Cohérence dans les systèmes de stockage distribués : fondements théoriques avec applications au cloud storage

Paolo Viotti

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2017-ENST-0016

EDITE - ED 130

Doctorat ParisTech
T H È S E
pour obtenir le grade de docteur délivré par

TELECOM ParisTech
Spécialité « INFORMATIQUE et RESEAUX »

présentée et soutenue publiquement par
Paolo VIOTTI
le 6 avril 2017

Consistency in Distributed Storage Systems : Theoretical Foundations with Applications to Cloud Storage

Directeur de thèse : **M. Marko VUKOLIĆ**

Jury

M. Pietro MICHIARDI, Professeur, Data Science Department, EURECOM	Président
M. Nuno NEVES, Professeur, Department of Computer Science, Universidade de Lisboa	Rapporteur
M. Marc SHAPIRO, Directeur de Recherche, UPMC-LIP6 and Inria	Rapporteur
Mme Sonia BEN MOKHTAR, Chercheuse, CNRS and INSA Lyon	Examinatrice
M. Petr KUZNETSOV, Professeur, INFRES, Télécom ParisTech	Examinateur

THÈSE

TELECOM ParisTech
école de l'Institut Télécom - membre de ParisTech

Abstract

Engineering distributed systems is an onerous task: the design goals of performance, correctness and reliability are intertwined in complex tradeoffs, which have been outlined by multiple theoretical results. These tradeoffs have become increasingly important as computing and storage have shifted towards distributed architectures. Additionally, the general lack of systematic approaches to tackle distribution in modern programming tools, has worsened these issues — especially as nowadays most programmers have to take on the challenges of distribution. As a result, there exists an evident divide between programming abstractions, application requirements and storage semantics, which hinders the work of designers and developers.

This thesis presents a set of contributions towards the overarching goal of designing reliable distributed storage systems, by examining these issues through the prism of *consistency*. We begin by providing a uniform, declarative framework to formally define consistency semantics. We use this framework to describe and compare over fifty non-transactional consistency semantics proposed in previous literature. The declarative and composable nature of this framework allows us to build a partial order of consistency models according to their semantic strength. We show the practical benefits of composability by designing and implementing Hybris, a storage system that leverages different models and semantics to improve over the weak consistency generally offered by public cloud storage platforms. We demonstrate Hybris' efficiency and show that it can tolerate arbitrary faults of cloud stores at the cost of tolerating outages. Finally, we propose a novel technique to verify the consistency guarantees offered by real-world storage systems. This technique leverages our declarative approach to consistency: we consider consistency semantics as invariants over graph representations of storage systems executions. A preliminary implementation proves this approach practical and useful in improving over the state-of-the-art on consistency verification.

Résumé

La conception des systèmes distribués est une tâche onéreuse : les objectifs de performance, d'exactitude et de fiabilité sont étroitement liés et ont donné naissance à des compromis complexes décrits par de nombreux résultats théoriques. Ces compromis sont devenus de plus en plus importants à mesure que le calcul et le stockage se sont déplacés vers des architectures distribuées. De plus, l'absence d'approches systématiques de ces problèmes dans les outils de programmation modernes les a aggravé — d'autant que de nos jours la plupart des programmeurs doivent relever les défis liés aux applications distribués. En conséquence, il existe un écart évident entre les abstractions de programmation, les exigences d'application et la sémantique de stockage, ce qui entrave le travail des concepteurs et des développeurs.

Cette thèse présente un ensemble de contributions tourné vers la conception de systèmes de stockage distribués fiables, en examinant ces questions à travers le prisme de la cohérence. Nous commençons par fournir un cadre uniforme et déclarative pour définir formellement les modèles de cohérence. Nous utilisons ce cadre pour décrire et comparer plus de cinquante modèles de cohérence non transactionnelles proposés dans la littérature. La nature déclarative et composite de ce cadre nous permet de construire un classement partiel des modèles de cohérence en fonction de leur force sémantique. Nous montrons les avantages pratiques de la composabilité en concevant et en implémentant Hybris, un système de stockage qui utilise différents modèles pour améliorer la cohérence faible généralement offerte par les services de stockage dans les nuages. Nous démontrons l'efficacité d'Hybris et montrons qu'il peut tolérer les erreurs arbitraires des services du nuage au prix des pannes. Enfin, nous proposons une nouvelle technique pour vérifier les garanties de cohérence offertes par les systèmes de stockage du monde réel. Cette technique s'appuie sur notre approche déclarative de la cohérence : nous considérons les modèles de cohérence comme invariants sur les représentations graphiques des exécutions des systèmes de stockage. Une mise en œuvre préliminaire prouve cette approche pratique et utile pour améliorer l'état de l'art sur la vérification de la cohérence.

Acknowledgements

It has been said, not without reason, that pursuing a PhD degree is a solitary endeavor. Although I am in no position to deny this belief, I will not fail to acknowledge the invaluable support of colleagues, experienced researchers, family and friends.

First, I thank my PhD advisor, Marko Vukolić, for believing in me and giving me the chance to embark on this journey; for keeping me on a path of profitable inquiry — despite my tendency to explore, to wander, and to get wonderfully lost; and for teaching me the inherent value of solid and rigorous research work.

I thank all members of the thesis committee for taking the time to read this thesis and providing their constructive and helpful feedback. A special thank goes also to Prof. Pietro Michiardi and Prof. Maurizio Filippone for serving as inspiring examples of passionate and hustling researchers in the young yet promising Data Science department of Eurecom.

