica in Brazil becomes primary.

The CS calls the *change_primary*($site_i$) operation to make the configuration change. If a secondary replica does not exist in $site_i$, the operation is performed in three steps. First, the CS creates a new empty replica at $site_i$. It also invalidates the configuration cached in clients. As we shall see later, when a cached configuration is invalid, a client needs to contact the CS when executing certain operations. Second, once every cached configuration becomes invalid, the CS makes $site_i$ a WRITE_ONLY primary site. In this mode, all write operations are forwarded to both the primary site and $site_i$, but $site_i$ is not allowed to execute read operations. Finally, once $site_i$ catches up with the primary replica, the CS makes it the solo primary site. If a replica

Rank	Consistency	Latency(ms)	Utility
1	Strong	50	1
2	Eventual	500	0.9

Figure 7.4: Password Checking SLA

exists in $site_i$, the first step is skipped. We will explain the implementation of this operation in Section 7.5.3.

7.3.4.4 Add Primary Replica

For applications that require clients to read up-to-date data as fast as possible, the system may benefit from having multiple primary sites that are strongly consistent. In multi-primary mode, write operations are applied synchronously in several sites before the client is informed that the operation has completed.

Consider the password checking SLA shown in Figure 7.4, and assume that the primary replica is placed in the West US region initially, and two secondary replicas are placed in Asia and Europe. Clients located in the West US always read from the local datacenter and check their passwords with strong consistency, while clients located in the rest of the globe always read eventually consistent password data. Of course, checking passwords with eventual consistency has a major security drawback. If hackers gain access to a user's password, then changing this password is not enough to stop them since secondary replicas do not immediately receive the new writes (containing the new password). Hackers can still login to the system (for some bounded time) if passwords are read from a secondary site.

One way of preventing this problem is to remove all secondary replicas in the system. Although this approach works, the primary replica might become a bottleneck. Clients may observe high latency during login time because all request would go to the West US replica. A better approach is to change the utility of the second subSLA to zero, and then ask the CS for an explicit reconfiguration (instead of waiting for the next reconfiguration round). In this case, the CS could boost the overall utility by changing all secondary replicas into primary replicas. The CS might also decide to add a primary replica in some datacenters where a secondary replica does not even exist.

The operation that performs the configuration transformation is called *add_primary*($site_i$). Its execution is very similar to *change_primary*($site_i$) with one exception. In the third step, instead of making the WRITE_ONLY $site_i$ the solo primary, $site_i$ is added to the list of primary replicas, thereby making the system multi-primary. In this mode, multiple rounds of operations are needed to execute a write. The protocol that we use is described in Section 7.5.2.3.

Operation	Effect	Cost
Decrease synchronization period of secondary replica at $site_i$	Increase hit ratios of subSLAs with bounded, causal, or RMW consistencies for clients near $site_i$	Increase in communication
Add $site_i$ as a secondary replica	Increase hit ratios of subSLAs with eventual or intermediary consistencies for clients near $site_i$	Additional storage; increased communication
Upgrade $site_i$ from secondary to primary, and downgrade $site_j$ from primary to secondary	Increase hit ratios of subSLAs with strong or intermediary consistency for clients near $site_i$; decrease hit ratios of subSLAs with strong or intermediary consistency for clients near $site_j$	No change
Add $site_i$ as a primary replica (upgraded from secondary)	Increase hit ratios of subSLAs with strong or intermediary consistency for clients near $site_i$	Increased communication; increased write latency

Figure 7.5: Summary of Common Reconfiguration Operations, Effects on Hit Ratios, and Costs.

7.3.4.5 Summary

Figure 7.5 summarizes the reconfiguration operations that are generally considered by the CS (inverse and other less common operations are not shown). Note that the listed effects are only potential benefits. Adjusting the synchronization period or adding a secondary replica to $site_i$ does not impact the observed consistency or write latency of clients that are not near this site. These operation can possibly increase the hit ratios of subSLAs with intermediary consistencies observed by clients close to $site_i$. Adding a secondary replica can increase the hit ratios of subSLAs with eventual consistency. Making $site_i$ the solo primary increases the hit ratios of subSLAs with both strong and intermediary consistencies for clients close to $site_i$. However, clients close to the previous primary replica now may miss subSLAs with strong or intermediary consistencies. Adding a primary replica can boost strong consistency without having a negative impact on read operations; but, it increases the cost of write operations for all clients.

7.4 Client Execution Modes

Since the CS may reconfigure the system periodically, clients need to be aware of possible changes in the locations of primary and secondary replicas. Instead of clients asking the CS for the latest configuration before executing each operation, Tuba allows clients to cache the configuration of a

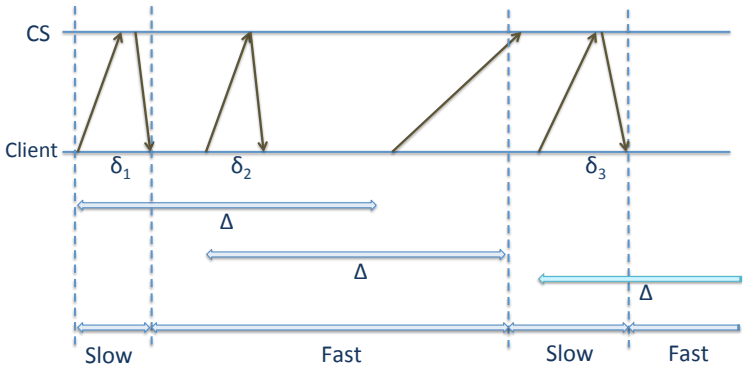


Figure 7.6: Clients Fast and Slow Execution Modes

tablet (called the *cview*) and use it for performing read and write operations. In this section, we explain how clients avoid two potential safety violations: (i) performing a read operation with strong consistency on a non-primary replica, or (ii) executing a write operation on a non-primary replica.

Based on the freshness of a client's cview, the client is either in fast or slow mode. Roughly speaking, a client is in the fast mode for a given tablet if it knows that it has the latest configuration. That is, it knows exactly the locations of primary and secondary replicas, and it is guaranteed that the configuration will not change in the near future. On the other hand, whenever a client suspects that a configuration may have changed, it enters slow mode until it refreshes its local cache.

Initially, every client is in slow mode. In order to enter fast mode, a client requests the latest configuration of a tablet (Figure 7.6). If the CS has not scheduled a change to the location of a primary replica, the client obtains the current configuration along with a promise that the CS will not modify the set of primary replicas within the next Δ seconds. Suppose the duration from when the client issues its request to when it receives the latest configuration is measured to be δ seconds. The client then enters the fast mode for $\Delta - \delta$ seconds. During this period, the client is sure that the CS will not perform a reconfiguration that compromises safety.

In order to remain in fast mode, a client needs to periodically refresh its cview. As long as it receives the latest configuration within the fast mode window, it will remain in fast mode, and its fast mode window is extended.

The CS can force all clients to enter slow mode at any time by preventing them from refreshing their configuration views. This feature is used before executing *change_primary*() and *add_primary*() operations (see Section 7.5.3).

Fast Mode. When a client is in fast mode, read and single-primary write operations involve a single round-trip to one selected replica. No additional overhead is imposed on these oper-

ations. Multi-primary write operations use a three-phase protocol in fast or slow mode (see Section 7.5.2.3).

Slow Mode. Being in slow mode (for a given tablet) means that the client is not totally sure about the latest configuration, and the client needs to take some precautions. Slow mode has no affect on read operations with relaxed consistency, i.e., with any desired consistency except strong consistency. Because read operations with strong consistency must always go to a primary replica, when a client is in slow mode it needs to confirm that such an operation is indeed executed at a current primary replica. Upon completion of a strong consistency read, the client validates that the responding replica selected from its cview is still a primary replica. If not, the client retries the read operation.

Unlike read operations, write operations are more involved when a client is in slow mode. More precisely, any client in slow mode that wishes to execute a write operation on a tablet needs to take a non-exclusive lock on the tablet's configuration before issuing the write operation. On the other hand, the CS needs to take an exclusive lock on the configuration if it decides to change the set of primary replicas. This lock procedure is required to ensure the linearizability [70] of write operations.

7.5 Implementation

Tuba is built on top of Microsoft Azure Storage (MAS) [30] and provides a similar API for reading and writing blobs. Every MAS storage account is associated with a particular storage site. Although MAS supports Read-Access Geo-Redundant Storage (RA-GRS) in which both strong and eventual consistencies are provided, it lacks intermediary consistencies, and replication is limited to a single primary site and a single secondary site. Our implementation extends MAS with: (i) multi-site geo-replication (ii) consistency-based SLAs, and (iii) automatic reconfiguration.

A user of Tuba supplies a set of storage accounts. This set determines all available sites for replication. The CS then selects primary and secondary replica sites by choosing storage accounts from this set. Thus, a configuration is a set of MAS storage accounts tagged with PRIMARY or SECONDARY.

In the rest of this section, we explain the communication between clients and the CS, and how operations are implemented in Tuba. We ignore the implementation of consistency guarantees and consistency-based SLAs since these aspects of Tuba are taken directly from the Pileus system [140].

7.5.1 Communication

Clients communicate with the CS through a designated Microsoft Azure Storage container. Clients periodically write their latency and hit/miss ratios to storage blobs in this shared container. The

CS reads this information and stores the latest configuration as a blob in this same container. Likewise, clients periodically read the latest configuration blob from the shared container and cache it locally.

As we explained in Section 7.4, when a client reads the latest configuration, it enters fast mode for $\Delta - \delta$ seconds. Since there is no direct communication between the client and the CS, we also need to ensure that the CS does not modify a primary replica and install a new configuration within the next Δ seconds. Our solution is simple. When the CS wants to perform certain reconfiguration operations (i.e., changing or adding a primary replica), it writes a special reconfiguration-in-progress (*RiP*) flag to the configuration blob's metadata. The CS then waits for at least Δ seconds before installing the new configuration. If a client fails to refresh its cview on time or if it finds that the *RiP* flag is set, then the client enters slow mode. Once the CS completes the operations needed to reconfigure the system, it overwrites the configuration blob with the latest configuration and clears the *RiP* flag. Clients will re-enter fast mode when they next retrieve the new configuration.

7.5.2 Client Operations

7.5.2.1 Read Operation

For each read operation submitted by an application, the client library selects a replica based on the client's latency, cview, and a provided SLA (as in Pileus). The client then sends a read request to the chosen replica. Upon receiving a reply, if the client is in fast mode or if the read operation does not expect strong consistency, the data is returned immediately to the application. Otherwise, the client confirms that the contacted replica had been the primary replica at the time it answered the read request. More precisely, when a client receives a read reply message in slow mode, it reads the latest configuration and confirms that the timestamp of the configuration blob has not changed.

7.5.2.2 Single-primary Write Operation

To execute a single-primary write operation, a client first checks that it is in fast mode and that the remaining duration of the fast mode interval is longer than the expected time to complete the write operation. If not, it refreshes its cview. Assuming the *RiP* flag is not set, the client then writes to the primary replica. Once the client receives a positive response to this write operation, the client checks that it is still in fast mode. If so, the write operation is finished. If the write operation takes more time than expected such that the client enters slow mode during the execution of the write operation, the client confirms that the primary replica has not changed.

When a client discovers a reconfiguration in progress and remains in slow mode, we considered two approaches for performing writes. The simplest approach is for the client to wait until a new configuration becomes available. In other words, it could wait until the *RiP* flag is removed from

the configuration blob's metadata. The main drawback is that no write operation is allowed on the tablet being reconfigured for Δ seconds and, during this period, the CS does nothing while waiting for all clients to enter slow mode.

Instead, Tuba allows a client in slow mode to execute a write operation by taking a lock. A client acquires a non-exclusive lock on the configuration to ensure that the CS does not change the primary replica before it executes the write operation. The CS, on the other hand, grabs an exclusive lock on the configuration before changing it. This locking mechanism is implemented as follows using MAS's existing lease support. To take a non-exclusive lock on the configuration, a client obtains a lease on the configuration blob and stores the lease-id as metadata in the blob. Other clients wishing to take a non-exclusive lock simply read the lease-id from the blob's metadata and renew the lease. To take an exclusive lock, the CS breaks the client's lease and removes the lease-id from the metadata. The CS then acquires a new lease on the configuration blob. Note that no new write is allowed after this point. After some safe threshold equal to the maximum allowed leased time, the CS updates the configuration.

7.5.2.3 Multi-primary Write Operation

Tuba permits configurations in which multiple servers are designated as primary replicas. A key implementation challenge was designing a protocol that atomically updates any number of replicas on conventional storage servers and that operates correctly in the face of concurrent readers and writers. Our multi-primary write protocol involves three phases: one in which a client marks his intention to write on all primary replicas, one where the client updates all of the primaries, and one where the client indicates that the write is complete. To guard against concurrent writers, we leverage the concept of ETags in Microsoft Azure, which is also part of the HTML 1.1 specification. Each blob has a string property called an ETag that is updated whenever the blob is modified. Azure allows clients to perform a conditional write operation on a blob; the write operation executes only if the provided ETag has not changed.

When an application issues a write operation to a storage blob and there are multiple primary replicas, the Tuba client library performs the following steps.

Step 1: Acquire a non-exclusive lock on the configuration blob. This step is the same as previously described for a single-primary write in slow mode. In this case, the configuration is locked even if the client is in fast mode since the multi-primary write may take longer than Δ seconds to complete. This ensures that the client knows the correct set of primary replicas throughout the protocol.

Step 2: At the main primary site, add a special write-in-progress (*WiP*) flag to the metadata of the blob being updated. The main primary site is the one listed first in the set of primary replicas. This metadata write indicates to readers that the blob is being updated, and it returns an ETag that is used later when the data is actually written. Updates to different blobs can take place in parallel.

Step 3: Write the *WiP* flag to the blob's metadata on all other primary replicas. Note that these writes can be done in any order or in parallel.

Step 4: Perform the write on the main primary site using the ETag acquired in Step 2. Note that since writes are performed first at the main primary, this replica always holds the *truth*, i.e. the latest data. Other primary replicas hold stale data at this point. This conditional write may fail because the ETag is not current, indicating that another client is writing to the same blob. In the case of concurrent writers, the last writer to set the *WiP* flag will successfully write to the main primary replica; clients whose writes fail abandon the write protocol and possibly retry those writes later.

Step 5: Perform conditional writes on all the other primary replicas using the previously aquired Etags. These writes can be done in parallel. Again, a failed write indicates that a concurrent write is in progress. In this case, this client stops the protocol even though it may have written to some replicas already; such writes will be (or may already have been) overwritten by the latest writer (or by a recovery process as discussed in section 5.4).

Step 6: Clear the *WiP* flags in the metadata at all non-main primary sites. These flags can be cleared in any order or in parallel. This allows clients to now read from these primary replicas and obtain the newly written data. To ensure that one client does not prematurely clear the flag while another client is still writing, these metadata updates are performed as conditional writes using the ETags obtained from the writes in the previous step.

Step 7: Clear the *WiP* flag in the metadata on the main primary using a conditional write with the ETag obtained in Step 4. Because this is done as the final step, clients can check if a write is in progress simply by reading the metadata at the main primary replica.

An indication that the write has been successfully completed can be returned to the caller at any time after Step 4 where the data is written to the main primary. Waiting until the end of the protocol ensures that the write is durable since it is held at multiple primaries.

If a client attempts a strongly consistent read while another client is performing a multi-primary write, the reader may obtain a blob from the selected primary replica whose metadata contains the *WiP* flag. In this case, the client redirects its read to the main primary replica who always holds the latest data. Relaxed consistency reads, to either primary or secondary replicas, are unaffected by writes in progress.

7.5.3 CS Reconfiguration Operations

In this section, we only explain the implementation of *change_primary*() and *add_primary*() since the implementation details of adjusting a synchronization period and adding/removing secondary replicas are straightforward.

As we explained before, *change_primary*($site_i$) is the operation required for making $site_i$ the solo primary. If a secondary replica does not exist in $site_i$, the operation is performed in three steps. Otherwise, the first step is skipped.

Step 1: The CS starts by creating a replica at $site_i$, and synchronizing it with the primary replica.

Step 2: Before making $site_i$ the new primary replica, the CS synchronizes $site_i$ with the existing primary replica. Because write operations can run concurrently with a *change_primary*($site_i$) operation, $site_i$ might never be able to catch up with the primary replica. To address this issue, the CS first makes $site_i$ a WRITE_ONLY replica by creating a new temporary configuration. As its name suggests, write operations are applied to both WRITE_ONLY replicas and primary replicas (using the multi-primary write protocol described previously).