I would also like to thank the colleagues at Eurecom with whom I shared part of the journey that brought me here: Alberto Benegiamo, Shengyun Liu, Yongchao Tian, Xiaohu Wu, Jingjing Zhang, Romain Favraud, Laurent Gallo, Mariano Graziano, Andrea Enrici, Martina Cardone, Xiwen Jiang, Irfan Khan, Robin Thomas, Luigi Vigneri, Fabio Pulvirenti, Daniele Venzano, Kurt Cutajar, Francesco Pace, Marco Milanesio, Paul Pidou, Hung Phan and Trung Nguyen. Thank you all for your cheerful and sympathetic company!

Heartfelt thanks also go to the friends who made my stay in Côte D'Azur enjoyable and fun, including: Michelle Chong, Luca Schiatti, Carla Armata, Giulia Brotto, Valeria Neglia, Cecilia Morandin, Adriano Di Martino and Rim Tebourbi.

Last but not least, I acknowledge the blind luck of having been born and raised in a wealthy part of the world, and to have enjoyed the supreme privilege of education. For all this and for their continuous love and support, I am grateful to my parents.

Non vorrei partire con vantaggio: temo che mi sentirei un profittatore, e dovrei chinare la fronte per tutta la vita davanti a ciascuno dei miei compagni non privilegiati. Accetto, ma vorrei nascere a caso, come ognuno: fra i miliardi di nascituri senza destino, fra i predestinati alla servitù o alla contesa fin dalla culla, se pure avranno una culla. Preferisco nascere negro, indiano, povero, senza indulgenze e senza condoni. Lei mi capisce, non è vero? Lei stesso lo ha detto, che ogni uomo è artefice di se stesso: ebbene, è meglio esserlo appieno, costruirsi dalle radici. Preferisco essere solo a fabbricare me stesso, e la collera che mi sarà necessaria, se ne sarò capace; se no, accetterò il destino di tutti.
Il cammino dell'umanità inerme e cieca sarà il mio cammino.

P. Levi, *Vizio di forma*

One equal temper of heroic hearts,
made weak by time and fate, but strong in will
to strive, to seek, to find, and not to yield.

A. Tennyson, *Ulysses*

Contents

Chapter 1

Introduction

Over the past decade, the rise of cloud computing, as well as a steady decrease of the cost of storage and computation [12, 152], have enabled the widespread adoption of Internet-scale and mobile applications. As users of these pervasive services generate increasingly large amounts of data (so-called "big data"), enterprises and governments strive to collect and analyze them to extract profitable information [147]. In turn, the growing popularity of *data-intensive* applications, together with the demise of Moore's law [224] have prompted designers to resort to scale-out, *distributed* architectures [223, 31] to boost performance and tolerate failures. The data storage layer has a prominent role in the architecture of such systems, as it is commonly charged with the crucial task of addressing the challenges of concurrency and distribution [97, 197]. Distributed systems are indeed notoriously difficult to design and program because of two inherent sources of uncertainty that plague their executions. First, due to communication asynchrony, ordering and timing of messages delivery is subject to nondeterminism. Second, failures of system components and communication attempts may lead to incomplete outcomes or even corrupt application state.

To cope with these challenges, researchers have conceived a set of theoretical results [111, 120] that shaped the design space by exposing a multifaceted tradeoff of correctness, reliability and performance. Following and informing these studies, practitioners have developed and commercialized different database systems. The design of the first generation of database systems reflected an era in which computation was mostly centralized and vertical scaling was still a viable option. In this setting, databases commonly offered strong, transactional semantics, such as *serializability* [50]. In a nutshell, serializability guarantees that the concurrent execution of bundles of operations (i.e. *transactions*), will be equivalent to some serial execution. However, in the last decade, a new generation of database systems — often called "NoSQL" systems — made its appearance. The common trait of this new, loosely defined family of systems is their weak semantic guarantees about database state and query results. In effect, the NoSQL stores forego transactional, strong semantics to provide greater scalability and performance

through distribution [96, 237]. In practice, the burden of checking and enforcing correctness constraints is left, often ambiguously, to the programmer — frequently resulting in error-prone distributed applications [33]. Hence, a new breed of research works and products have strived to bridge the divide between correctness and scalability, by offering optimized implementations of strong semantic abstractions [192, 257] or by exposing tradeoffs matching specific application semantics [34, 125].

In this thesis, we deal with the design and engineering conundrum of building scalable and correct distributed storage systems by framing it through the prism of *consistency*. Consistency is a correctness semantic notion that lies at the heart of the intricate set of constraints and requirements involved in the design of reliable distributed systems. Its outcomes directly affect both user experience and the engineering aspects of a system, spanning across different stack layers and domains. For this reason, consistency constitutes an ideal tool to represent and analyze distributed systems. However, we currently lack a comprehensive theoretical foundation to define and compare consistency semantics. In turn, the lack of a rigorous consistency terminology and grammar prevents designers from uncovering and experimenting further possible tradeoffs of correctness and performance. Furthermore, a practical framework of elemental semantics would allow us to reason about the potential benefit of their compositions, and would enable more accurate techniques to test the correctness of existing systems.

This dissertation presents an attempt to fill this void in the research literature. Specifically, the contributions presented in this thesis can be summarized as follows:

— We devise a semantic model to describe consistency notions. Using such model we provide formal definitions for the most popular semantics defined so far by researchers and practitioners for non-transactional stores.

— We design and evaluate a key-value storage system that improves over the robustness and consistency of public cloud storage, yet retains scalability and operational benefits. Our design exploits the favorable composability of two different consistency semantics, and make efficient use of public and private storage resources.

— We describe and evaluate a novel approach to consistency verification. Our approach focuses on automatically verifying the validity of consistency semantics as declarative invariants of storage systems executions.

In the remainder of this chapter, we expound on each of our key contributions and lay out the dissertation outline.

1.1 A semantic framework to express consistency

In spite of its relevance in the design of concurrent and distributed systems, the concept of consistency has historically lacked a common frame of reference to describe its aspects across communities of researchers and practitioners. In the past, some joint efforts between research and industry have helped formalize, compare and even standardize *transactional* semantics [21, 126, 8]. However, these works fall short of including the advances of the last decade of research on databases, and they do not consider *non-transactional* semantics. Recently, non-transactional consistency has enjoyed a resurgence due to the increasing popularity of NoSQL stores. Hence, new models have been devised to account for various combinations of fault tolerance concerns and application invariants. Researchers have been striving to formulate the minimum requirements in terms of correctness and, therefore, coordination, to allow for the design of fast yet functional distributed systems [34, 30]. Furthermore, an ongoing and exciting research trend has been tackling this issue leveraging different tools and stack layers, spanning from programming languages [16] to data structures [213] and application-level static checkers [219, 125]. However, despite active recent research on different aspects of storage correctness, we currently lack a sound and comprehensive model to reason about consistency.

As a first contribution of this thesis, we propose a framework to define and compare non-transactional consistency semantics. This framework aims at capturing all the salient aspects of consistency within a minimum set of declarative, composable properties. We use this framework to provide a formal taxonomy of more than fifty consistency semantics introduced in the previous three decades of research. Thanks to these new formal definitions, we are able to compare and position them in a partially ordered hierarchy according to their semantic "strength." In addition, we map those semantics to corresponding implementations of prototypes and full-fledged systems described in research literature.

We believe this contribution will serve as a first normalization step and bring about further clarity in future debates over design of distributed systems. Moreover, we hope that this work will help researchers engage in further explorations of compositions of elemental consistency semantics. The second contribution of this thesis, which we summarize in the next section, represents — among other things — a first step in this research direction.

1.2 Robust hybrid cloud storage

The recent advent of highly available and scalable storage systems under the commercial label of "cloud storage" has radically changed the way companies and end users deal with data storage. Cloud storage offers unprecedented on-demand scalability, enhanced durability leveraging planet-wide data replication, and the convenient offloading of operational tasks. However, cloud storage also presents new challenges concerning data privacy, integrity and availability [128]. Moreover, the lack of interoperability standards among different cloud providers significantly increases switching costs [4]. Finally, cloud stores are notorious for offering weak consistency guarantees [237].

To overcome these issues, researchers have conceived compound systems that use multiple cloud stores to distribute trust and increase reliability — in the approach often referred to as *multi-cloud*. At the same time, cloud providers have started offering *hybrid* cloud services, leveraging on-premises resources to store confidential data. Unfortunately, both the hybrid and the multi-cloud approaches present substantial drawbacks. The multi-cloud approach suffers from severe performance limitations inherent to its cross-cloud communication protocols. Besides, the performance and monetary costs of tolerating arbitrary cloud failures are prohibitively high, and the consistency guarantees only proportional to those of the underlying clouds [53]. On the other hand, commercial hybrid cloud storage still presents security and reliability concerns related to the use of a single cloud.

As the second contribution of this thesis, we present the design and evaluation of a hybrid multi-cloud key-value store we called Hybris. Hybris combines the benefits of both hybrid and multi-cloud storage through a composition of public clouds and private resources. It tolerates arbitrary (e.g., malicious) cloud faults at the affordable price of tolerating outages. Interestingly, Hybris' design is a practical instance of composition of consistency semantics. Its resulting consistency is the strongest of those offered by the individual subsystems. Namely, Hybris provides strong consistency even when public clouds are weakly consistent.

Hybris is efficient and scalable as it stores lightweight metadata on private premises, while replicating or erasure-coding bulk data on a few cloud stores. We implemented the Hybris protocol and benchmarked it in different contexts including geo-replicated deployments and as backend of a storage synchronization application. Our evaluation shows that Hybris outperforms comparable state-of-the-art robust cloud storage systems and approaches the performance of bare-bone commodity cloud stores.

1.3 Automated declarative consistency verification

Although with Hybris we experimented the design of a storage system, most of real-world engineering entails dealing with off-the-shelf commercial systems. Therefore, a crucial part of software engineering consists in understanding what are the assumptions and guarantees that each component of a given architecture brings about, and how they compose. Unfortunately, those assumptions and guarantees are often poorly expressed — if at all — by the original developers. Moreover, the terminology adopted in their specifications often does not use a standard set of unambiguous concepts, further hindering the integration task. This is especially true when it comes to storage systems and their consistency guarantees. Oftentimes, consistency specifications are in fact expressed in an informal, hence imprecise way; they refer to specific settings and their validity hinges on ad-hoc assumptions. As a result, they are incompatible and incomparable, thus leading to convoluted and error-prone testing techniques.

Researchers have responded to this issue by suggesting different approaches. Some works in the literature proposed and perfected algorithms to establish whether an execution respects strong consistency semantics [191, 122]. Another popular approach consists in quantifying eventual consistency by means of client-side staleness measurements [47, 169]. Finally, transactional storage semantics have been traditionally verified by adopting a graph-based approach [8, 254]. While all these techniques are valid and effective, they fail to encompass the entirety of the consistency spectrum and the composition of the elemental consistency semantics.

As third contribution of this thesis, we propose an automated declarative approach to consistency verification. In order to achieve this, we advocate the adoption of the declarative semantic model to express consistency that we introduce in Chapter 2. Such an axiomatic model allows to consider consistency as a mere invariant of storage system executions. In turn, considering consistency as an invariant allows applying advanced verification techniques, such as *property-based* testing [83]. Hence, we developed Cover, a prototype of a verification framework that implements property-based consistency verification. Conver lets the user select a set of consistency predicates that have to be respected during executions. Then, it automatically generates random executions consisting of concurrent read and write operations, possibly tailored to better verify specific consistency semantics. During each execution, Conver verifies the validity of the chosen consistency predicates and, in case of inconsistency, it outputs a visualization highlighting the sequence of operations that led to the anomaly. Moreover, Conver can inject faults in the storage system by terminating processes and creating network partitions. In summary, Conver relieves programmers from the burden of generating and running accurate tests to verify the correctness of storage systems.