The CS installs this configuration as follows:

(i) It writes the *RiP* flag to the configuration blob's metadata, and waits Δ seconds to force all clients into slow mode.

(ii) Once all clients have entered the slow mode, the CS breaks the lease on the configuration blob and removes the lease-id from the metadata.

(iii) It then acquires a new lease on the blob and waits for some safe threshold.

(iv) Once the threshold is passed, the CS safely installs the temporary configuration, and removes the *RiP* flag.

Consequently, clients again switch to fast mode execution while the $site_i$ replica catches up with the primary replica.

Step 3: The final step is to make $site_i$ the primary replica, once $site_i$ is completely up-to-date. The CS follows the procedure explained in the previous step to install a new configuration where the old primary replica is downgraded to a secondary replica, and the WRITE_ONLY replica is promoted to be the new primary. Once the new configuration is installed, $site_i$ is the sole primary replica.

Note that write operations are blocked from the time when the CS takes an exclusive lease on the configuration blob until it installs the new configuration in both steps 2 and 3. However, this duration is short: a round trip latency from the CS to the configuration blob plus the safe threshold.

The *add_primary*() operation is implemented exactly like *change_primary*() with one exception. In the third step, instead of making $site_i$ the solo primary, this site is added to the list of primary replicas.

7.5.4 Fault-Tolerance

Replica Failure. A replica being unavailable should be a very rare occurrence since each of our replication sites is a collection of three Azure servers in independent fault domains. In any case, failed replicas can easily be removed from the system through reconfiguration. Failed secondary replicas can be ignored by clients, while failed primary replicas can be replaced using previously discussed reconfiguration operations.

Client Failure. Most read and write operations from clients are performed at a single replica and maintain no locks or leases. The failure of one client during such operations does not adversely affect others. However, Tuba does need to deal explicitly with client failures that may leave a multi-primary write partially completed. In particular, a client may crash before successfully writing to all primary replicas or before removing the *WiP* flags on one or more primary replicas.

When a client, through normal read and write operations, finds that a write to a blob has been in progress for an inordinate amount of time, it invokes a recovery process to complete the write. The recovery process knows that the main primary replica holds the truth. It reads the blob from the main primary and writes its data to the other primary replicas using the multi-write protocol described earlier. Having multiple recovery processes running simultaneously is acceptable since they all will be attempting to write the same data. The recovery process, after successfully writing to every primary replica, clears all of the *WiP* flags for the recovered blob.

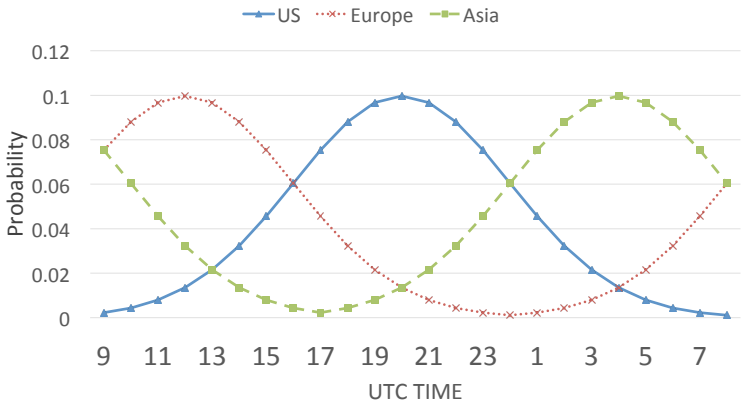
CS Failure. Importantly, the Tuba design does not depend on an active CS in continuous operation. The CS may run only occasionally to check whether a reconfiguration is warranted. Since clients read the latest configuration directly from the configuration blob, and do not rely on responses from the CS, they can stay in fast mode even when the CS is not available as long as the configuration blob is available (and the *RiP* flag is not set). Since the configuration blob is replicated in MAS, it obtains the high-availability guarantees provided by Azure. If higher availability is desired, the configuration blob could be replicated across sites using Tuba's own multi-primary write protocol.

The only troubling scenario is if the CS fails while in the midst of a reconfiguration leaving the *RiP* flag set on the configuration blob. This is not a concern when the CS fails while adjusting a synchronization period or adding/removing a secondary replica. Likewise, a failure before the second step of changing/adding a primary replica does not pose any problem. Even if a CS failure leaves the *RiP* flag set, clients can still perform reads and writes in slow mode.

Recovery is easy if the CS fails during step 2 or during step 3 of changing/adding a primary replica (i.e., after setting the *RiP* flag and before clearing it). When the CS wants to performs a reconfiguration, it obtains an ETag upon setting the *RiP* flag. To install a new configuration, the CS writes the new configuration conditional on the obtained ETag.

A client clears the *RiP* flag upon waiting too long in slow mode. Doing so will prevent the CS from writing a new configuration blob and abort any reconfiguration in progress in the unlikely event that the CS had not crashed but was simply operating slowly. In other words, the CS cannot write the new configuration if some client had impatiently cleared the *RiP* flag and consequently changed the configuration blob's ETag.

Finally, if the CS fails after step 2 of adding/changing a primary replica, clients can still enter fast mode. In case the CS was executing *change_primary*() before its crash, write operations will execute in multi-primary mode. Thus, they will be slow until the CS recovers and finishes step 3.



	SUS	WEU	SEA
US Clients (West US)	53	153	190
Europe Clients (West Europe)	132	<1	277
Asia Clients (Hong Kong)	204	296	36

Figure 7.7: Client Distribution and Latencies (in ms)

7.6 Evaluation

In this section, we present our evaluation results, and show how Tuba improves the overall utility of the system compared with a system that does not perform automatic reconfiguration.

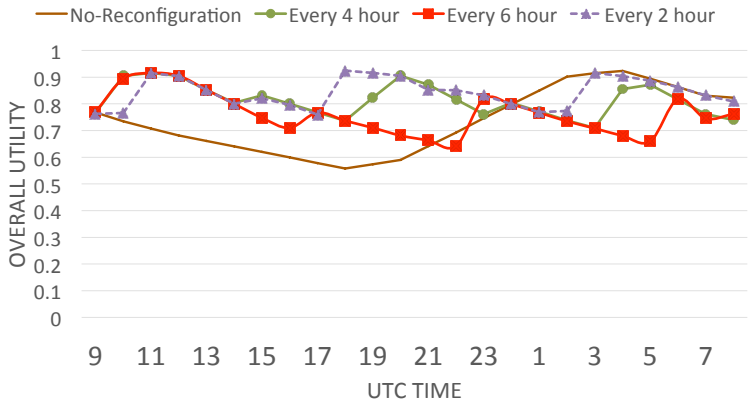
7.6.1 Setup and Benchmark

To evaluate Tuba, we used three storage accounts located in the South US (SUS), West Europe (WEU), and South East Asia (SEA). We modeled the number of active clients with a normal distribution, and placed them in the US West Coast, West Europe, and Hong Kong (Figure 7.7). This is to mimic the workload of clients in different parts of the world during working hours. The mean of the normal distribution is set to 12 o'clock local time, and the variance is set to 8 hours. Considering the above normal distribution, the number of online clients at each hour is computed as a total number of clients times the probability distribution at that hour. The total number of clients at each site is 150 over a 24 hour period. Hence, each tablet is accessed by 450 distinct clients in one day.

We used the YCSB benchmark [37] with workload B (95% Reads and 5% writes) to generate the load. Each tablet contains 10^5 objects, and each object has a 1KB payload. Figure 7.8 shows the SLA used in our evaluation, which resembles one used by a social networking application

Rank	Consistency	Latency(ms)	Utility
1	Strong	100	1
2	RMW	100	0.7
3	Eventual	250	0.5

Figure 7.8: SLA for Evaluation



	Reconf. Every		
	6h	4h	2h
AOU	0.76	0.81	0.85
AOU Impr. over No Reconf.	5%	12%	18%
AOU Impr. over Max. Ach.	20%	45%	65%

AOU: Averaged Overall Utility in 24 hours;
No Reconf. AOU: 0.72; Max. Ach. AOU: 0.92

Figure 7.9: Utility improvement with different reconfiguration rates

[140].

The initial setup places the primary replica in SEA and a secondary replica in WEU. We set the geo-replication factor to two, allowing the CS to replicate a tablet in at most two datacenters. Moreover, we disallowed multi-primary schemes during reconfigurations.

7.6.2 Macroscopic View

Figure 7.9 compares the overall utility for read operations when reconfiguration happens every 2, 4, and 6 hours over a 24 hour period, and when no reconfiguration happens. We note that without reconfiguration Tuba performs exactly as Pileus. The average overall utility (AOU) is computed as the average utility delivered for all read operations from all clients. The average utility improvement depends on how frequently the CS performs reconfigurations. When no re-

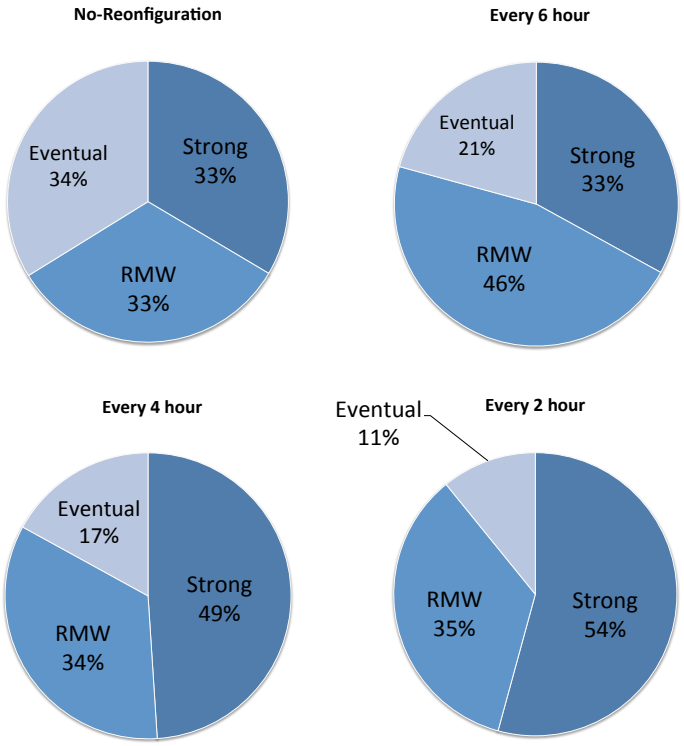


Figure 7.10: Hit Percentage of subSLAs

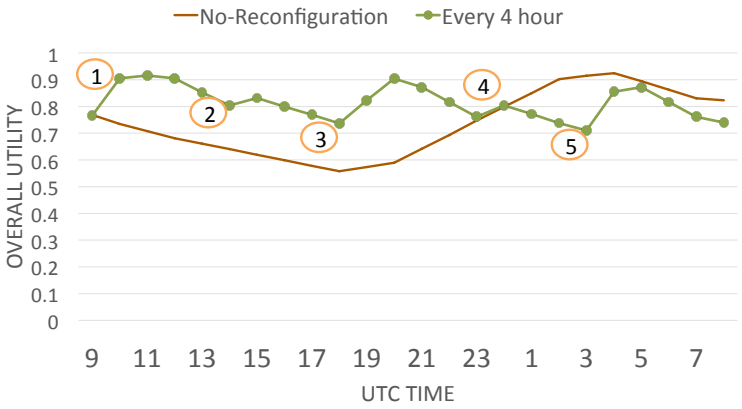
configuration happens in the system, the AOU in the 24 hour period is 0.72. Observe that without constraints, the maximum achievable AOU would have been 1. However, limiting replication to two datacenters and a single primary decreases the maximum achievable AOU to 0.92.

Performing a reconfiguration every 6 hours improves the overall utility for almost 12 hours, and degrades it for 8 hours. This results in a 5 percent AOU improvement. When reconfiguration happens every 4 hours, the overall utility improves for 16 hours. This leads to a 12 percent AOU improvement. Finally, with 2 hour reconfigurations, AOU is improved 18 percent. Note that this improvement is 65 percent of the maximum possible improvement.

Interestingly, when no reconfiguration happens, the overall utility is better than other configurations around UTC midnight. The reason behind this phenomena is that at UTC midnight, the original replica placement is well suited for the client distribution at that time.

Figure 7.10 shows the hit percentages of different subSLAs. With no reconfiguration, 34% of client reads return eventually consistent data (i.e., hit the third subSLA). With 2 hour reconfigurations, Tuba reduces this to 11% (a 67% improvement). Likewise, the percentage of reads returning strongly consistent data increases by around 63%.

Although the computed AOU depends heavily on the utility values specified in the SLA, we believe that the qualitative comparisons in this study are insensitive to the specific values.



Epoch	Primary	Secondary	Reconfiguration
0	SEA	WEU	*change_primary*(*WEU*)
1	WEU	SEA	*add_secondary*(*SUS*) *remove_secondary*(*SEA*)
2	WEU	SUS	*change_primary*(*SUS*)
3	SUS	WEU	*add_secondary*(*SEA*) *remove_secondary*(*WEU*)
4	SUS	SEA	*change_primary*(*SEA*)
5	SEA	SUS	

Figure 7.11: Tuba with Reconfigurations Every 4 hour

Certainly, the hit percentages in Figure 7.10 would be unaffected by varying utilities as long as the rank order of the subSLAs is unchanged.

In addition to reduced utility, systems without support for automatic reconfiguration have additional drawbacks stemming from the way they are manually reconfigured. A system administrator must stop the system (at least for certain types of configuration changes), install the new configuration, inform clients of the new configuration, and then restart the system. Such systems are unable to perform frequent reconfigurations. Moreover, the effect of a reconfiguration on throughput can be substantial since all client activity ceases while the reconfiguration is in progress.

7.6.3 Microscopic View

Figure 7.11 shows how Tuba adapts the system configuration in our experiment where reconfiguration happens every 4 hours. The first five reconfigurations are labeled on the plot. Initially, the primary replica is located in SEA, and the secondary replica is located in WEU. Upon the first reconfiguration, the CS decides to make WEU the primary replica. Though the number of clients in Asia is decreasing at this time, the overall utility stays above 0.90 for two hours before

	Fast Mode		Slow Mode	
	Read	Write	Read	Write
Client in Europe	54	143	270	785
Client in Asia	297	899	533	1598

Figure 7.12: Average Latency (in ms) of Read/Write Operations in Fast and Slow Modes

starting to degrade.

The second reconfiguration happens around 2PM (UTC time) when the overall utility is decreased by 10%. At this time, the CS detects poor utility for users located in the US, and decides to move the secondary replica from SEA to SUS. Since the geo-replication factor is set to 2, the CS necessarily removes the secondary replica in SEA to comply with the constraint. At 6PM, the third reconfiguration happens, and SUS becomes the primary replica. This reconfiguration improves the AOU to more than 0.90. In the fourth reconfiguration, the CS decides to create a secondary replica again in the SEA region. Like the second reconfiguration, in order to respect the geo-replication constraint, the secondary replica in WEU is removed. Note that the fourth reconfiguration is suboptimal since the CS does not predict clients' future behavior and solely focuses on their past behavior. A better reconfiguration would have been to make SEA the primary replica rather than the secondary replica. After 4 hours, the CS performs another reconfiguration and again is able to boost the overall utility of the system.

Although the CS performs *adjust_sync_period*() with two hour reconfiguration intervals, this operation is never selected by the CS when reconfigurations happen every 4 hours. This is because changing the primary or secondary replica boosts the utility enough that reducing the synchronization period would result in little additional benefit.

7.6.4 Fast Mode vs. Slow Mode

In this experiment, we compare the latency of read and write executions in fast and slow modes. Since the latency of read operations with any consistency other than strong does not change in fast and slow modes, we solely focus on the latency of executing read operations with strong consistency and write operations. We placed the configuration blob in the West US (WUS) datacenter, a data tablet in West Europe (WEU), and clients in Central Europe and East Asia. The latency (in ms) between the two clients and the two storage sites are as follows:

	WEU	WUS
Client in Europe	54	210
Client in Asia	296	230

Figure 7.12 compares the average latencies of read and write operations in slow and fast modes. Executing strongly consistent read operations in slow mode requires also reading the

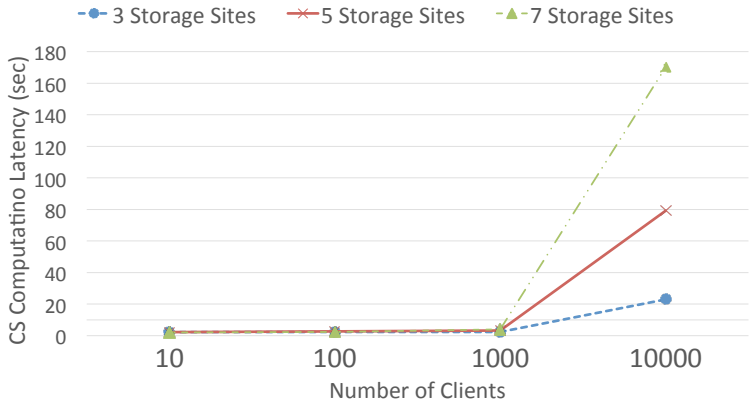


Figure 7.13: Scalability of the CS

configuration blob to ensure that the primary replica has not changed. Therefore, the latency of a read operation in slow mode is more than 200 ms longer than in fast mode.

Executing write operations in slow mode requires three additional RPC calls to the US (where the configuration blob is stored) in the case where no client has written a lease-id to the configuration's metadata (as in this experiment). Specifically, slow mode writes involve reading the latest configuration, taking a non-exclusive lease on the configuration blob, and writing the lease-id to the configuration's metadata. If a lease-id is already set in the configuration's metadata, the last phase is not needed, and two RPC calls are enough. We note that, with additional support from the storage servers, the overhead of write operations in slow mode could be trimmed to only one additional RPC call. This is achievable by taking or renewing the lease in one RPC call to the server that stores the configuration.

7.6.5 Scalability of the CS

As we explained in Section 7.3.3, the CS considers a potentially large number of candidates when selecting a new configuration. To better understand the limitations of the selection algorithm used by our CS, we studied its scalability in practice. We put clients at four sites: East US, West US, West Europe, and Southeast Asia. Each client's SLA has three subSLAs, and all SLAs are distinct; thus, no ratio aggregation is possible. Initially, the East US site is chosen as the primary replica, and no secondary replica is deployed. We also impose the following three constraints: (i) Do not replicate in East US, (ii) Replicate in at least two sites, and (iii) Replicate in a maximum of three sites. We ran the CS on a dual-core 2.20 GHz machine with 3.5GB of memory.

Figure 7.13 plots the latency of computing a new configuration with 3, 5, and 7 available storage sites when the CS performs an exhaustive search of all possible configurations. With one hundred clients, it takes less than 3 seconds to compute the expected utility gain for every configuration and to select the best one. With one thousand clients, the computation time for 3

available storage sites is still less than 3 seconds, while it reaches 3.8 seconds for 7 sites. When the number of clients reaches ten thousand, the CS computes a new configuration for 3 available storage sites in 20 seconds, and for 7 available storage sites in 170 seconds.

This performance is acceptable for many systems since typically the set of cloud storage sites (i.e., the datacenters in which data can be stored) is small and reconfigurations are infrequent. For systems with very large numbers of clients and a large list of possible storage sites, heuristics for pruning the search space could yield substantial improvements and other techniques like ILP or constraint programming should be explored.

7.7 Related Work

Lots of previous work has focused on data placement and adaptive replication algorithms in LAN environments (e.g., [13, 72, 92, 134, 148]). These techniques are not applicable for WAN environments mainly because: (i) Intra-datacenter transfer cost is negligible compare to inter-datacenter cost, (ii) Data should be placed in the datacenters that are closest to users, and (iii) The system should react to users' mobility around the globe. Therefore, in the remaining of this section, we only review solutions tailored specifically for WAN environments.

Kadambi et al. [76] introduce a mechanism for selectively replicating large databases globally. Their main goal is to minimize the bandwidth required to send updates and bandwidth required to forward reads to remote datacenters while respecting policy constraints. They extend Yahoo! PNUTs [36] with a per-record selective replication policy. Their dynamic placement algorithm is based on [148] and responds to changes in access patterns by creating and removing replicas. They replicate all records in all locations either as a full or as a stub. The full replica is a normal copy of the replica while the stub contains only the primary-key and some metadata. Instead of recording access patterns as in Tuba, they rely on a simple approach: a stub replica becomes full when a read operation is delivered at its location, and a full replica demotes when a write operation is observed in another location or if there has not been any read at that location for some period. Unlike Tuba, changing the primary replica is not studied in this work. Moreover, once data is inserted in a tablet, constraint changes are not allowed. In contrast, Tuba allows modifying or adding new constraints, and they will be respected in the next reconfiguration cycle.

Tran et al. [142] introduce a key-value store called Nomad that allows migrating of data between datacenters. They propose and implement an abstraction called overlays. These overlays are responsible for caching and migrating object containers across datacenters. Nomad considers the following three migration policies: (i) count, (ii) time, and (iii) rate. Hence, users can specify the number of times, a certain period, and the rate that data is accessed from the same remote location as migration policies. In comparison, Tuba focuses on maximizing the overall utility of the storage system and respecting replication constraints.

Volley [6] relies on access logs to determine data locations. Their goal is to improve datacenter

capacity skew, inter-datacenter traffic, and client latency. In each round, Volley computes the data placement for *all* data items, while the granularity in Tuba is a tablet. Unlike Tuba, Volley does not take into account the configuration costs or constraints. Moreover, Volley does not propose any migration mechanisms.

Venkataramani et al. [143] propose a bandwidth-constrained placement algorithm for WAN environments. Their main goal is to place copies of objects at a collection of caches to minimize access time. However, complex coordination between distributed nodes and the assumption of a fixed size for all objects makes this scheme less practical.

7.8 Conclusion

Tuba is a replicated key-value store based on Pileus that allows applications to select their desired consistency and dynamically selects replicas that can maximize the utility delivered to read operations. Additionally, it automatically reconfigures itself while respecting user defined constraints so that it adapts to changes in users locations or request rates. Tuba is built on top of Windows Azure Storage (WAS), and extends WAS with broad consistency choices, consistency-based SLAs, and explicit geo-replication configurations.

Our experiments with clients distributed in different datacenters around the world show that Tuba with two hour reconfiguration intervals increases the reads that return strongly consistent data by 63% and improves average utility up to 18%. This confirms that automatic reconfiguration can yield substantial benefits which are realizable in practice.

CHAPTER 8

CONCLUSION

In this thesis, we studied various issues related to consistency criteria in transactional, and non-transactional data stores. In the first part, we focused on transactional data stores, and studied: (i) what are the desired scalability properties of a consistency criterion, and of a transactional system; (ii) how well-known consistency criteria (like Snapshot Isolation or Serializability) behave with respect to these properties; (iii) whether it is possible to design a more scalable strong consistency criterion without introducing additional anomalies, and (iv) performing a fair apples-to-apples comparison among transactional protocols ensuring various consistency criteria. We address the above problems with the following contributions:

Scalability properties. We first identified the following four scalability properties: Genuine Partial Replication, Wait-Free Queries, Forward Freshness Snapshot, and Minimal Commitment Synchronization. We also compared various transactional protocols in terms of their assumptions, and the above scalability properties.

Scalability of Strong Consistency Criteria. We showed that ensuring snapshot isolation in a genuine partial replication system is impossible with obstruction-free updates, and interactive transactions in a failure-free asynchronous system. To state this impossibility result, we proved that SI is decomposable into a set of simpler properties, and proved that two of these properties, namely snapshot monotonicity and strictly consistent snapshots cannot be ensured under GPR. As a corollary, we also showed that a GPR system with obstruction-free updates cannot support SSER, nor Opacity. Moreover, a GPR system with obstruction-free updates and WFQ cannot ensure PSI.

NMSI & Jessy. To sidestep the above impossibility result, we introduced Non-monotonic Snapshot Isolation (NMSI). It requires: (i) transactions to always read committed versions of objects, (ii) transactions to always take consistent snapshots, and (iii) no two concurrent write-conflicting transactions both commit. NMSI is a strong consistency criterion that is able to ensure all four scalability properties. Therefore, it completely leverages the intrinsic parallelism of a workload and minimizes the impact of concurrent transactions on each other. We assessed empirically these benefits by comparing our NMSI implementation (called Jessy) with several replication protocols representative of well-known criteria. Our experiments show that performance of NMSI is close to RC (i.e, the weakest criterion) in a disaster-prone configuration, and up to two times faster than Parallel Snapshot Isolation PSI.

G-DUR. Deferred update replication (DUR) is a classical technique to construct transactional data stores. Protocols that follow the DUR approach share a common algorithmic structure consisting of a speculative execution phase followed by a termination phase. We leveraged the above insight to introduce a generic framework called Generic Deferred Update Replication (G-DUR). G-DUR brings several benefits to practitioners and researchers in the field of transactional storage:

- It eases fast prototyping of a transactional protocol following the DUR approach. We presented our implementations of six state-of-the-art replication protocols published in the past few years [106, 117, 127, 129, 131, 136].

- G-DUR fosters apples-to-apples comparison of transactional protocols. We illustrated this by presenting an empirical evaluation in a geo-replicated environment.

- With G-DUR, a developer can study in detail the limitations and overheads of her protocol. We showed this point with the GMU protocol Peluso et al. [106]. We also presented a variation of P-Store [127] that leverages workload locality. Our variant performs up to 70% faster than the original.

In the second part of this thesis, we focused on the following problems: how to reconfigure a non-transactional data store while ensuring consistencies, and respecting user defined cost objectives, and constraints. To this end, we proposed the following system.

Tuba. Tuba is a replicated key-value store that allows applications to select their desired level of consistency. It dynamically selects replicas to maximize the utility delivered to read operations. In addition, it automatically reconfigures itself while respecting user defined constraints so that it adapts to changes in users locations or request rates. Tuba is built on top of Windows Azure Storage (WAS), and extends WAS with broad consistency choices, consistency-based SLAs, and explicit geo-replication configurations.

Our experiments with clients distributed in different datacenters around the world show that Tuba, with two hour reconfiguration interval, increases the reads that return strongly

consistent data by 63% and improves average utility by up to 18%. This confirms that automatic reconfiguration can yield substantial benefits which are practical.

8.1 Future Work

Decomposition of other strong consistencies In Chapter 4, we decomposed SI into a set of necessary and sufficient properties. For the future work, it remains to decompose other consistency criteria into a set of necessary and sufficient properties. We believe that it is easier to understand, and reason about a consistency criteria using these decomposed properties. Moreover, it is easier to prove a correctness of a transactional system by using these decomposed properties compared to using the classical phenomena based definition of consistency criteria [3].

Impossibility of SER in GPR In Section 4.3, we discussed that if Lemma 4.1 holds for concurrent conflicting transactions, then SER is not attainable in a GPR system with WFQ and OFU. One future research direction is to prove that Lemma 4.1 always holds for concurrent conflicting transactions when a system ensures OFU and WFQ.

Extending G-DUR. Currently, G-DUR is limited to several strong consistency criteria, and does not support session guarantees. One future research direction is to extend the G-DUR study, and add new criteria and protocols (e.g., Causal+ [90, 91], Read Atomicity [18]) along with a support for session guarantees. Another direction of interest is the dynamic adaptation of consistency to the workload. To that regard, we believe that G-DUR can greatly improve our development and evaluation time thanks to its library of execution and termination plug-ins.

Geo-replicated Search Indexes. In this thesis, we assumed that an object is accessed via its primary key. When accessing an object other than through its primary key, a request should be sent to all replicas in the system, and a coordinator needs to wait for replies from all replica groups. Hence, in a partially replicated system, this approach is very slow and not scalable. To sidestep this problem, one solution is to build a distributed search index, for instance a distributed B+Tree or a skip list, for every non-primary key of interest. The search index can later be used for implementing sql-based query language on top of key-value stores for executing complex sql queries. This problem has been studied in the context of single-site data stores (e.g., [7, 46, 149]) but not for geo-replicated systems. One direction of interest is to leverage non-monotonic snapshot isolation for building search indexes.

Reconfigurable transactional store. Our work on Tuba system focuses on storage systems with read/write operations. A future research direction is to extend it to transactional storage systems. Previous research on data allocation algorithms, and automatic configuration of database

systems, targets mainly LAN environments, and is designed for databases with serializability or snapshot isolation [14, 134, 148]. However, they are not well suited to a geo-replicated transactional storage with high latency, and weaker consistency models.

Part III: Appendix

APPENDIX A

PROOF OF SI DECOMPOSITION

In Theorem 4.1, we claim that SI equals the intersection of ACA, SCONS, WCF and MON. This section is devoted to a rigorous proof of this claim. Proposition A.1 states that SI $\subseteq$ ACA $\cap$ SCONS $\cap$ WCF $\cap$ MON holds. In Proposition A.2, we prove that the converse is true. We start our proof by the three technical lemmata bellow.

Lemma A.1. *Consider a history $h \in$ SI and two versions x_i and x_j of some object x. If $x_i \ll_h x_j$ holds then $T_j \rhd^* T_i$ is true.*

Proof. Assume some history $h \in$ SI such that $x_i \ll_h x_j$ holds. Let h_s be an extended history for h that satisfies rules D1 and D2. According to the model, transaction T_j first reads some version x_k, then writes version x_j.

First, assume that there is no write to x between $w_i(x_i)$ and $w_j(x_j)$. Since x belongs to $ws(T_i) \cap ws(T_j)$, rule D2 tells us that either $c_i <_{h_s} s_j$, or $c_j <_{h_s} s_i$ holds. We observe that because $x_i \ll_h x_j$ holds, it must be true that $c_i <_{h_s} s_j$. Since there is no write to x between $w_i(x_i)$ and $w_j(x_j)$, $x_k \ll x_i$ holds, or $k = i$. Observe that in the former case rule D1.3 is violated. Thus, transaction T_j reads version x_i.

To obtain the general case, we apply inductively the previous reasoning. ■

Lemma A.2. *Let $h \in$ SI be an history, and $\mathbb{S}$ be a function such that $h_s = \mathbb{S}(h)$ satisfies D1 and D2. Consider $T_i, T_j \in h$. If $T_i \rightarrow T_j$ holds then $s_i <_{h_s} s_j$.*

Proof. Consider two transactions T_i and T_j such that the snapshot of T_i precedes the snapshot of T_j. By definition of the snapshot precedence relation, there exist $T_k, T_l \in h$ such that $r_i(x_k), r_j(y_l) \in h$ and either *(i)* $r_i(x_k) <_h c_l$, or *(ii)* $w_l(x_l) \in h$ and $c_k <_h c_l$. Let us distinguish each case:

(Case $r_i(x_k) <_h c_l$) By definition of function $\mathcal{S}$, s_i precedes $r_i(x_k)$ in h_s. From $r_j(y_l) \in h$ and rule D1.2, $c_l <_{h_s} s_j$ holds. Hence, $s_i <_{h_s} s_j$ holds.

(Case $c_k <_h c_l$) From *(i)* $r_i(x_k), w_l(x_l) \in h$, *(ii)* $c_k <_h c_l$ and *(iii)* rule D1.3, we obtain $s_i <_{h_s} c_l$. From $r_j(y_l) \in h$ and rule D1.2, $c_l <_{h_s} s_j$ holds. It follows that $s_i <_{h_s} s_j$ holds.

■

Lemma A.3. *Consider a history $h \in ACA \cap CONS \cap WCF$, and two versions x_i and x_j of some object x. If $x_i \ll_h x_j$ holds then $c_i <_h c_j$.*

Proof. Since both T_i and T_j write to x and h belongs to WCF either $T_j \rhd^* T_i$ or $T_i \rhd^* T_j$ holds. We distinguish the two cases below:

(Case $T_j \rhd^* T_i$) First, assume that $T_j \rhd T_i$ holds. Note y an object such that $r_j(y_i)$ is in h. Since h belongs to ACA, $c_i <_h r_j(y_i)$ holds. Because h is an history, $r_j(y_i) <_h c_j$ must hold. Hence we obtain $c_i <_h c_j$. By a short induction, we obtain the general case.

(Case $T_i \rhd^* T_j$) Let us note x_k the version of x read by transaction T_i. From the definition of an history and since h belongs to to ACA, we know that $w_k(x_k) <_h c_k <_h r_i(x_k) <_h w_i(x_i)$ holds. As a consequence, $x_k \ll_h x_i$ is true. Since *(i)* h belongs to CONS, *(ii)* $T_i \rhd^* T_j$, and *(iii)* T_j writes to x, it must be the case that $x_j \ll_h x_k$. We deduce that $x_j \ll_h x_i$ holds; a contradiction.

■

Proposition A.1. $SI \subseteq ACA \cap SCONS \cap WCF \cap MON$

Proof. Choose h in SI. Note $\mathcal{S}$ a function such that history $h_s = \mathcal{S}(h)$ satisfies rules D1 and D2.

($h \in$ ACA) It is immediate from rules D1.1 and D1.2.