1.4 Outline and previously published work

The remainder of this thesis proceeds as follows. Chapter 2 introduces a declarative semantic framework to define consistency semantics and surveys various consistency models. Two tables summarizing predicates expressing consistency semantics and their implementations are presented in Appendices C and A. In Appendix B, we prove the strength relations between consistency semantics, following the formal definitions we provide in Chapter 2. Chapter 3 discusses the challenges concerning cloud storage and presents Hybris. Hybris' correctness proofs and listings of algorithms are postponed to Appendix D. Chapter 4 considers the hurdles of verifying databases correctness conditions and introduces a property-based approach. The dissertation then concludes in Chapter 5 with a discussion of topics for future work.

This dissertation includes contributions from previously published research work done jointly with several collaborators. In particular, Chapter 2 includes material from:

> P. Viotti and M. Vukolić, "Consistency in non-transactional distributed storage systems," *ACM Computing Surveys*, vol. 49, no. 1, 2016.

Chapter 3 presents work published in:

> P. Viotti, D. Dobre, and M. Vukolić, "Hybris: robust hybrid cloud storage," *ACM Transactions on Storage*, vol. 13, issue 3, no. 9, 2017.

which, in turn, revises and extends:

> D. Dobre, P. Viotti, and M. Vukolić, "Hybris: robust hybrid cloud storage," *ACM Symposium on Cloud Computing*, 2014.

Chapter 4 revises and extends:

> P. Viotti, C. Meiklejohn and M. Vukolić, "Towards property-based consistency verification," *ACM Workshop on the Principles and Practice of Consistency for Distributed Data*, 2016.

Outside the main scope of this thesis, during my PhD, I also worked on the implementation and testing of a state machine replication protocol that guarantees *cross fault tolerance*, which resulted in the following publication:

> S. Liu, P. Viotti, C. Cachin, V. Quéma and M. Vukolić,
> "XFT: Practical Fault Tolerance beyond Crashes,"
> *USENIX Symposium on Operating Systems Design and Implementation*, 2016.

Chapter 2

Consistency in Non-Transactional Distributed Storage Systems

In this chapter, we develop a formal framework to express non-transactional consistency semantics. We use this framework to define consistency semantics described in the past decades of research. These new, formal definitions, enable a structured and comprehensive view of the consistency spectrum, which we illustrate by contrasting the "strength" and features of individual semantics.

2.1 Introduction

Faced with the inherent challenges of failures, communication asynchrony and concurrent access to shared resources, distributed system designers have continuously sought to hide these fundamental concerns from users by offering abstractions and semantic models of various strength. The ultimate goal of a distributed system is seemingly simple, as, ideally, it should just be a fault-tolerant and more scalable version of a centralized system. The ideal distributed system should leverage distribution and replication to boost availability by masking failures, provide scalability and/or reduce latency, yet maintain the simplicity of use of a centralized system — and, notably, its *consistency* — providing the illusion of sequential access. Such *strong* consistency criteria can be found in early works that paved the way of modern storage systems [163], as well as in the subsequent advances in defining general, practical correctness conditions, such as *linearizability* [138]. Unfortunately, the goals of high availability and strong consistency, in particular linearizability, have been identified as mutually conflicting in many practical circumstances. Negative theoretical results and lower bounds, such as the FLP impossibility proof [111] and the CAP theorem [120], shaped the design space of distributed systems. As a result, distributed system designers have either to give up the idealized goals of scalability and availability, or relax consistency.

In recent years, the rise of commercial Internet-scale computing has motivated system designers to prefer availability over consistency, leading to the advent of weak and eventual consistency [228, 210, 237]. Consequently, much research has been focusing on attaining a better understanding of those weaker semantics [33], but also on adapting [38, 257, 246] or dismissing and replacing stronger ones [135]. Along this line of research, tools have been conceived to handle consistency at the level of programming languages [16], data objects [214, 66] or data flows [19].

Today, however, despite roughly four decades of research on various flavors of consistency, we lack a structured and comprehensive overview of different consistency notions that appeared in distributed storage research. In this chapter, we aim to help fill this void by providing an overview of over 50 different consistency notions. Our survey ranges from linearizability to eventual and weak consistency, defining precisely many of these, in particular where the previous definitions were ambiguous. We further provide a partial order among different consistency notions, ordering them by their semantic "strength", which we believe will reveal useful in further research. Finally, we map the consistency semantics to different practical systems and research prototypes. The scope of this chapter is restricted to consistency models that apply to any replicated object having a sequential specification and exposing non-transactional operations. We focus on non-transactional storage systems as they have become increasingly popular in recent years due to their simple implementations and good scalability. As such, this chapter complements the existing survey works done in the context of transactional consistency semantics [8].

This chapter is organized as follows. In Section 2.2 we define the model we use to represent a distributed system and set up the framework for reasoning about different consistency semantics. To ensure the broadest coverage and make our work faithfully reflect the features of modern storage systems, we model distributed systems as asynchronous, i.e. without predefined constraints on timing of computation and communication. Our framework, which we derive from the work by Burckhardt [65], captures the dynamic aspects of a distributed system, through *histories* and *abstract executions* of such systems. We define an execution as a set of actions (i.e. *operations*) invoked by some processes on the storage objects through their interface. To analyze executions we adopt the notion of history, i.e. the set of operations of a given execution. Leveraging the information attached to histories, we are able to properly capture the intrinsic complexity of executions. Namely, we can group and relate operations according to their features (e.g., by the processes and objects they refer to, and by their timings), or by the dynamic relationships established during executions (e.g., causality). Additionally, abstract executions augment histories with orderings of operations that account for the resolution of update conflicts and their propagation within the storage system.