($h \in$ WCF) Consider two independent transactions T_i and T_j modifying the same object x. By the definition of an history, $x_i \ll_h x_j$, or $x_j \ll_h x_i$ holds. Applying Lemma A.1, we conclude that in the former case T_j depends on T_i, and that the converse holds in the later.

($h \in$ SCONSa) By contradiction. Assume three transactions T_i, T_j and T_l such that $r_i(x_j), r_i(y_l) \in h$ and $r_i(x_j) <_h c_l$ are true. In h_s, the snapshot point s_i of transaction T_i is placed prior to every operation of T_i in h_s. Hence, s_i precedes $r_i(x_j)$ in h_s. This implies that $s_i <_{h_s} c_l \wedge r_i(y_l) \in h_s$ holds. A contradiction to rule D1.2.

($h \in$ SCONSb) Assume for the sake of contradiction four transactions T_i, T_j, $T_{k \neq j}$ and T_l such that: $r_i(x_j), r_i(y_l)$, $w_k(x_k) \in h$, $c_k <_h c_l$ and $c_k \not<_h c_j$ are all true. Since transaction T_j and T_k both write x, by rule D2, we know that $c_j <_{h_s} c_k$ holds. Thus, $c_j <_{h_s} c_k <_{h_s} c_l$ holds. According to rule D1.2, since $r_i(y_l)$ is in h, $c_l <_{h_s} s_i$ is true. We consequently obtain that $c_j <_{h_s} c_k < s_i$ holds. A contradiction to rule D1.3.

($h \in MON$) If $\rightarrow^*$ is not a partial order, there exist transactions $T_1, \ldots, T_{n \geq 1}$ such that: $T_1 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$. Applying Lemma A.2, we obtain that the relation $s_1 <_{h_s} s_1$ is true. A contradiction.

■

Proposition A.2. *ACA ∩ SCONS ∩ WCF ∩ MON ⊆ SI*

Proof. Consider some history h in ACA ∩ SCONS ∩ WCF ∩ MON. If history h belongs to SI then there must exist a function $\mathcal{S}$ such that $h' = \mathcal{S}(h)$ satisfies rules D1 and D2. In what follows, we build such an extended history h', then we prove its correctness.

[Construction] Initially h' equals h. For every transaction T_i in h' we add a snapshot point s_i in h', and for every operation o_i in h', we execute the following steps:

S1. We add the order (s_i, o_i) to h'.

S2. If o_i equals $r_i(x_j)$ for some object x then

S2a. we add the order (c_j, s_i) to h',

S2b. and, for every committed transaction T_k such that $w_k(x_k)$ is in h, if $c_k <_h c_j$ does not hold then we add the order (s_i, c_k) to h'.

[Correctness] We now prove that h' is an extended history that satisfies rules D1 and D2.

- h' is an extended history.

  Observe that for every transaction T_i in h', there exists a snapshot point s_i, and that according to step S1, s_i is before all operations of transaction T_i. It remains to show that order $<_{h'}$ is acyclic. We proceed by contradiction.

  Since h is a history, it follows that any cycle formed by relation $<_{h'}$ contains a snapshot point s_i. Furthermore, according to steps S1 and S2 above, we know that for some operation $c_{j \neq i}$, relation $c_j <_{h'} s_i <^*_{h'} c_j$ holds.

  By developing relation $s_i <^*_{h'} c_j$, we obtain the following three relations. The first two relations are terminal, while the last is recursive.

  - Relation $s_i <_{h'} c_j$ holds. This relation has to be produced by step S2b. Hence, there exist operations $r_i(x_k), w_j(x_j)$ in h' such that $c_j <_h c_k$ does not hold. Observe that since h belongs to ACA ∩ CONS ∩ WCF, by Lemma A.3, it must be the case that $c_k <_h c_j$ holds.
  - Relation $s_i <_{h'} o_i <^*_h c_j$ holds for some read operation o_i in T_i. (If $o_i <^*_h c_j$ with o_i a write or a terminating operation, we may consider a preceding read that satisfies the same relation.)
  - Relation $s_i <_{h'} o_i <^*_{h'} c_j$ holds for some read operation o_i in T_i, and $o_i <^*_{h'} c_j$ does not imply $o_i <^*_h c_j$. (Again if o_i is a write or a terminating operation, we may consider a preceding read that satisfies this relation.) Relation $o_i <^*_{h'} c_j$ cannot be produced by steps S1 and S2. Hence, there must exist a commit operation c_k and a snapshot point s_l such that $s_i <_{h'} o_i <_h c_k <_{h'} s_l <^*_{h'} c_j$ holds.

  From the result above, we deduce that there exist snapshot points $s_1, \ldots, s_{n \geq 1}$ and commit points $c_{k_1} \ldots c_{k_n}$ such that:

  (A.1) $$s_1 \prec c_{k_1} <_{h'} s_2 \prec c_{k_2} \ldots s_n \prec c_{k_n} <_{h'} s_1$$

where $s_i \prec c_{k_i}$ is a shorthand for either *(i)* $s_i <_{h'} c_{k_i}$ with $r_i(x_j), w_{k_i}(x_{k_i}) \in h$ and $c_j <_h c_{k_i}$, or *(ii)* $s_i <_{h'} o_i <_h c_{k_i}$ with o_i is some read operation.

We now prove that for every i, $T_i \rightarrow T_{i+1}$ holds. Consider some i. First of all, observe that a relation $c_{k_{i-1}} < s_i$ is always produced by step S2a. Then, since relation $s_i \prec c_{k_i} <_{h'} s_{i+1}$ holds we may consider the two following cases:

- Relation $s_i <_{h'} c_{k_i} <_{h'} s_{i+1}$ holds with $r_i(x_j), w_{k_i}(x_{k_i}) \in h$ and $c_j <_h c_{k_i}$. From $c_{k_i} <_{h'} s_{i+1}$ and step S2a, there exists an object y such that $r_{i+1}(y_{k_i})$. Thus, by definition of the snapshot precedence relation, $T_i \rightarrow T_{i+1}$ holds.
- Relation $s_i \prec c_{k_i}$ equals $s_i <_{h'} o_i <_h c_{k_i}$ where o_i is some read operation of T_i, Since $c_{k_i} <_{h'} s_{i+1}$ is produced by step S2a, we know that for some object y, $r_{i+1}(y_{k_i})$ belongs to h. According to the definition of the snapshot precedence, $T_i \rightarrow T_{i+1}$ holds.

Applying the result above to Equation A.1, we obtain: $T_1 \rightarrow T_2 \ldots \rightarrow T_n \rightarrow T_1$. History h violates MON, a contradiction.

- h' satisfies rules D1 and D2.

  (h' satisfies D1.1) Follows from $h \in$ ACA,

  (h' satisfies D1.2) Immediate from step S1.

  (h' satisfies D1.3) Consider three transactions T_i, T_j and T_k such that operations $r_i(x_j)$, $w_j(x_j)$ and $w_k(x_k)$ are in h. The definition of an history tells us that either $x_k \ll_h x_j$ or the converse holds. We consider the following two cases:

  (Case $x_k \ll_h x_j$) Since h belongs to ACA∩CONS∩WCF, Lemma A.3 tells us that $c_k <_h c_j$ holds. Hence, $c_k <_{h'} c_j$ holds.

  (Case $x_j \ll_h x_k$) Applying again Lemma A.3, we obtain that $c_j <_h c_k$ holds. Since $<_h$ is a partial order, then $c_j <_h c_k$ does not hold. By step S2b, the order (s_i, c_k) is in h'.

  (h' satisfies D2) Consider two conflicting transaction (T_i, T_j) in h'. Since h belongs to WCF, one of the following two cases occurs:

  (Case $T_i \rhd^* T_j$) At first glance, assume that $T_i \rhd^* T_j$ holds. By step S2a, s_i is in h' after every operation c_j such that $r_i(x_j)$ is in h', and by step S1, s_i precedes the first operation of T_i. Thus $c_j <_{h'} s_i$ holds, and h' satisfies D2 in this case. To obtain the general case, we applying inductively the previous reasoning.

  (Case $T_j \rhd^* T_i$) The proof is symmetrical to the case above, and thus omitted.

■

From the conjunction of Proposition A.1 and Proposition A.2, we deduce that:

Theorem 4.1. *SI = SCONS ∩ MON ∩ WCF ∩ ACA*

This decomposition is *well-formed* in the sense that the four properties SCONS, MON, WCF and ACA are distinct and that no strict subset of {SCONS, MON, WCF, ACA} attains SI.

Proposition A.3. *For every $S \subsetneq \{SCONS, MON, WCF, ACA\}$, it is true that $\cap_{X \in S} X \neq SI$.*

Proof. For every set $S \subsetneq \{\text{SCONS}, \text{MON}, \text{WCF}, \text{ACA}\}$ containing three of the four properties, we exhibit below an history in $\cap_{X \in S} X \setminus \text{SI}$. Trivially, the result then holds for every S. (SCONS $\cap$ ACA $\cap$ WCF) History h_6 in Section 4.1. (MON $\cap$ ACA $\cap$ WCF) History h_5 in Section 4.1 (SCONS $\cap$ MON $\cap$ WCF) History $r_1(x_0).w_1(x_1).r_a(x_0).c_1.c_a$. (SCONS $\cap$ MON $\cap$ ACA) History $r_1(x_0).r_2(x_0).w_1(x_1).w_2(x_2).c_1.c_2$ ■

APPENDIX B

CORRECTNESS OF JESSY

In this section, we prove the correctness of Jessy. In the first section, we prove the safety of Jessy, and show that Jessy generates NMSI histories in Proposition B.1. In the second section, we first prove liveness of Jessy in Proposition B.2. We then show that Jessy ensures WFQ (Proposition B.3) and NTU (Proposition B.4). Finally, we prove that Jessy^{ofu} guarantees OFU in Proposition B.5.

B.1 Safety

Lemma B.1. *If a transaction T_i commits (respectively aborts) at some process in replicas($ws(T_i)$)$\cup$ $coord(T_i)$, it commits (resp. aborts) at every correct process in replicas($ws(T_i)$)$\cup coord(T_i)$.*

Proof. This proposition follows from the properties of atomic multicast, the fact that the queue Q is FIFO, the preconditions at lines 14 to 15 in Algorithm 4, and the definitions of *vote*() and *outcome*(). ■

By using the above lemma, we now prove that Jessy ensures NMSI as its safety property.

Proposition B.1. *Every history admissible by Jessy belongs to NMSI.*

Proof. We first observe that transactions in Jessy always read committed versions of the objects (line 16 in Algorithm 3). Moreover, we know by Theorem 5.1 that reads are consistent when Jessy uses dependence vectors, and that this property also holds in case Jessy employs partitioned dependence vectors (Proposition 5.2). It thus remains to show that histories generated by Jessy are write-conflict free (WCF).

To prove that WCF holds, we consider two independent write-conflicting transactions T_i and T_j, and we assume for the sake of contradiction that they both commit. We note p_i (resp.

p_j) the coordinator of T_i (resp. T_j). Since T_i and T_j write-conflict, there exists some object x in $ws(T_i) \cap ws(T_j)$. One can show that the following claim holds:

(C1) For any two replicas p and q of x, denoting $committed_p$ (resp. $committed_q$) the set $\{T_j \in committed : x \in ws(T_j)\}$, at the time p (resp. q) decides T_i, it is true that $committed_p$ equals $committed_q$.

According to line 20 of Algorithm 4 and the definition of function *outcome*(), p_i (respectively p_j) received a positive VOTE message from some process q_i (resp. q_j) replicating x. Observe that T_i (resp. T_j) is in variable $\mathbb{Q}$ at process q_i (resp. q_j) before this process sends its VOTE message. It follows from claim C1 that either (1) at the time q_i sends its VOTE message, $T_j <_{\mathbb{Q}} T_i$ holds, or (2) at the time q_j sends its VOTE message, $T_i <_{\mathbb{Q}} T_j$ holds. Assume that case (1) holds (the reasoning for case (2) is symmetrical). From the precondition at line 15 in Algorithm 4, we know that process q_i must wait that T_j is decided before casting a vote for T_i. From Lemma B.1, we deduce that T_j is committed at process q_i. Hence, *certify*(T_i) returns *false* at process q_i; a contradiction. ■

B.2 Liveness and Progress

Lemma B.2. *For every transaction T_i, if coord(T_i) executes T_i and coord(T_i) is correct, then eventually T_i is submitted to the termination protocol at coord(T_i).*

Proof. Transaction T_i executes all its write operations locally at its coordinator. Now, upon executing a read request on some object x, if x was modified previously by T_i, the corresponding value is returned. Otherwise, *coord*(T_i) sends a read request to *replicas*(x). To prove this lemma, we have to show that eventually one of the replica replies to the coordinator.

According to our model, there exists one correct process replica of x. In what follows, we name it p. Observe that since links are quasi-reliable, p eventually receives the read request from *coord*(T_i). Upon receiving this request, process p tries returning a version of x compatible with all versions previously read by T_i.

Consider that Jessy uses dependence vectors (the reasoning for partitioned dependence vectors is similar), and assume, by contradiction, that p never finds such a compatible version. From the definition of *compat*(T_i, x_j, y_l), this means that the following predicate is always true:

$$\forall (x,v,l) \in ds : V(w_l(x_l))[x] < V(r_i(y_j))[x] \\ \vee\ V(w_l(x_l))[y] > V(r_i(y_j))[y]$$

This means that there exists a version x_k upon which transaction T_i depends, and such that $V(w_k(x_k))[x] = V(r_i(y_j))[x]$. Transaction T_k committed at some site. As a consequence, Lemma B.1 tells us that eventually T_k commits at process p. We conclude by observing that since Jessy satisfies both CONS and WCF, $V(w_k(x_k))[y] > V(r_i(y_j))[y]$ cannot hold.

■

By leveraging the above lemma, we now prove the liveness of Jessy, and show that if the coordinator of T_i is correct, T_i always terminates.

Proposition B.2. *For every transaction T_i, if T_i is submitted at coord(T_i) and coord(T_i) is correct, every correct process in replicas($ws(T_i)$) $\cup$ coord(T_i) eventually decides, and terminates T_i.*

Proof. According to Lemma B.2 and the properties of atomic multicast, transaction T_i is delivered at every correct process in *replicas*($ws(T_i)$) $\cup$ *coord*(T_i). It is then enqueued in variable $\mathcal{Q}$ (lines 10 to 11 in Algorithm 4).

Because $\mathcal{Q}$ is FIFO, processes dequeue transactions in the order they deliver them (lines 14 to 15). The uniform prefix order and acyclicity properties of genuine atomic multicast ensure that no two processes in the system wait for a vote from each other. It follows that every correct replica in *replicas*($ws(T_i)$) eventually dequeues T_i, and sends the outcome of function *certify*(T_i) to *replicas*($ws(T_i)$) $\cup$ *coord*(T_i) (lines 16 to 17).

Since there exists at least one correct replica for each object modified by T_i eventually every correct process in *replicas*($ws(T_i)$) $\cup$ *coord*(T_i) collects enough votes to decide upon the outcome of T_i. ■

The following two proposition proves WFQ and NTU as the progress property for read-only and update transactions in Jessy.

Proposition B.3. *Jessy ensures wait-free queries.*

Proof. Consider some read-only transaction T_i and assume that *coord*(T_i) is correct, Lemma B.2 tells us that T_i is eventually submitted at *coord*(T_i).

According to the definition of predicate *outcome*, *outcome*(T_i) always equals true. Hence, the precondition at line 20 in Algorithm 4 is always true, whereas precondition at line 26 is always false. It follows that T_i eventually commits. ■

Proposition B.4. *Jessy ensures non-trivial updates.*

Proof. We need to prove that: in every execution ρ such that $h = \mathfrak{F}(\rho)$ is quiescent (i.e., no transaction is pending), for every transaction $T_i \notin h$, there exists an extension ρ' of ρ such that transaction T_i commits in history $\mathfrak{F}(\rho')$.

Consider some update transaction T_i such that *coord*(T_i) is correct. Proposition B.2 proves that transaction T_i eventually terminates.

At the time T_i starts its execution, there is no pending transaction in history h. This implies that there exists a correct replica at every replica group that has committed all write-conflicting transactions with T_i. Therefore, there exist an execution ρ' such that T_i reads from these replicas. Consequently, T_i depends on all write-conflicting transactions, and the outcome of *certify*(T_i) is true.

■

In what follows, we prove that Jessy^{ofu} ensures OFU for update transactions.

Proposition B.5. *Jessy^{ofu} ensures obstruction-free updates.*

Proof. We need to prove that: in every execution ρ, for every transaction T_i in $\mathfrak{F}(\rho)$, if T_i aborts in h then T_i write-conflicts with some concurrent pending transaction in h.

Consider some update transaction T_i such that $coord(T_i)$ is correct. Proposition B.2 proves that transaction T_i eventually terminates.

Assume that at the time T_i starts its execution, every transaction write-conflicting with T_i has terminated. This implies that there exists q_w replicas that has committed write-conflicting transactions with T_i. Observe that when $coord(T_i)$ executes a read operation, it waits for q_r replicas to reply such that $q_r \cap q_w \neq \varnothing$. Therefore, in any execution ρ', T_i depends on all write-conflicting transactions. Consequently, the outcome of $certify(T_i)$ is true. ■

APPENDIX C

RÉSUMÉ DE LA THÈSE

Contents

C.1 Résumé

Les applications basées sur l'informatique en nuage (*cloud*), comme celles de réseau social ou de commerce électronique, nécessitent de répliquer les données sur plusieurs sites, afin d'améliorer la réactivité, d'être disponibles, et de tolérer les désastres. Il est donc capital de savoir assurer la cohérence sur un système de grande échelle, comportant des connexions WAN, lentes et soumises à panne. C'est le sujet de la présente thèse.

Dans une première partie, nous étudions la cohérence dans les systèmes transactionnels, en nous concentrant sur le problème de réconcilier la *scalabilité* (c-à-d la capacité à passer à l'échelle) avec des garanties transactionnelles fortes. Nous identifions quatre propriétés critiques pour la scalabilité : *(i)* seules les répliques mises à jour par une transaction T participent à l'exécution de T ; *(ii)* une transaction en lecture seule n'attend jamais une transaction concurrente, et est toujours confirmée (*commit*) ; *(iii)* une transaction peut lire des versions confirmées après son démarrage ; et *(iv)* deux transactions se synchronisent uniquement si leurs écritures sont en conflit. Nous montrons qu'aucun des critères de cohérence forte existants n'assurent l'ensemble de ces propriétés. Nous définissons un nouveau critère, appelé Non-Monotonic Snapshot Isolation ou NMSI, qui est le premier à être compatible avec les quatre propriétés à la fois. Nous présentons aussi une mise en œuvre de NMSI, appelée Jessy, que nous comparons expérimentalement à plusieurs critères connus. Notre dernière contribution, dans cette première partie, est un canevas permettant de comparer de façon non biaisée différents protocoles transactionnels. Elle se base sur la constatation qu'une large classe de protocoles transactionnels distribués est basée sur une même structure, *Deferred Update Replication* ou DUR. Les protocoles de cette classe ne diffèrent que par les comportements spécifiques d'un petit nombre de fonctions génériques. Nous présentons donc un canevas générique pour les protocoles DUR, appelé G-DUR, avec une bibliothèque de réalisations finement optimisées des comportements désirés. Notre étude empirique montre que :

(i) G-DUR permet de développer différents protocoles transactionnels avec quelques centaines de lignes de code ; *(ii)* il assure une comparaison équitable et non biaisée entre protocoles ; *(iii)* par simple remplacement de comportements, un développeur peut utiliser G-DUR pour comprendre les goulots d'étranglement de son protocole ; *(iv)* par conséquence, cela permet d'améliorer un protocole existant ; et *(v)* pour un protocole donné, G-DUR permet d'évaluer le coût de divers degrés de fiabilité.

La seconde partie de la thèse a pour sujet la cohérence dans les systèmes de stockage non transactionnels. C'est ainsi que nous décrivons Tuba, un stockage clef-valeur qui choisit dynamiquement ses répliques, afin de maximiser l'utilité perçue par les opérations de lecture, selon un objectif de niveau de cohérence fixé par l'application. Ce système reconfigure automatiquement son ensemble de répliques, tout en respectant les objectifs de cohérence fixés par l'application, afin de s'adapter aux changements dans la localisation des clients ou dans le débit des requête. Nous avons mis en œuvre Tuba au-dessus de *Microsoft Azure Storage (MAS)*. Tout en fournissant une API similaire, Tuba rajoute à MAS un large choix de types de cohérence, de SLA de cohérence, et un service de configuration géo-répliquée. Notre évaluation montre que Tuba accroît le nombre de lectures renvoyant une version fortement cohérente de 63%, et accroît l'utilité moyenne de 18%, par rapport à un système statiquement configuré.

C.2 Introduction

Les applications en nuage sont disponibles depuis de nombreux points d'accès distribués et lointains. Pour améliorer leur réactivité, leur disponibilité, et pour être tolérant aux désastres, les systèmes de stockage en nuage sont répliqués sur plusieurs sites (centres de calculs ou *data centers*) géographiquements distincts (géo-replication).

Malheureusement, les critères classiques pour la cohérence forte ne passent pas bien à l'échelle, sous forte charge et avec des distances élevées. Plusieurs travaux ont cependant déjà été menés pour définir des critères de cohérence qui permettent de garantir à la fois des garanties compréhensibles pour les applications, et qui réussissent à passer à l'échelle [3, 22, 57, 63, 68, 90, 106, 136]. Cependant, les implications en terme de performance et de passage à l'échelle de ces critères de cohérence, ainsi que les protocoles qui les assurent ne sont toujours pas bien compris.

À contrario, plusieurs auteurs estiment que les systèmes géo-répliqués ne devraient fournir que de la cohérence à terme *(Eventual Consistency)* [1, 144]. En effet, le résultat d'impossibilité CAP nous apprend que, en cas de fautes réseau, il faut nécessairement faire un choix entre la disponibilité *(availability)* et la cohérence *(consistency)* [58]). De plus, la cohérence forte induit une latence élevée, d'autant plus dans les réseaux longue distance *(Wide-Area Networks, WAN)*. Malheureusement, la cohérence à terme est à la fois complexe à appréhender pour les développeurs, et insuffisante pour certaines applications (par exemple, pour une application bancaire).

C.2.1 Contributions

Les contributions de cette thèse sont divisées en deux parties.

Dans la première, nous expliquons comment assurer la cohérence dans des systèmes transactionnels. Nous utilisons une approche systématique, afin d'étudier les limites de la scalabilité des systèmes de cohérence transactionnelle, et pour réconcilier les deux objectifs concurrents de passage à l'échelle et de garantie forte.

Dans la seconde partie, nous expliquons comment assurer de la cohérence dans un système de stockage de type nuage, non transactionnel. En particulier, nous expliquons comment procéder pour effectuer de la reconfiguration automatique du système de stockage, tout en assurant la cohérence.

C.2.1.1 Partie I

Étude des protocoles transactionnels en réplication partielle La réplication totale (*full réplication*, où tous les processus ont une copies de chaque objet) ne passe pas à l'échelle, puisque toutes les répliques doivent exécuter toutes les mises à jour. De plus, le stockage et la gestion de grande quantités de données demandent des ressources de stockage importantes sur chaque processus. La réplication partielle évite ces problèmes, en ne répliquant qu'une fraction des données sur un processus donné. Cela permet au système d'accéder et modifier des sous-ensembles indépendants des données de manière concurrente.

Notre première contribution est une étude des protocoles transactionnels en réplication partielle. Notre étude comparative se base sur des métriques classiques (comme le modèle de fautes, l'hypothèse de synchronisation, ou le critère de cohérence), et sur les propriétés de scalabilité. Plus précisément, nous définissons dans un premier temps quatre propriétés cruciales pour le passage à l'échelle.

(i) *Wait-Free Queries* : une requête (transaction en lecture seule n'attend jamais une transaction concurrente, et est toujours confirmée. Cette propriété diminue fortement le surcoût de la synchronisation, dans le cas (courant) où la charge contient une forte proportion de transactions en lecture seule.

(ii) *Forward Freshness* : une transaction peut lire la version d'un objet qui aurait été confirmée après le début de la transaction. Cette propriété permit de baisser la caducité des lectures, et donc le taux d'annulations.

(iii) *Genuine Partial Replication (GPR)* : seules les répliques mises à jour par une transaction T effectuent des opérations pour exécuter T. Cette propriété assure que des transactions qui n'entrent pas en conflit n'interfèrent pas les unes avec les autres. Ceci permet d'exploiter pleinement le parallélisme de la charge de travail.

(iv) *Minimal Commit Synchronization* : deux transactions ne se synchronisent entre elles que si leurs écritures sont en conflit. Il n'y a donc de synchronisation qu'en cas d'absolue nécessité.

Nous passons ensuite en revue un certain nombre de protocoles transactionnels, et nous montrons qu'aucun d'entre eux ne permet d'assurer l'ensemble de ces quatre propriétés.

Passage à l'échelle des critères de cohérence forte Notre seconde contribution consiste à étudier en détail les limitations de passage à l'échelle de plusieurs critères de cohérence. Nous étudions en particulier *Snapshot Isolation (SI)*, une approche très utilisée car elle donne de bons résultats, tant dans le domaine des bases de données distribuées, que dans celui de la mémoire transactionnelle.

Nous décomposons d'abord SI en une conjonction de propriétés nécessaires et suffisantes. Ensuite, nous étudions les implications de chacune de ces propriétés sur la scalabilité, et en particulier sur le critère GPR, et montrons que certaines d'entre elles sont impossibles à garantir, si l'on s'impose des hypothèses de progrès raisonnables. Il s'ensuit que d'autres critères de cohérence forte (p.ex. la sérialisabilité) sont également incompatibles avec GPR, et ne passent donc pas non plus à l'échelle.

Non-Monotonic Snapshot Isolation (NMSI) Pour contourner ce résultat d'impossibilité, nous introduisons un nouveau critère de cohérence, appelé *Non-monotonic Snapshot Isolation* (NMSI). Pour respecter NMSI, toute transaction doit lire une version confirmée de tout objet, et prendre un instantané cohérent. De plus, deux transactions concurrentes de peuvent pas être toutes deux confirmées. NMSI est le critère de cohérence le plus fort assurant l'ensemble des quatre propriétés de scalabilité. Nous présentons également un protocole asurant NMSI, appellé Jessy. Jessy utilise des vecteurs de dépendance, une nouvelle structure de données qui permet le calcul efficace d'instantanés cohérents. Enfin, nous effectuons une évaluation empirique de la scalabilité de NMSI, par comparaison soignée et équitable face à plusieurs critères classiques, dont SER, US, SI et PSI. Cette expérience montre que NMSI a des performances proches de celles de RC (le critère le plus faible), même dans une configuration non tolérante aux pannes,

et que nous obtenons dans certains cas des performances deux fois plus rapides que *Parallel Snapshot Isolation (PSI)*.

Generic Deferred Update Replication Il n'est pas facile de s'y retrouver dans la jungle des critères de cohérence et des protocoles transactionnels. Malgré l'abondance de la littérature, les article utilisent un vocabulaire, des formalismes et des points de vue différents. Les différentes implémentations ne sont pas elles-même comparables en raisons des hypothèses différentes posées sur les environnements. Il n'est donc pas aisé d'établir une comparaison objective et scientifique de leurs comportements dans des cas réels.

Pour traiter ces défis, nous proposons une nouvelle approche. Notre analyse montre que beaucoup des protocoles de mise à jour différées et répliquées (deferred update réplication) ont une structure commune, et ne se différencient que par des instantiations de quelques fonctions génériques (e.g., [12, 100, 103, 104, 106, 117, 125, 127, 129–131, 136]). Nous expliquons cette analyse comme une structure algorithmique commune, avec quelques points de réalisation bien identifiés. Cette structure générique est mise en œuvre dans un protocole spécifique en sélectionnant les plug-in appropriés d'une librairie. Par le choix des plug-ins adaptés, il est alors relativement aisé d'obtenir une implémentation d'un protocole avec de hautes performances.

(1) Nous avons donc configuré G-DUR pour implémenter et comparer empiriquement six protocoles transactionnels parmi les plus communs : [106, 117, 127, 129, 131, 136].

(2) Nous montrons également comment un développeurs peut utiliser G-DUR pour comprendre finement les limitations d'un protocole donné. Nous prenons un protocole récemment publié [106], et identifions ses points critiques en remplaçant méthodiquement ses plug-ins par d'autres moins performants.

(3) Cette approche permet également d'aider un développeur à améliorer des protocoles déjà existant. Nous illustrons ce point par une modification de P-Store [127] qui améliore la vitesse de localité de la charge (workload locality) de près de 70%.

(4) Dans notre dernière expérience, nous évaluons le cout de plusieurs degrés de dépendance (dependability). Pour ce faire, nous prenons un protocole qui assure la stérilisation, et nous étudions le prix de la tolérance aux fautes en faisant varier le degré de réplication ainsi l'algorithme utilisé pour les commit.

C.2.1.2 Partie II

Un Système de Stockage Cloud Auto-Configurable Configurer automatique un système de stockage sur cloud permet d'améliorer le service globalement rendu. Tuba est un système répliqué de stockage clef-valeur. Comme certains systèmes déjà existant, il permet aux applications de sélectionner le degré de cohérence qu'elles souhaitent, tout en sélectionnant dynamiquement les répliques pour maximiser l'utilisation des opérations de lecture. De plus, contrairement aux systèmes actuels, il configure automatiquement le jeu de répliques tout en respectant des contraintes définies par l'application, pour s'adapter aux localisations des clients et au taux de requêtes. Tuba a été construit sur le Microsoft Azure Storage (MAS) et fourni une API similaire Il permet d'ajouter à MAS un large choix de paramètres de cohérence, des SLAs basées sur la cohérence ainsi qu'un service de configuration pour la géo-réplication. En comparaison avec un système configuré statiquement, nos évaluation montrent que Tuba augmente de 63% le taux de retour sur les lectures de données fortement cohérentes (*strongly consistent data*) et améliore l'utilisation générale jusqu'à 18%.

C.3 Passage à l'échelle du Critère de Cohérence Forte

Dans ce chapitre, nous étudions le coût du passage à l'échelle de plusieurs critères de cohérence. Nous nous concentrons dans ce chapitre sur Snapshot Isolation (SI) car c'est une approche populaire à la fois dans la réplication de bases de données distribuées et dans les mémoires transactionnelles logicielles [25, 111].

C.3.1 Décomposition SI

Avant d'introduire quatre propriétés dans la conjonction est équivalente à SI, nous définissons SI à partir d'un historique composé d'opérations de lecture, d'écriture et de commit.

Considérons une fonction $\mathcal{S}$ prenant en entrée un historique h et retournant un historique étendu h_s en ajoutant un *snapshot point* à h pour chaque transaction dans h. Étant donné une transaction T_i, le snapshot point de T_i dans h_s, dénoté s_i, précède chaque opération d'une transaction T_i dans h_s. Un historique h est dans SI si et seulement il existe une function $\mathcal{S}$ telle que $h_s = \mathcal{S}(h)$ et h_s satisfasse les règles suivantes :

D1 (Règle de Lecture)

$\forall r_i(x_{j\neq i}), w_{k\neq j}(x_k), c_k \in h_s :$

$c_j \in h_s$ $(D1.1)$

$\wedge\ c_j <_{h_s} s_i$ $(D1.2)$

$\wedge\ (c_k <_{h_s} c_j \vee s_i <_{h_s} c_k)$ $(D1.3)$

D2 (Règle d'Écriture)

$\forall c_i, c_j \in h_s :$

$ws(T_i) \cap ws(T_j) \neq \varnothing$

$\Rightarrow \left(c_i <_{h_s} s_j \vee c_j <_{h_s} s_i\right)$

Nous définissons maintenant 4 propriétés dont la conjonction est équivalente à SI.

C.3.1.1 Annulation en cascade (Absence of Cascading Aborts)

Intuitivement, une transaction en lecture seule doit s'annuler si elle observe les effets d'une transaction non confirmée qui s'annule par la suite.

Definition C.1 (Éviter les annulations en cascade)**.** L'historique h évite les annulations en cascade si pour chaque lecture $r_i(x_j)$ dans h, c_j précède $r_i(x_j)$ dans h. ACA indique l'ensemble des historiques qui évite les annulations en cascade.

C.3.1.2 Instantanés cohérents et strictement cohérents

Les instantanés cohérents (Consistent Snapshot) et strictement cohérents (Strictly Consistent Snapshots) sont définis en raffinant la causalité en une relation de dépendance.

Definition C.2 (Dépendance)**.** Considérons un historique h et 2 transactions T_i et T_j. Nous notons $T_i \rhd T_j$ quand $r_i(x_j)$ est dans h. La transaction T_i dépend de la transaction T_j quand $T_i \rhd^* T_j$ est vérifié. Les transactions T_i et T_j sont indépendantes si aucune des conditions $T_i \rhd^* T_j$ et $T_j \rhd^* T_i$ n'est vérifiée.