Section 2.3 brings the main contribution of this chapter: a survey of more than 50 different consistency semantics previously proposed in the context of non-transactional distributed

storage systems.[1] We define many of these models employing the framework specified in Section 2.2, i.e. using declarative compositions of logic predicates over graph entities. In turn, these definitions enable us to establish a partial order of consistency semantics according to their semantic strengths — which we illustrate in Figure 2.1. For sake of readability, we also loosely classify consistency semantics into *families*, which group them by their common traits.

We discuss our work in the context of related surveys on consistency in Section 2.4. We further complement our survey with a summary of all consistency predicates defined in this work, which we postpone to Appendix A. In Appendix B, we provide proofs of the relative strengths of the consistency semantics formally specified in this chapter. In addition, for all consistency models mentioned in this work, we provide references to their original, primary definitions, as well as pointers to research papers that describe related implementations (Appendix C). Specifically, we reference implementations that appeared in recent proceedings of the most relevant venues. We believe that this is a useful contribution on its own, as it will allow scholars to navigate more easily through the extensive body of literature that deals with the subtleties of consistency.

[1]Note that, while we focus on consistency semantics proposed in the context of distributed storage, our approach maintains generality as our consistency definitions are applicable to other replicated data structures beyond distributed storage.

2.2 System model

In this section, we specify the main concepts behind the reasoning about consistency semantics carried out in the rest of this chapter. We rely on the concurrent object abstraction, as presented by Lynch and Tuttle [183] and by Herlihy and Wing [138], for the definitions of fundamental "static" elements of the system, such as objects and processes. Moreover, to describe the dynamic behavior of the system (i.e. executions), we build upon the axiomatic mathematical framework laid out by Burckhardt [65]. We decided to rely on an axiomatic framework since operational specifications of consistency models — especially of *weak* consistency models — can become unwieldy, overly complicated and hard to reason about. In comparison, axiomatic specifications are more expressive and concise, and are amenable to static checking — as we will see in Chapther 4.

2.2.1 Preliminaries

Objects and Processes We consider a distributed system consisting of a finite set of *processes*, modeled as I/O automata [183], interacting through *shared* (or *concurrent*) *objects* via a fully-connected, asynchronous communication network. Unless stated otherwise, processes and shared objects (or, simply, objects) are *correct*, i.e. they do not fail. Processes and objects have unique identifiers. We define $ProcessIds$ as the set of all process identifiers and $ObjectIds$ as the set of all object identifiers.

Additionally, each object has a unique *object type*. Depending on its type, the object can assume *values* belonging to a defined domain denoted by $Values$, [2] and it supports a set of primitive non-transactional *operation types* (i.e. $OpTypes = \{rd, wr, inc, \ldots\}$) that provide the only means to manipulate the object. For simplicity, and without loss of generality, unless specified otherwise, we further classify operations as either *reads* (rd) or *writes* (wr). Namely, we model as a write (or *update*) any operation that modifies the value of the object. Conversely, a read returns to the caller the current value held by the object's replica without causing any change to it. We adopt the term *object replicas*, or simply *replicas*, to refer to the different copies of a same named shared object maintained in the storage system for fault tolerance or performance enhancement. Ideally, replicas of the same shared object should hold the same data at any time. The coordination protocols among replicas are however determined by the implementation of the shared object.

Time Unless specified otherwise, we assume an asynchronous computation and communication model, namely, with no bounds on computation and communication latencies. However, when describing certain consistency semantics, we will be using terms such as *recency* or *staleness*. This terminology relates to the concept of *real time*, i.e. an ideal and global notion of

[2] For readability, we adopt a notation in which $Values$ is implicitly parametrized by the object type.

time that we use to reason about histories a posteriori. However, this notion is not accessible by processes during executions. We refer to the real time domain as $Time$, which we model as the set of positive real numbers, i.e. $\mathbb{R}^+$.

2.2.2 Operations, histories and abstract executions

Operations We describe an operation issued by a process on a shared object as the tuple $(proc, type, obj, ival, oval, stime, rtime)$, where:

- $proc \in ProcessIds$ is the id of the process invoking the operation.
- $type \in OpTypes$ is the operation type.
- $obj \in ObjectIds$ is the id of the object on which the operation is invoked.
- $ival \in Values$ is the operation input value.
- $oval \in Values \cup \{\nabla\}$ is the operation output value, or ∇ if the operation never returns.
- $stime \in Time$ is the operation invocation real time.
- $rtime \in Time \cup \{\Omega\}$ is the operation return real time, or Ω if the operation never returns.

By convention, we use the special value $\sqcup \in Values$ to represent the input value (i.e. $ival$) of reads and the return value (i.e. $oval$) of writes. For simplicity, given operation op, we will use the notation $op.par$ to access its parameter named par as expressed in the corresponding tuple (e.g., $op.type$ represents its type, and $op.ival$ its input value).

Histories A *history* H is the set of all operations invoked in a given execution. We further denote by $H|_{wr}$ (resp., $H|_{rd}$) the set of write (resp., read) operations of a given history H (e.g., $H|_{wr} = \{op \in H : op.type = wr\}$).

We further define the following relations on elements of a history: [3]

- rb (*returns-before*) is a natural partial order on H based on real-time precedency. Formally: $rb \triangleq \{(a, b) : a, b \in H \wedge a.rtime < b.stime\}$.
- ss (*same-session*) is an equivalence relation on H that groups pairs of operations invoked by the same process — we say such operations belong to the same *session*. Formally: $ss \triangleq \{(a, b) : a, b \in H \wedge a.proc = b.proc\}$.
- so (*session order*) is a partial order defined as: $so \triangleq rb \cap ss$. Since we assume that there can be at most one operation pending per session at any given time, so totally orders each session.
- ob (*same-object*) is an equivalence relation on H that groups pairs of operations invoked on the same object. Formally: $ob \triangleq \{(a, b) : a, b \in H \wedge a.obj = b.obj\}$.