Formellement, les instantanés cohérents sont définis comme suit :

Definition C.3 (Instantané cohérent)**.** Une transaction T_i dans un historique h observe un instantané cohérent si et seulement si, pour chaque objet x, si (i) T_i lit la version x_j, (ii) T_k écrit la version x_k, et (iii) T_i dépend de T_k, alors la version x_k est suivie par la version x_j dans l'ordre de version induit par h ($x_k \ll_h x_j$). Nous écrivons $h \in$ CONS quand toutes les transactions dans h observent un instantané cohérent.

SI requiert qu'une transaction observe l'état confirmé des données à un *point logique* dans le passé. Ce pré-requis est plus fort que l'instantané cohérent. Pour une transaction T_i, cela implique (SCONSa) qu'il existe un snapshot point pour T_i, et (SCONSb) que si une transaction T_i observe les effets d'une transaction T_j, elle doit également observer les effets de toutes les transactions qui précèdent T_j selon une horloge logique. Un historique est appelé strictement cohérent si les conditions SCONSa et SCONSb sont vérifiées.

Definition C.4 (Instantané Strictement cohérent)**.** Les instantanés dans l'historique h sont strictement cohérents quand pour toute transaction confirmée, T_i, T_j, $T_{k\neq j}$ et T_l, les deux propriétés suivantes sont vérifiées :

- $\forall r_i(x_j), r_i(y_l) \in h : r_i(x_j) \not<_h c_l$ (SCONSa)

- $\forall r_i(x_j), r_i(y_l), w_k(x_k) \in h : c_k <_h c_l \Rightarrow c_k <_h c_j$ (SCONSb)

Nous notons SCONS l'ensemble des historiques strictement cohérents.

C.3.1.3 Instantané monotone

De plus, SI exige ce que nous appelons *Instantanés monotone* (Monotonic Snapshots).

Par exemple, bien que l'historique h_5 ci-dessous satisfasse SCONS, cet historique n'appartient pas à SI. En fait, puisque T_a lit $\{x_0, y_2\}$, et T_b lit $\{x_1, y_0\}$, il n'y a pas historique étendu qui pourrait garantir la Règle de Lecture de SI.

$$h_5 = r_a(x_0) \longrightarrow r_1(x_0).w_1(x_1).c_1 \longrightarrow r_b(x_1).c_b$$
$$r_b(y_0) \longrightarrow r_2(y_0).w_2(y_2).c_2 \longrightarrow r_a(y_2).c_a$$

Bien que SI requiert des instantanés monotoniques, la raison sous-jacente est si complexe que des travaux précédents [25, par exemple] ne garantissent pas cette propriété, bien que se revendiquant d'être SI. Ci-dessous, nous introduisons une relation d'ordre entre les instantanés pour formaliser les instantanés monotoniques.

Definition C.5 (Précédence des instantanés)**.** Soit un historique h et 2 transactions distinctes T_i et T_j. L'instantané lu par T_i précède l'instantané lu par T_j dans l'historique h, noté $T_i \rightarrow T_j$, quand $r_i(x_k)$ et $r_j(y_l)$ appartiennent à h et soit (i) $r_i(x_k) <_h c_l$ est vérifiée, ou (ii) la transaction T_l lit x et $c_k <_h c_l$ est vérifiée.

C.3.2 Write-Conflict Freedom

La règle D2 de SI interdit à deux transactions concurrentes en conflit d'écritures de confirmer toutes les deux. Étant donné que dans notre modèle, nous supposons que chaque écriture est précédée par une lecture équivalente sur le même objet, chaque transaction de mise-à-jour dépend d'une transaction de mise-à-jour précédente (ou sur la transaction initiale T_0). Par conséquent, sous la propriété SI, les transactions concurrentes en conflit doivent être indépendantes :

Definition C.6 (Write-Conflict Freedom)**.** Un historique h est write-conflict free si deux transactions indépendantes n'écrivent jamais sur le même objet. Nous dénotons par WCF les historiques qui satisfassent cette propriété.

C.3.3 La décomposition

Theorem 4.1 ci-dessous établit qu'un historique h est dans SI si et seulement si (1) chaque transaction dans h voit un état confirmé, (2) chaque transaction dans h observe un instantané

strictement cohérent, (3) les instantanés sont monotones, et (4) h est write-conflict free. Une preuve détaillée est disponible dans Appendix A.

Theorem C.1. $SI = ACA \cap SCONS \cap MON \cap WCF$

C.3.4 L'impossibilité de SI avec GPR

Cette section tire profit de notre précédent résultat de décomposition pour montrer que SI ne peut intrinsèquement pas passer à l'échelle. De façon plus détaillée, nous montrons qu'aucune des prpriétés MON, SCONSa ou SCONSb n'est atteignable dans un système GPR asynchrone sans fautes Π quand les mises-à-jour sont obstruction-free et les requêtes sont wait-free.

Theorem C.2. *Aucun système asynchrone sans fautes GPR n'implémente MON.*

Theorem C.3. *Aucun système asynchrone sans fautes GPR n'implémente SCONSb.*

Theorem C.4. *Aucun système asynchrone sans fautes GPR n'implémente SCONSa.*

En conséquence de ce qui précède, aucun système asynchrone, même s'il est sans fautes, peut garantir à la fois GPR et SI. En particulier, même si le système est augmenté avec des détecteurs de fautes [33], une approche classique pour modéliser un synchronisme partiel, SI ne peut pas être implémenter sous GPR. Ce fait entrave fortement l'utilisation de SI à grande échelle.

C.4 Non-monotonic Snapshot Isolation

Dans le chapitre précédent, nous avons montré que GPR ne peut pas été assuré par aucun critère de cohérence forte. Dans ce chapitre, nous présentons un nouvel critère de cohérence, nommé Non-monotone Snapshot Isolation (NMSI), qui satisfait les propriétés de sécurité forte et s'assure les propriétés d'extensibilité suivant :

(i) WFQ, (ii) GPR, (iii) la fraîcheur avant (Forward Freshness) et (iv) la synchronisation avec l'engagement minimal (Minimal Commitment Synchronization).

NMSI est le critère du moins de cohérence qui satisfait les propriétés de sécurité forte: il ne permet pas la réplique divergente, et les transactions sous NMSI sont en dessus de la hiérarchie de Herlihy[69].

En gros, en dessous de NMSI, les transactions doivent lire les versions commis, et prendre les clichés cohérents.

En conséquence, contrairement à SI et PSI, NMSI n'est pas obligé de respecter les contraintes pour les clichés cohérents (i.e., SCONSa ou SCONSb).

De plus, deux transactions qui ont des conflits d'écritures concomitants ne peuvent pas être commis ensemble. Plus officiellement :

Definition C.7 (Non-monotone Cliché Isolation (NMSI))**.** Une histoire h est membre du NMSI si et seulement si h est membre de $\mathrm{ACA} \cap \mathrm{CONS} \cap \mathrm{WCF}$.

Anomalies Applicatives Table 5.1(a) compare NMSI avec les autres critères en fonction des anomalies que une application peut observer. L'écriture faussée (write-skew), l'anomalie classique de SI, peut être observé sous NMSI. (Cahill et al. [29] a montré comment une application peut l'éviter facilement). La violation en temps réel arrive quand une transaction T_i observe les effets de certain transaction T_j, en même temps n'observe pas les effets de toutes les transactions qui précédent T_j. L'anomalie arrive sous le contrainte de la sérialisabilité. En réalité, elle ne pose pas un problème. Sous NMSI, une application doit observer des clichés non-monotones. L'anomalie arrive également sous US et PSI. Selon Garcia-Molina and Wiederhold [57], nous croyons que c'est un petit prix à payer pour améliorer la performance.

Propriétés d'extensibilité Dans le Table 5.1(b), nous montrons les propriétés d'extensibilité pour chaque critère. Pour une comparaison juste, nous considérons l'implémentation non-triviale

	SSER	SER	US	SI	PSI	NMSI	RC
Dirty Reads	x	x	x	x	x	x	x
Non-Repeat. Reads	x	x	x	x	x	x	-
Read Skew	x	x	x	x	x	x	-
Dirty Writes	x	x	x	x	x	x	x
Lost Updates	x	x	x	x	x	x	-
Write Skew	x	x	x	-	-	-	-
Non-monotonic Snapshot among R-O txns	x	x	x	-	-	-	-
Non-monotonic Snapshot among R-O and UP txns	x	x	-	-	-	-	-
Real-time Violation	x	-	-	-	-	-	-

(a) *Anomalies pas permis*

	SSER	SER	US	SI	PSI	NMSI	RC
Genuine Partial Replication	x	x	-	x	x	-	-
Forward Freshness Snapshot	-	-	-	x	x	-	-
Min. Commitment Synchronization	x	x	x	-	-	-	-

(b) *Propriétés de passage à l'échelle*

Table C.1: Comparaison des critères de consistance

pour chaque critère, *i.e.* l'implémentation qui guaranti les mis-à-jour obstruction-free et donc accepte les histoires fraîches positivement. Nous rappelons que une histoire est fraîche positivement quand cheque transaction est capable d'observer *au moins* le cliché le plus récent du système avant elle commence.

La requête sans-attendre est une propriété cruciale, car la plupart de la charge de travail se constitue une grande proportion des transactions lecture-seulement. En conséquence, nous la prenons en compte dans le Table 5.1(b). Dans le chapitre précédent, nous avons montré que aucune entre SSER, SER, SI et PSI est réalisable sous le contrainte GPR quand les requêtes sont sans-attendre et les transactions mis-à-jours sont obstruction-free. Peluso et al. [106] a montré qu'on peut combiner can combine GPR et les requêtes sans-attendre sous US. De plus, NMSI peut aussi satisfaire les deux propriétés conjointement. Comme nous avons montré dans le chapitre Chapter 4, PSI et SI s'appuient sur la fraîcheur base (base freshness) qui ne peut pas coexister avec la fraîcheur avant (forward freshness). Pour éviter les anomalies de l'écriture faussée, SSER, SER, et US sont obligés de certifier les transactions mis-à-jour en fonction des conflits lecture-écriture et écriture-lecture. Par conséquent, ils ne permettent pas la synchronisation des

engagements minimals.

Nous présentons Jessy équipé de la propriété de progrès NTU (nommé Jessy). En Section 5.3, nous augmentons Jessy afin de le permetter les mis-à-jours obstruction-free, (nommé Jessy^{ofu}).

C.5 Generic Deferred Update Replication

Dans les chapitres précédents, nous avons introduit plusieurs critères de cohérence ainsi que les protocoles qui permettent de les garantir, et nous les avons comparé entre eux. Cependant, il est toujours difficile de comprendre quelles sont les différences les plus importantes et de comparer de façon scientifique leur comportement réel.

Dans ce chapitre, nous proposons une nouvelle approche : notre intuition est que les protocoles de *Deferred Update Replication* ou DUR partagent une structure commune, et diffèrent uniquement par leurs instanciations spécifiques de quelques fonctions génériques (e.g, [12, 100, 103, 104, 106, 117, 125, 127, 129–131, 136]). Par exemple, ils ont tous une phase de lecture qui diffère par le choix de la version de l'objet à lire ; et une phase de terminaison, qui diffère uniquement par la façon de détecter et résoudre les problèmes de concurrence.

Nous exprimons cette idée par une structure algorithmique commune, avec des points de réalisation clairement identifiés. Cette structure générique est instanciée en un protocole spécifique en sélectionnant les modules d'extension (*plug-ins*) adéquats depuis une bibliothèque. Par exemple, pour un protocole sérializable, le module d'extension de lecture va choisir la plus récente version validée de l'objet, et le module d'extension de terminaison annulera toute transaction si elle est concurrente avec une autre transaction qui a déjà été validée.

L'objectif principal de DUR est de fournir des clients avec une abstraction pour le stockage de données transactionnelles. Sous le capot, ce support de stockage est distribué et répliqué à travers de multiples répliques. Les répliques sont synchronisés pour offrir aux clients un accès disponible et cohérent aux données.

Generic Deferred Update Replication ou G-DUR est conçu comme une implémentation générique et adaptable de DUR. Figure 6.1 présente l'architecture globale de l'intergiciel. Les clients envoient leurs transactions aux instances G-DUR. Un client peut exécuter une transactions de façon interactive, i.e., G-DUR ne requiert pas que tout le code d'une transaction soit soumis d'un coup. Cependant, certaines optimisations ne sont possibles que dans ces cas là. Une transaction commence par une opération *begin*, suivie par une ou plusieurs opérations CRUD (i.e., Create, Read, Update ou Delete), et termine par une instruction *commit* ou *abort*. Les opérations create, update et delete sont implémentées comme des opérations d'écriture. Par la suite, nous nous contenterons de faire allusion aux opérations de lecture et d'écriture.

Chaque instance G-DUR *coordonne* les requêtes transactionnelles reçues depuis un client.

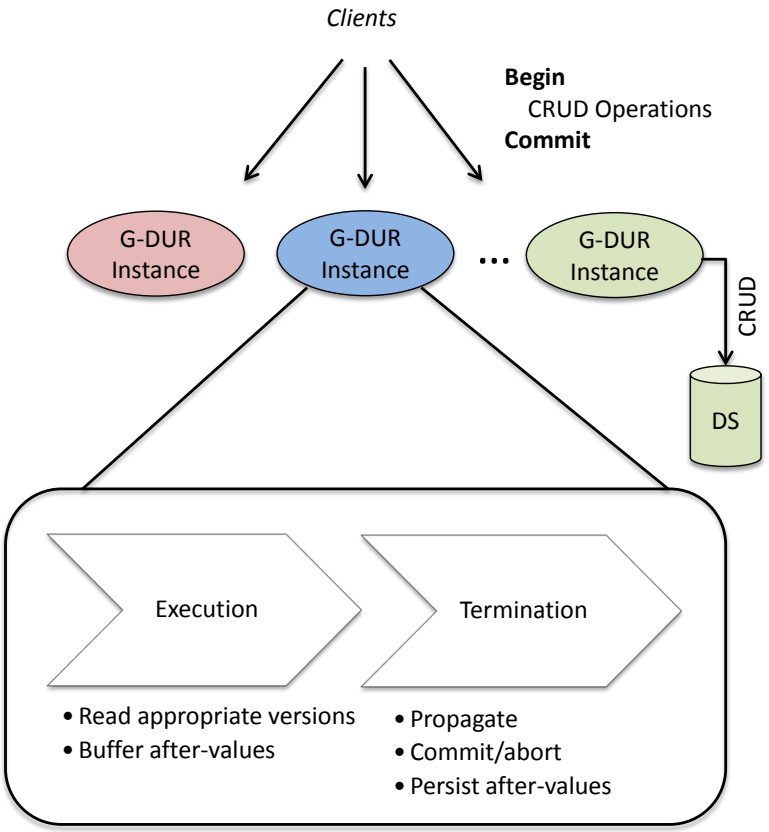


Figure C.1: Architecture G-DUR

Ainsi, une instance stocke localement un sous-ensemble des données disponibles globalement, et exécute deux protocoles d'*exécution* et de *terminaison* personnalisables (voir le bas de Figure 6.1). Le protocole d'exécution est responsable de la lecture des données et de leur mise en tampon. Le protocole de terminaison s'occupe de la propagation des effets de bord de la transaction, de sa validation et de la persistance des données.

Nous démontrons son potentiel à travers d'intensives expérimentations que nous avons conduit dans un environnement géo-répliqué.

(1) Nous avons conçu G-DUR pour implémenter six protocoles transactionnels importants [106, 117, 127, 129, 131, 136]. L'implémentation de chaque protocole dans G-DUR nécessite seulement 200 à 600 lignes de code. Nous avons également évalué de façon empirique chacun de ces protocoles. Nos comparaisons montrent clairement la différence entre ces protocoles, et entre les différents critères de cohérence qu'ils implémentent. De plus, notre étude montre qu'ils

ont des domaines de performances spécifiques ce qui nous permet d'identifier leurs limitations respectives.

(2) Nous montrons comment un développeur peut utiliser G-DUR pour comprendre les limitations d'un protocole. Nous avons utilisé un protocole récemment publié [106], et identifié ses limitations en remplaçant méthodiquement ses modules d'extension par d'autres plus faibles.

(3) La précédente approche permet aussi à un développeur d'améliorer les protocoles existants. Nous illustrons cet aspect en présentant une variation de P-Store [127] qui tire profit de la localité du workload pour être jusqu'à 70% plus rapide que le protocole original.