[3] For better readability, we implicitly assume relations are parametrized by a history.

— $concur$ as the symmetric binary relation designating all pairs of real-time *concurrent* operations invoked on the same object. Formally: $concur \triangleq ob \setminus rb$.

For $(a, b) \in rel$ we may alternatively write $a \xrightarrow{rel} b$. We further denote by rel^{-1} the inverse relation of rel. Moreover, for the sake of a more compact notation, we use binary relation projections. For instance, $rel|_{wr \to rd}$ identifies all pairs of operations belonging to rel consisting of a write and a read operation. Furthermore, if rel is an equivalence relation, we adopt the notation $a \approx_{rel} b \triangleq [a \xrightarrow{rel} b]$. We recall that an equivalence relation rel on set H partitions H into equivalence classes $[a]_{rel} = \{b \in H : b \approx_{rel} a\}$. We write $H/\approx_{rel}$ to denote the set of equivalence classes determined by rel.

We complement the $concur$ relation with the function $Concur : H \to 2^H$ that denotes the set of write operations concurrent with a given operation:

$$Concur(a) \triangleq \{b \in H|_{wr} : (a, b) \in concur\} \tag{2.1}$$

Abstract executions We model system executions using the concept of *abstract execution*, following Burckhardt [65]. An abstract execution is a multi-graph $A = (H, vis, ar)$ built on a given history H, which it complements with two relations on elements of H, i.e. vis and ar. Whereas histories describe the observable outcomes of executions, vis and ar, intuitively, capture the nondeterminism of the asynchronous environment (e.g., message delivery order), as well as implementation-specific constraints (e.g., conflict-resolution policies). In other words, vis and ar determine the relations between pairs of operations in a history that explain and justify its outcomes. More specifically:

— vis (*visibility*) is an acyclic natural relation that accounts for the propagation of write operations. Intuitively, a be visible to b (i.e. $a \xrightarrow{vis} b$) means that the effects of a are visible to the process invoking b (e.g., b may read a value written by a). Two write operations are *invisible* to each other if they are not ordered by vis.

— ar (*arbitration*) is a *total* order on operations of a history that specifies how the system resolves conflicts due to operations that are concurrent or invisible to each other. In practice, such a total order can be achieved in various ways: through the adoption of a distributed timestamping [160] or consensus protocol [56, 129, 159], using a centralized serializer, or a deterministic conflict resolution policy.

Depending on constraints expressed by vis, during an execution processes may observe different partial orderings of write operations, which we call *serializations*. We further define the *happens-before* order (hb) as the transitive closure of the union of so and vis, denoted by:

$$hb \triangleq (so \cup vis)^+ \tag{2.2}$$

2.2.3 Replicated data types and return value consistency

Rather than defining the system state at a given time as a set of values held by shared objects, following Burckhardt [65], we employ a graph abstraction called (operation) *context*. The operation context encodes the information of an abstract execution A taking a projection on visibility (vis) with respect to a given operation op. Formally, we define as $\mathcal{C}$ the set of contexts of all operations in a given abstract execution A, and the context of a specific operation op as:

$$C = cxt(A, op) \triangleq A|_{op, vis^{-1}(op), vis, ar} \tag{2.3}$$

Furthermore, we adopt the concept of *replicated data type* [65] to define the type of shared object implemented in the distributed system (e.g., read/write register, counter, set, queue, etc.). For each replicated data type, a function $\mathcal{F}$ specifies the set of expected return values of an operation $op \in H$ in relation to its context, i.e. $\mathcal{F}(op, cxt(A, op))$. Using $\mathcal{F}$, we can define *return value consistency* as:

$$\text{RVal}(\mathcal{F}) \triangleq \forall op \in H : op.oval \in \mathcal{F}(op, cxt(A, op)) \tag{2.4}$$

Essentially, return value consistency is a predicate on abstract executions that prescribes that the return value of any given operation of that execution belongs to the set of its intended return values.

Given operation $b \in H$ and its context $cxt(A, b)$, we let $a = prec(b)$ be the (unique) latest operation preceding b in ar, such that: $a.oval \neq \nabla \wedge a \in H|_{wr} \cap vis^{-1}(b)$. In other words, $prec(b)$ is the last write visible to b according to the ordering specified by ar. If no such preceding operation exists (e.g., if b is the first operation of the execution according to ar), by convention $prec(b).ival$ is a default value $\perp$.

In this paper we adopt the replicated read/write register (i.e. read/write storage) as our reference data type, which is defined by the following return value function:

$$\mathcal{F}_{reg}(op, cxt(A, op)) = prec(op).ival \tag{2.5}$$

Note that, while the focus of this survey is on read/write storage, the consistency predicates defined in this chapter take $\mathcal{F}$ as a parameter, and therefore directly extend to other replicated data types.

2.2.4 Consistency semantics

Following Burckhardt [65], we define *consistency semantics*, sometimes also called *consistency guarantees*, as conditions on attributes and relations of abstract executions, expressed as first-order logic predicates. We write $A \models \mathcal{P}$ if the consistency predicate $\mathcal{P}$ is true for abstract execution A. Hence, defining a consistency model amounts to collecting all the required

consistency predicates and then specifying that histories must be justifiable by at least one abstract execution that satisfies them all.

Formally, given history H and $\mathcal{A}$ as the set of all possible abstract executions on H, we say that history H satisfies some consistency predicates $\mathcal{P}_1, \ldots \mathcal{P}_n$ if it can be extended to some abstract execution that satisfies them all:

$$H \models \mathcal{P}_1 \wedge \cdots \wedge \mathcal{P}_n \Leftrightarrow \exists A \in \mathcal{A} : \mathcal{H}(A) = H \wedge A \models \mathcal{P}_1 \wedge \cdots \wedge \mathcal{P}_n \tag{2.6}$$

In the notation above, given the abstract execution $A = (H, vis, ar)$, $\mathcal{H}(A)$ denotes H.

2.3 Non-transactional consistency semantics

In this section we analyze and survey the consistency semantics of systems which adopt single operations as their primary operational constituent (i.e. non-transactional consistency semantics). The consistency semantics described in the rest of the chapter are summarized in Figure 2.1, which presents a partial ordering of consistency semantics according to their semantic strength, as well as a more loosely defined clustering into *families* of consistency models. This classification draws both from the relative strength of different consistency semantics, and from the underlying common factors that underpin their definitions. In Appendix B, we provide proofs for some of the strength relations between consistency semantics that we formally specify in this section. The remaining relations showed in Fig. 2.1 reflect intuitive notions, claims or formal proofs reported in peer-reviewed research literature.

In the remainder of this section, we examine each family of consistency semantics. Section 2.3.1 introduces linearizability and other strong consistency models, while in Section 2.3.2 we consider eventual and weak consistency. Next we analyze PRAM and sequential consistency (Section 2.3.3), and, in Section 2.3.4, the models based on the concept of session. Section 2.3.5 presents an overview of consistency semantics explicitly dealing with causality, while in Section 2.3.6 we study staleness-based models. This is followed by an overview of fork-based models (Section 2.3.7). Section 2.3.8 and 2.3.9 respectively deal with tunable and per-object semantics. Finally, we survey the family of consistency models based on synchronization primitives (Section 2.3.10).

2.3.1 Linearizability and related "strong" consistency semantics

The gold standard and the central consistency model for non-transactional systems is **linearizability**, defined by Herlihy and Wing [138]. Roughly speaking, linearizability is a correctness condition that establishes that each operation shall appear to be applied instantaneously at a certain point in time between its invocation and its response. Linearizability, often informally dubbed *strong consistency*,[4] has long been regarded as the ideal correctness condition. Linearizability features the *locality* property: a composition of linearizable objects is itself linearizable — hence, linearizability enables modular design and verification.

Although very intuitive to understand, the strong semantics of linearizability makes it challenging to implement. In this regard, Gilbert and Lynch [120], formally proved the *CAP* theorem, an assertion informally presented in previous works [146, 95, 86, 61], that binds linearizability to the ability of a system of maintaining a non-trivial level of availability when confronted with network partitions. In a nutshell, the CAP theorem states that, in the presence

[4]Note that the adjective "strong" has also been used in literature to identify indistinctly *linearizability* and *sequential consistency* (which we define in Section 2.3.3), as they both entail single-copy-semantics and require that a single ordering of operations be observed by all processes.

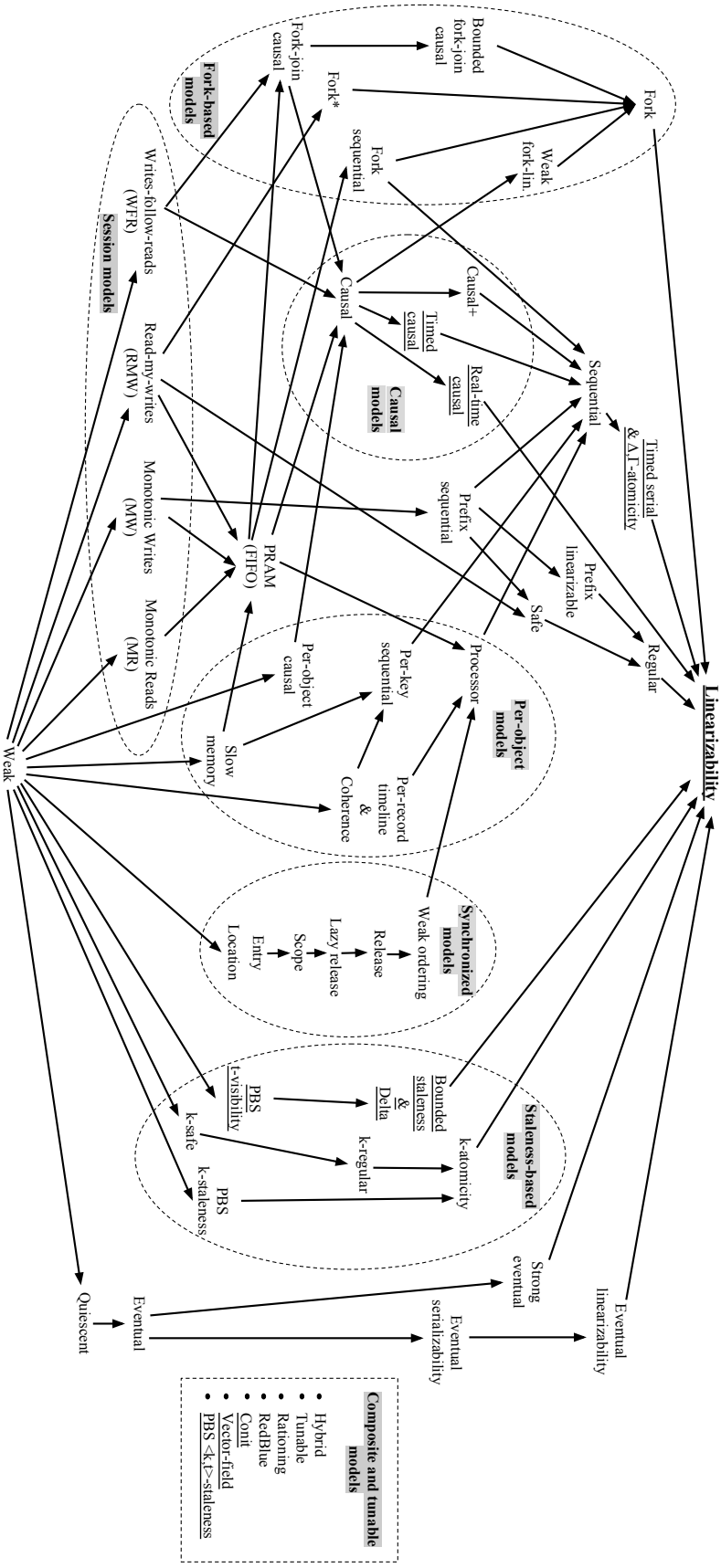


Figure 2.1 – Hierarchy of non-transactional consistency models.
A directed edge from consistency semantics A to consistency semantics B means that any execution that satisfies B also satisfies A. Underlined models explicitly deal with timing guarantees.