(4) Dans nos dernières expérimentations, nous évaluons le coût de plusieurs degrés de fiabilité. Pour cela, nous prenons un protocole qui garanti la sérializabilité et nous étudions le coût de la tolérance au fautes en faisant varier le degré de réplication et l'algorithme utilisé pendant les commit.

C.6 Un Système de Stockage Cloud Auto-Configurable

Les systèmes de stockage dans le cloud peuvent satisfaire mieux les nécessitées de leur applications en sélectionnant quand et où les données sont répliquées. Les applications Internet ont souvent des besoins incompatibles. D'un cote, ils doivent être rapides et avec une grande disponibilité. D'autre cote, les répliques doivent rester synchronisée pour maintenir la garantie de consistance et pour fournir une illusion de sérialisabilité. Au lieu de favoriser un besoin sur les autres (e.g., [38, 90, 91]) ou de choisir un compromis en développant l'application (e.g., SimpleDB [11] de Amazon, ou PNUTs [36] de Yahoo!), un modèle émergent est de laisser le système de stockage dans le cloud choisir pendant l'exécution un niveau de service appropriée [140]. Bien que ce abord évite aux application de rester figées dans un compromis fixe entre la consistance et la latence, il ignore des décisions de configuration importantes.

Les défies de configuration qui n'ont pas encore trouvé une réponse et doivent être abordé par les systèmes de stockage dans le cloud comprennent: (i) ou mettre la réplique principal et secondaire (ii) quant bien des répliques secondaires sont désirées, et (iii) avec quelle fréquence les répliques secondaires doivent se synchroniser avec la réplique primaire. Ces défis sont exacerbés par le fait que les utilisateurs Internet sont localises à des endroits géographiques différents avec des différents fuseau horaires et modèle d'accès. En outre, les systèmes doivent considérer le croissantes contraintes légales, de sécurité et de coût concernant la réplication dans certain pais et d'éviter de répliquer dans des outres.

Pour une communauté d'utilisateurs stable, une configuration statique faite par un administrateur de système peut être acceptable. Mais beaucoup d'applications modernes, comme magasinage, réseaux sociaux, journaux, et jeux vidéo, n'ont pas seulement des utilisateurs en évolution à échelle mondiale mais aussi ressentent des modèles d'accès qui changent dans le temps, sur base journalière ou saisonnière. Par conséquent, il est avantageux pour le système de stockage d'adapter *automatiquement* sa configuration soumise à des contraintes spécifique à l'application et geo-politiques.

Tuba est un base de données clé-valeur basé sur [140]. Il s'occupe des défis susmentionnée en configurant le système de stockage dans le cloud automatiquement et périodiquement. En plus de maximiser l'utilité des opérations en écriture, Tuba améliore aussi l'utilité totale du système de stockage en s'adaptant automatiquement au changes dans les pattern d'accès aux données et des contraintes. À cette fin, Tuba inclue un service de configuration qui reçoit périodiquement

par le client, avec les rapport de "hit" et de "miss", des accords de niveau de service (SLAs, pour service level agreements) bases sur la consistance. Le service change par conséquence du SLA l'emplacement des répliques primaire et secondaires pour améliorer l'utilité globale du système. La nouveauté clef de Tuba est que les opérations en lecture ainsi que les opérations en écriture peuvent être exécutées en parallèle aux opérations de reconfiguration.

Nous avons implémenté Tuba comme un intergiciel au dessus de Microsoft Azure Storage (MAS) [30]. MAS fournis une consistance forte et réalise de la geo-réplication pour être tolérant aux désastres. Puisque Tuba est basé sur Pileus, il fournis diffèrent choix de consistance exprimées à travers les SLAs. En outre, Tuba rentabilise la geo-réplication pour améliorer la localité et la disponibilité. Notre interface de programmation (souvent désignée par le terme API pour l'anglais Application Programming Interface) est une extension mineur de l'interface de programmation de MAS Blob Store, par conséquent les applications Azure existante peuvent bénéficier de la reconfiguration dynamique de Tuba avec peu de changements.

Un expérimentation avec des clients repartis dans de centres de calculs dans le monde entier montrent que avec une réconfiguration toute les deux heures augmente la portion de lectures que garantissent une consistance forte du 33% au 54%. Ceci confirme que la reconfiguration automatique peut produire des bénéficies substantiels que sont réalisables en pratique.

BIBLIOGRAPHY

[1] Daniel J. Abadi. Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. *Computer*, 45(2):37–42, February 2012.

[2] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

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

[4] Atul Adya, B Liskov, and Patrick O'Neil. Generalized isolation level definitions. In *Int. Conf. on Data Engineering (ICDE)*, number March, pages 67–78. IEEE Comput. Soc, 2000.

[5] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40:873–890, 1993.

[6] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Harbinder Bhogan. Volley: automated data placement for geo-distributed cloud services. In *Networked Sys. Design and Implem. (NSDI)*, page 2. USENIX Association, April 2010.

[7] Marcos K. Aguilera, Wojciech Golab, and Mehul A. Shah. A practical scalable distributed B-tree. *Int. Conf. on Very Large Data Bases (VLDB)*, 1(1):598–609, August 2008.

[8] JoséBacelar Almeida, PauloSérgio Almeida, and Carlos Baquero. Bounded Version Vectors. In Rachid Guerraoui, editor, *Distributed Computing*, volume 3274 of *Lecture Notes in Computer Science*, pages 102–116. Springer Berlin Heidelberg, 2004.

[9] Sérgio Almeida, João Leitão, and Luis Rodrigues. ChainReaction. In *Euro. Conf. on Comp. Sys. (EuroSys)*, page 85, New York, New York, USA, April 2013. ACM Press.

[10] Gustavo Alonso. Partial Database Replication and Group Communication Primitives (Extended Abstract). In *European Research Seminar on Advances in Distributed Systems (ERSADS)*, pages 171—-176, 1997.

[11] Amazon Web Services. Amazon SimpleDB. URL `https://aws.amazon.com/simpledb/`.

[12] Yair Amir and Ciprian Tutu. From total order to database replication. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*, pages 494–503, Washington, DC, USA, 2002. IEEE.

[13] Khalil Amiri, David Petrou, Gregory R. Ganger, and Garth A. Gibson. Dynamic function placement for data-intensive cluster computing. In *Usenix Annual Tech. Conf. (Usenix-ATC)*. USENIX Association, June 2000.

[14] Peter M. G. Apers. Data allocation in distributed database systems. *Trans. on Database Sys.*, 13:263—-304, 1988.

[15] J. E. Armendáriz-Iñigo, A. Mauch-Goya, J. R. González de Mendívil, and F. D. Muñoz Escoí. SIPRe: a partial database replication protocol with SI replicas. In *Symp. on Applied Computing (SAC)*, pages 2181—-2185, New York, New York, USA, 2008. ACM Press.

[16] Hagit Attiya, Eshcar Hillel, and Alessia Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. In *Symp. on Parallelism in Algorithms and Architectures (SPAA)*, page 69, New York, New York, USA, August 2009. ACM Press.

[17] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Highly Available Transactions : Virtues and Limitations. In *Int. Conf. on Very Large Data Bases (VLDB)*, 2014.

[18] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Scalable atomic visibility with RAMP Transactions. In *Int. Conf. on the Mgt. of Data (SIGMOD)*, 2014.

[19] Roberto Baldoni, Francesco Quaglia, and Michel Raynal. Distributed Database Checkpointing. In *Euro. Conf. on Parallel and Dist. Comp. (Euro-Par)*, volume 1685, pages 450–458, Berlin, Heidelberg, August 1999. Springer Berlin Heidelberg.

[20] Roberto Baldoni, Giacomo Cioffi, Jean-Michel Hélary, and Michel Raynal. Direct dependency-based determination of consistent global checkpoints. *Comput. Syst. Sci. Eng.*, 16(1):43–49, 2001.

[21] Sam Basu, Anindya and Charron-Bost, Bernadette and Toueg. Simulating reliable links with unreliable links in the presence of process crashes. In Keith Babaoğlu, Özalp and Marzullo, editor, *Distributed Algorithms*, Lecture Notes in Computer Science, pages 105–122. Springer Berlin Heidelberg, 1996.

[22] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *Int. Conf. on the Mgt. of Data (SIGMOD)*, pages 1–10, New York, New York, USA, 1995. ACM Press.

[23] JosepM. Bernabé-Gisbert, Raúl Salinas-Monteagudo, Luis Irún-Briz, and FrancescD. Muñoz Escoí. Managing Multiple Isolation Levels in Middleware Database Replication Protocols. In Minyi Guo, LaurenceT. Yang, Beniamino Martino, HansP. Zima, Jack Dongarra, and Feilong Tang, editors, *Parallel and Distributed Processing and Applications*, volume 4330 of *Lecture Notes in Comp. Sc.*, pages 511–523. Springer, 2006.

[24] Philip Bernstein, Vassos Radzilacos, and Vassos Hadzilacos. *Concurrency Control and Recovery in Database Systems*. Addison Wesley Publishing Company, 1987.

[25] Annette Bieniusa and Thomas Fuhrmann. Consistency in hindsight: A fully decentralized STM algorithm. In *Int. Parallel Dist. Processing Symposium (IPDPS)*, pages 1–12. IEEE, 2010.

[26] Ken Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *Trans. on Computer Sys.*, 5(1):47–76, January 1987.

[27] Ken Birman, Gregory Chockler, and Robbert van Renesse. Toward a cloud computing research agenda. *ACM SIGACT News*, 40(2):68, June 2009.

[28] Mike Blasgen, Jim Gray, Mike Mitoma, and Tom Price. The convoy phenomenon. *Operating Systems Review*, 13(2):20–25, April 1979.

[29] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable isolation for snapshot databases. In *Int. Conf. on the Mgt. of Data (SIGMOD)*, page 729, New York, New York, USA, June 2008. ACM Press.

[30] Brad Calder, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Ju Wang, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, Leonidas Rigas, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, and Jiesheng Wu. Windows Azure Storage. In *Symp. on Op. Sys. Principles (SOSP)*, pages 143—-157, New York, New York, USA, October 2011. ACM Press.

[31] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepole. C-JDBC: flexible database clustering middleware. In *Usenix Annual Tech. Conf. (Usenix-ATC)*, page 26, Boston, MA, June 2004. USENIX Association.

[32] A. Chan and R. Gray. Implementing Distributed Read-Only Transactions. *IEEE Transactions on Software Engineering*, SE-11(2):205–212, February 1985.

[33] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[34] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *Trans. on Computer Sys.*, 26(2):1–26, June 2008.

[35] Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors. *Replication*, volume 5959 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, 2010.

[36] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[37] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Symp. on Cloud Computing (SoCC)*, pages 143—-154, New York, NY, USA, 2010. ACM.

[38] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J J Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, pages 251—-264, Hollywood, CA, USA, 2012. USENIX Association.

[39] R. Correia, A., Jr. and Pereira, J. and Oliveira. AKARA: A Flexible Clustering Protocol for Demanding Transactional Workloads. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems: OTM 2008*, volume 5331 of *Lecture Notes in Computer Science*, pages 691–708. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[40] Cedric Coulon, Esther Pacitti, and Patrick Valduriez. Consistency Management for Partial Replication in a High Performance Database Cluster. In *Int. Conf. on Parallel and Dist. Sys. (ICPADS)*, volume 1, pages 809–815. IEEE, July 2005.

[41] James Cowling and Barbara Liskov. Granola: Low-Overhead Distributed Transaction Coordination. In *Usenix Annual Tech. Conf. (Usenix-ATC)*, Boston, MA, USA, June 2012. USENIX Association.

[42] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-Store: A Scalable Data Store for Transactional Multi key Access in the Cloud. In *Symp. on Cloud Computing (SoCC)*, page 163, New York, New York, USA, June 2010. ACM Press.

[43] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with snapshot isolation. In *Int. Conf. on Very Large Data Bases (VLDB)*, pages 715–726. VLDB Endowment, 2006.

[44] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, December 2004.

[45] Ricardo J Dias, João M Lourenço, and Nuno Preguiça. Efficient and Correct Transactional Memory Programs Combining Snapshot Isolation and Static Analysis. In *W. on Hot Topics in Parallelism (HotPar)*, 2011.

[46] Nuno Diegues and Paolo Romano. STI-BT : A Scalable Transactional Index. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*, number September, pages 104—-113, 2014.

[47] Danny Dolev and Christoph Lenzen. Early-deciding consensus is expensive. In *Symp. on Principles of Dist. Comp. (PODC)*, page 270, New York, New York, USA, July 2013. ACM Press.

[48] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. Clock-SI: Snapshot Isolation for Partitioned Data Stores Using Loosely Synchronized Clocks. In *Symp. on Reliable Dist. Sys. (SRDS)*, pages 173–184. IEEE, 2013.

[49] Sameh Elnikety, W. Zwaenepoel, and Fernando Pedone. Database Replication Using Generalized Snapshot Isolation. In *Symp. on Reliable Dist. Sys. (SRDS)*, pages 73–84. IEEE, October 2005.

[50] Alan Fekete. Allocating isolation levels to transactions. In *Symp. on Principles of Dist. Comp. (PODC)*, pages 206—-215, New York, New York, USA, June 2005. ACM Press.

[51] Alan D. Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *Trans. on Database Sys.*, 30(2):492–528, June 2005.

[52] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-Based Software Transactional Memory. *IEEE Trans. on Parallel and Dist. Sys. (TPDS)*, 21(12):1793–1807, December 2010.

[53] FaithEllen Fich. How Hard Is It to Take a Snapshot? In Ondrej Vojtáš, Peter and Bieliková, Mária and Charron-Bost, Bernadette and Sýkora, editor, *SOFSEM 2005: Theory and Practice of Computer Science*, Lecture Notes in Computer Science, pages 28–37. Springer Berlin Heidelberg, 2005.

[54] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[55] Udo Fritzke and Philippe Ingels. Transactions on partially replicated data based on reliable and atomic multicasts. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*, pages 284–291. IEEE Comput. Soc, 2001.

[56] Udo Fritzke, Philippe Ingels, Achour Mostéfaoui, and Michel Raynal. Fault-tolerant Total Order Multicast to asynchronous groups. In *Symp. on Reliable Dist. Sys. (SRDS)*, pages 228–234. IEEE, 1998.

[57] Hector Garcia-Molina and Gio Wiederhold. Read-only transactions in a distributed database. *Trans. on Database Sys.*, 7(2):209–234, June 1982.

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

[59] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, Lecture Notes in Comp. Sc., pages 393–481. Springer, 1978.

[60] Jim Gray and Leslie Lamport. Consensus on transaction commit. *Trans. on Database Sys.*, 31(1):133–160, March 2006.

[61] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. *ACM SIGMOD Record*, 25(2):173–182, 1996.

[62] Grid'5000. Grid'5000, a scientific instrument designed to support experiment-driven research in all areas of computer science related to parallel, large-scale or distributed computing and networking. \url{https://www.grid5000.fr/}, 2013.

[63] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Symp. on Principles and Practice of Parallel Prog. (PPoPP)*, pages 175–184, New York, NY, USA, 2008. ACM.

[64] Rachid Guerraoui and Michal Kapalka. On obstruction-free transactions. In *Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 304—-313, New York, New York, USA, June 2008. ACM Press.

[65] Rachid Guerraoui and Michal Kapalka. *Principles of Transactional Memory*. Morgan and Claypool Publishers, 2010.

[66] Rachid Guerraoui and André Schiper. Genuine atomic multicast in asynchronous distributed systems. *Theoretical Computer Science*, 254(1-2):297–316, March 2001.

[67] Rachid Guerraoui, Thomas Henzinger, and Vasu Singh. Permissiveness in Transactional Memories. In *Int. Symp. on Dist. Comp. (DISC)*, pages 305–319, Berlin, Heidelberg, 2008. Springer.

[68] R C Hansdah and Lalit M. Patnaik. Update serializability in locking. In Giorgio Ausiello and Paolo Atzeni, editors, *Lecture Notes in Comp. Sc.*, volume 243 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 1986.

[69] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[70] Maurice Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3): 463—-492, 1990.

[71] JoAnne Holliday, Divyakant Agrawal, and Amr El Abbadi. Partial database replication using epidemic communication. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*, pages 485–493. IEEE, 2002.

[72] Galen C. Hunt and Michael L. Scott. The Coign automatic distributed partitioning system. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, pages 187–200. USENIX Association, February 1999.

[73] Damien Imbs and Michel Raynal. Virtual world consistency: A condition for STM systems (with a versatile protocol with invisible read operations). *Theoretical Computer Science*, 444:113–127, July 2012.

[74] Ricardo Jiménez-Peris, Marta Patino-Martinez, Bettina Kemme, and Gustavo Alonso. Improving the scalability of fault-tolerant database clusters. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*, pages 477–484. IEEE Comput. Soc, 2002.

[75] Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S. Sudarshan. Automating the detection of snapshot isolation anomalies. In *Int. Conf. on Very Large Data Bases (VLDB)*, VLDB '07, pages 1263–1274. VLDB Endowment, September 2007.

[76] Sudarshan Kadambi, Jianjun Chen, Brian F. Cooper, David Lomax, Adam Silberstein, Erwin Tam, and Hector Garcia-molina. Where in the World is My Data ? In *Int. Conf. on Very Large Data Bases (VLDB)*, pages 1040–1050, 2011.

[77] Bettina Kemme and Gustavo Alonso. A new approach to developing and implementing eager database replication protocols. *Trans. on Database Sys.*, 25(3):333–379, September 2000.

[78] Bettina Kemme and Gustavo Alonso. Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. In *Int. Conf. on Very Large Data Bases (VLDB)*, VLDB '00, pages 134–143, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[79] L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, SE-3, 1977.

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

[81] Leslie Lamport. The part-time parliament. *Trans. on Computer Sys.*, 16(2):133–169, 1998.

[82] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, July 2006.

[83] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguic, and Rodrigo Rodrigues. Making Geo-Replicated Systems Fast as Possible , Consistent when Necessary. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, volume abs/1203.6, 2012.

[84] Geoffrey M. Voelker Lili Qiu, Venkata N. Padmanabhan. On the placement of web server replicas. In *Int. Conf. on Computer Communications (INFOCOM)*, pages 1587—-1596, 2001.

[85] Jun-Lin Lin and Margaret H. Dunham. A Survey of Distributed Database Checkpointing. *Distributed and Parallel Databases*, 5:289–319, 1997.

[86] Kwei-Jay Lin. Consistency issues in real-time database systems. In *Annual Hawaii International Conference on System Sciences*, volume 2 of *Volume II: Software Track*, pages 654–661. IEEE Comput. Soc. Press, 1989.

[87] Yi Lin, Bettina Kemme, Marta Patiño Martínez, and Ricardo Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *Int. Conf. on the Mgt. of Data (SIGMOD)*, page 419, New York, New York, USA, 2005. ACM Press.

[88] Yi Lin, Bettina Kemme, Marta Patiño Martínez, and Ricardo Jiménez-Peris. Consistent Data Replication: Is It Feasible in WANs? In *Euro. Conf. on Parallel and Dist. Comp. (Euro-Par)*, volume 3648, pages 633–643, 2005.

[89] Heiner Litz, David Cheriton, Amin Firoozshahian, Omid Azizi, and John P. Stevenson. SI-TM: reducing transactional memory abort rates through snapshot isolation. In *Int. Conf. on ArchiṠupport for Prog. Lang. and Systems (ASPLOS)*, pages 383–398, New York, New York, USA, February 2014. ACM Press.

[90] 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 *Symp. on Op. Sys. Principles (SOSP)*, pages 401—-416, New York, NY, USA, 2011. ACM.

[91] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *Networked Sys. Design and Implem. (NSDI)*, pages 1–14, Lombard, IL, USA, 2013.

[92] Rajmohan Rajaraman Madhukar R. Korupolu, C. Greg Plaxton. Placement Algorithms for Hierarchical Cooperative Caching. pages 586–595. Society for Industrial and Applied Mathematics, 1999.

[93] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring Paxos: A high-throughput atomic broadcast protocol. In *Int. Conf. on Dependable Sys. and Networks (DSN)*, pages 527–536. IEEE, June 2010.

[94] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. *Parallel and Distributed Algorithms*, pages 215–226, 1989.

[95] Microsoft Corporation. Transact-SQL Reference, 2014. URL `http://msdn.microsoft.com/en-us/library/ms173763.aspx`.

[96] Matthias Nicola and Matthias Jarke. Performance modeling of distributed and replicated databases. *IEEE Transactions on Knowledge and Data Engineering*, 12(4):645–672, 2000.

[97] Esther Pacitti, Cédric Coulon, Patrick Valduriez, and M. Tamer Özsu. Preventive Replication in a Database Cluster. *Distributed and Parallel Databases*, 18(3):223–251, November 2005.

[98] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[99] Douglas Stott Parker Jr., Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David A. Edwards, Stephen Kiser, and Charles S. Kline. Detection of Mutual Inconsistency in Distributed Systems. *IEEE Transactions on Software Engineering*, SE-9(3):240–247, May 1983.

[100] Marta Patiño Martínez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. Scalable Replication in Database Clusters. In *Int. Symp. on Dist. Comp. (DISC)*, pages 315–329. Springer, October 2000.

[101] Marta Patiño Martinez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. MIDDLE-R: Consistent database replication at the middleware level. *Trans. on Computer Sys.*, 23(4):375–423, November 2005.

[102] Fernando Pedone, Rachid Guerraoui, and André Schiper. Exploiting atomic broadcast in replicated databases. In Jeff Pritchard, David and Reeve, editor, *Euro. Conf. on Parallel and Dist. Comp. (Euro-Par)*, Lecture Notes in Comp. Sc., pages 513–520. Springer, 1998.

[103] Fernando Pedone, Rachid Guerraoui, and André Schiper. The Database State Machine Approach. *Distributed and Parallel Databases*, 14(1):71–98–98, July 2003.

[104] Sebastiano Peluso, Paolo Romano, and Francesco Quaglia. SCORe: a scalable one-copy serializable partial replication protocol. In *Int. Conf. on Middleware (MIDDLEWARE)*, pages 456–475. Springer, December 2012.

[105] Sebastiano Peluso, Paolo Romano, and Francesco Quaglia. Genuine replication, opacity and wait-free read transactions: can a STM get them all? In *W. on the Theory of Transactional Memory (WTTM)*, Madeira, Portugal, July 2012.

[106] Sebastiano Peluso, Pedro Ruivo, Paolo Romano, Francesco Quaglia, and Luis Rodrigues. When Scalability Meets Consistency: Genuine Multiversion Update-Serializable Partial Data Replication. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*, pages 455–465, Macau, China, 2012. IEEE.

[107] Sebastiano Peluso, Roberto Palmieri, Paolo Romano, Inesc-id Ist, and Francesco Quaglia. On Breaching the Wall of Impossibility Results on Disjoint-Access Parallel STM. Technical report, Virginia Tech, 2014.

[108] PostgreSQL. PostgreSQL Documentation, 2014. URL `http://www.postgresql.org/docs/9.4/static/transaction-iso.html`.

[109] Michel Raynal, G. Thia-Kime, and Mustaque Ahamad. From serializable to causal transactions for collaborative applications. In *EUROMICRO Conference: New Frontiers of Information Technology*, pages 314–321. IEEE, 1997.

[110] Richard Strohm. Oracle Database Concepts, 11g Release 1 (11.1), January 2011.

[111] Torvald Riegel, Christof Fetzer, and Pascal Felber. Snapshot isolation for software transactional memory. In *first ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[112] Danny Dolev Roman Vitenberg, Idit Keidar, Gregory V. Chockler. Group Communication Specifications: A Comprehensive Study. *ACM Computing Surveys*, 33, 1999.

[113] Masoud Saeida Ardekani and Douglas B Terry. A Self-Configurable Geo-Replicated Cloud Storage System. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, pages 367—-381, Broomfield, CO, October 2014. USENIX Association.

[114] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. The Impossibility of Ensuring Snapshot Isolation in Genuine Replicated STMs. In *W. on the Theory of Transactional Memory (WTTM)*, Rome, Italy, September 2011.

[115] Masoud Saeida Ardekani, Marek Zawirski, Pierre Sutra, and Marc Shapiro. The space complexity of transactional interactive reads. In *Int. W. on Hot Topics in Cloud Data Processing (HotCDP)*, pages 1–5, New York, New York, USA, April 2012. ACM Press.

[116] Masoud Saeida Ardekani, Pierre Sutra, and Pierpaolo Cincilla. https://github.com/msaeida/jessy, 2013. URL `https://github.com/msaeida/jessy`.

[117] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. Non-Monotonic Snapshot Isolation: scalable and strong consistency for geo-replicated transactional systems. In *Symp. on Reliable Dist. Sys. (SRDS)*, pages 163–172. IEEE, October 2013.

[118] Masoud Saeida Ardekani, Pierre Sutra, Marc Shapiro, and Nuno Preguiça. On the Scalability of Snapshot Isolation. In Felix Wolf, Bernd Mohr, and Dieter Mey, editors, *Euro. Conf. on Parallel and Dist. Comp. (Euro-Par)*, volume 8097, pages 369–381, Aachen, Germany, August 2013.

[119] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. G-DUR: A Middleware for Assembling, Analyzing, and Improving Transactional Protocols. In *Int. Conf. on Middleware (MIDDLEWARE)*, Bordeaux, France, December 2014.

[120] SAP. SAP HANA SQL and System Views References, 2014. URL `https://help.sap.com/saphelp_hanaone/helpdata/en/20/fdf9cb75191014b85aaa9dec841291/content.htm`.

[121] Florian Schintke, Alexander Reinefeld, Seif Haridi, and Thorsten Schütt. Enhanced Paxos Commit for Transactions on DHTs. In *Int. Conf. on Cluster, Cloud and Grid Computing (CCGrid)*, pages 448–454, Washington, DC, USA, May 2010. IEEE.

[122] Nicolas Schiper. *On Multicast Primitives in Large Networks and Partial Replication Protocols presented by*. PhD thesis, University of Lugano, 2009.

[123] Nicolas Schiper and Fernando Pedone. On the Inherent Cost of Atomic Broadcast and Multicast Algorithms in Wide Area Networks. In *Int. Conf. on Distributed Comp. and Net. (ICDCN)*, number 4904 in Lecture Notes in Comp. Sc., pages 147–157. Springer, 2008.

[124] Nicolas Schiper, Rodrigo Schmidt, and Fernando Pedone. Brief Announcement: Optimistic Algorithms for Partial Database Replication. In *Int. Symp. on Dist. Comp. (DISC)*, volume 4305 of *Lecture Notes in Comp. Sc.*, pages 557–559. Springer, 2006.

[125] Nicolas Schiper, Rodrigo Schmidt, and Fernando Pedone. Optimistic algorithms for partial database replication. In *Int. Conf. on Principles of Dist. Sys. (OPODIS)*, pages 81–93, Berlin, Heidelberg, December 2006. Springer.

[126] Nicolas Schiper, Pierre Sutra, and Fernando Pedone. Genuine versus Non-Genuine Atomic Multicast Protocols for Wide Area Networks: An Empirical Study. In *Symp. on Reliable Dist. Sys. (SRDS)*, pages 166–175. IEEE, September 2009.

[127] Nicolas Schiper, Pierre Sutra, and Fernando Pedone. P-Store: Genuine Partial Replication in Wide Area Networks. In *Symp. on Reliable Dist. Sys. (SRDS)*, pages 214–224. IEEE, October 2010.

[128] Rodrigo Schmidt and Fernando Pedone. A formal analysis of the deferred update technique. In *Int. Conf. on Principles of Dist. Sys. (OPODIS)*, pages 16–30. Springer, December 2007.

[129] Daniele Sciascia and Fernando Pedone. Scalable Deferred Update Replication. In *Int. Conf. on Dependable Sys. and Networks (DSN)*, pages 1–12. IEEE, 2012.

[130] Daniele Sciascia and Fernando Pedone. Geo-replicated storage with scalable deferred update replication. In *Int. Conf. on Dependable Sys. and Networks (DSN)*. IEEE, 2013.

[131] Damián Serrano, Marta Patiño Martínez, Ricardo Jiménez-Peris, and Bettina Kemme. Boosting Database Replication Scalability through Partial Replication and 1-Copy-Snapshot-Isolation. In *Pacific Rim Int. Symp. on Dependable Comp. (PRDC)*, pages 290–297. IEEE, December 2007.

[132] Damián Serrano, Marta Patiño Martínez, Ricardo Jiménez-Peris, and Bettina Kemme. An Autonomic Approach for Replication of Internet-based Services. In *Symp. on Reliable Dist. Sys. (SRDS)*, pages 127–136. IEEE, October 2008.

[133] Nir Shavit and Dan Touitou. Software transactional memory. In *Symp. on Principles of Dist. Comp. (PODC)*, pages 204–213, New York, New York, USA, August 1995. ACM Press.

[134] Gokul Soundararajan, Cristiana Amza, and Ashvin Goel. Database replication policies for dynamic content applications. In *Euro. Conf. on Comp. Sys. (EuroSys)*, number 4, page 89, New York, New York, USA, October 2006. ACM.

[135] António Sousa, Pinto Ferreira, Fernando Pedone, Rui Oliveira, and Francisco Moura. Partial replication in the Database State Machine. In *Int. Symp. on Network Computing and Applications (NCA)*, pages 298–309, 2001.

[136] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyuan Li. Transactional storage for geo-replicated systems. In *Symp. on Op. Sys. Principles (SOSP)*, pages 385—-400, New York, NY, USA, 2011. ACM.

[137] Michael F Spear, Virendra J Marathe, William N Scherer, and Michael L. Scott. Conflict Detection and Validation Strategies for Software Transactional Memory. In *Int. Symp. on Dist. Comp. (DISC)*, pages 179–193, Berlin, Heidelberg, 2006. Springer.

[138] Pierre Sutra and Marc Shapiro. Fault-Tolerant Partial Replication in Large-Scale Database Systems. In *Euro. Conf. on Parallel and Dist. Comp. (Euro-Par)*, pages 404–413. Springer, 2008.

[139] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Int. Conf. on Para. and Dist. Info. Sys. (PDIS)*, pages 140–149. IEEE, September 1994.

[140] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Symp. on Op. Sys. Principles (SOSP)*, pages 309–324, New York, New York, USA, November 2013. ACM Press.

[141] Alexander Thomson, Thaddeus Diamond, Philip Shao, and Daniel J. Abadi. Calvin : Fast Distributed Transactions for Partitioned Database Systems. In *Int. Conf. on the Mgt. of Data (SIGMOD)*, pages 1–12. ACM, 2012.

[142] Nguyen Tran, Marcos K. Aguilera, and Mahesh Balakrishnan. Online migration for geo-distributed storage systems. In *Usenix Annual Tech. Conf. (Usenix-ATC)*, Berkeley, CA, USA, 2011. USENIX Association.

[143] Arun Venkataramani, Phoebe Weidmann, and Mike Dahlin. Bandwidth constrained placement in a WAN. In *Symp. on Principles of Dist. Comp. (PODC)*, pages 134–143, New York, New York, USA, August 2001. ACM Press.

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

[145] Matthias Wiesmann and André Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE Transactions on Knowledge and Data Engineering*, 17(4):551–566, 2005.

[146] Matthias Wiesmann, Fernando Pedone, André Schiper, Matthias Wiesmann Fern, Bettina Kemme, and Gustavo Alonso. Database Replication Techniques: a Three Parameter Classification. In *Symp. on Reliable Dist. Sys. (SRDS)*, pages 206—-. IEEE, 2000.

[147] Matthias Wiesmann, Fernando Pedone, André Schiper, Bettina Kemme, and Gustavo Alonso. Understanding replication in databases and distributed systems. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*, pages 464–474. IEEE, 2000.

[148] Ouri Wolfson, Sushil Jajodia, and Yixiu Huang. An adaptive data replication algorithm. *Trans. on Database Sys.*, 22(2):255–314, June 1997.

[149] Sai Wu, Dawei Jiang, Beng Chin Ooi, and Kun-Lung Wu. Efficient B-tree based indexing for cloud data processing. *Int. Conf. on Very Large Data Bases (VLDB)*, 3(1-2):1207–1218, September 2010.

[150] Maysam Yabandeh and Daniel Gómez Ferro. A critique of snapshot isolation. In *Euro. Conf. on Comp. Sys. (EuroSys)*, page 155, New York, New York, USA, April 2012. ACM Press.

[151] Marek Zawirski, Annette Bieniusa, Valter Balegas, Sérgio Duarte, Carlos Baquero, Marc Shapiro, and Nuno Preguiça. SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine. Technical Report RR-8347, Inria, August 2013. URL `http://fr.arxiv.org/abs/1310.3107`.

[152] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. Transaction chains. In *Symp. on Op. Sys. Principles (SOSP)*, pages 276–291, New York, New York, USA, November 2013. ACM Press.