of network partitions, availability or linearizability are mutually incompatible: the distributed storage system must sacrifice one or the other (or both).

Burckhardt [65] breaks down linearizability into three components:

$$\textsc{Linearizability}(\mathcal{F}) \triangleq \textsc{SingleOrder} \wedge \textsc{RealTime} \wedge \textsc{RVal}(\mathcal{F}) \tag{2.7}$$

where:

$$\textsc{SingleOrder} \triangleq \exists H' \subseteq \{op \in H : op.oval = \nabla\} : vis = ar \setminus (H' \times H) \tag{2.8}$$

and

$$\textsc{RealTime} \triangleq rb \subseteq ar \tag{2.9}$$

In other words, SingleOrder imposes a single global order that defines both vis and ar, whereas RealTime constrains arbitration (ar) to comply to the returns-before partial ordering (rb). Finally, RVal($\mathcal{F}$) specifies the return value consistency of a replicated data type. We recall that, as per Eq. 2.5, in case of read/write storage this is the value written by the last write (according to ar) visible to a given read operation rd.

A definition closely related to linearizability is Lamport's **atomic** register semantics [164]. Lamport describes a single-writer, multi-reader (SWMR) shared register to be atomic *iff* each read operation not overlapping in time with a write returns the last value written on the register, and each operation appears to be applied on the shared register at a point in time (a *linearization point*) between its invocation and its response. [5] It is easy to show that atomicity and linearizability are equivalent for read-write registers. However, linearizability is a more general condition designed for generic shared data structures that allow for a broader set of operational semantics than those offered by registers.

Lamport [164] also defines two slightly weaker semantics for SWMR registers: **safe** and **regular**. In the absence of read-write concurrency, they both guarantee that a read returns the last written value, exactly like the atomic semantics. However, the set of return values differs for a read operation concurrent with a write. For a safe register, a read concurrent with some write may return any value. On the other hand, for a regular register, a read operation concurrent with some writes may return either the value written by the most recent complete write, or a value written by a concurrent write. This difference is illustrated in Figure 2.2.

Formally, regular and safe semantics can be defined as follows:

$$\textsc{Regular}(\mathcal{F}) \triangleq \textsc{SingleOrder} \wedge \textsc{RealTimeWrites} \wedge \textsc{RVal}(\mathcal{F}) \tag{2.10}$$

[5] The existence of an instant at which each operation becomes atomically *visible* was originally postulated in [162].

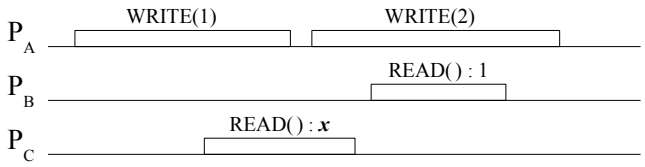


Figure 2.2 – An execution exhibiting read-write concurrency (real time flows from left to right). The register is initialized to 0. Atomic (linearizable) semantics prescribes x to be 0 or 1. Regular semantics allows x to be 0, 1 or 2. With safe semantics x may be any value.

$$\textsc{Safe}(\mathcal{F}) \triangleq \textsc{SingleOrder} \wedge \textsc{RealTimeWrites} \wedge \textsc{SeqRVal}(\mathcal{F}) \tag{2.11}$$

where

$$\textsc{RealTimeWrites} \triangleq rb|_{wr \to op} \subseteq ar \tag{2.12}$$

is a restriction of real-time ordering only for writes that precede a read or another write, and

$$\textsc{SeqRVal}(\mathcal{F}) \triangleq \forall op \in H : Concur(op) = \emptyset \Rightarrow op.oval \in \mathcal{F}(op, cxt(A, op)) \tag{2.13}$$

restricts the return value consistency only to read operations that are not concurrent with any write.

2.3.2 Weak and eventual consistency

At the opposite end of the consistency spectrum lies **weak** consistency. Although this term has been frequently used in literature to refer to any consistency model weaker than sequential consistency, recent works [237, 46] associate it to a more specific albeit rather vague definition: a weakly consistent system does not guarantee that reads return the most recent value written, and some (often underspecified) requirements have to be satisfied for a value to be returned. In effect, weak consistency does not provide ordering guarantees — hence, no synchronization protocol is actually required. This model is implemented in contexts in which a synchronization protocol would be too costly, and a fortuitous exchange of information between replicas is good enough. Typical examples of weak consistency are the relaxed caching policies that are applied across various tiers of a web application, such as the cache implemented in web browsers.

Eventual consistency is a slightly stronger notion than weak consistency. Namely, under eventual consistency, replicas converge towards identical copies after an undefined period of quiescence. In other words, if no new write operations are invoked on the object, *eventually* all reads will return the same value. Eventual consistency was first defined by Terry et al. [228] and then further popularized more than a decade later by Vogels [237] with the advent of highly available storage systems (i.e. *AP* systems in the CAP theorem parlance). Eventual consistency is especially suited in contexts where coordination is not practical or too expensive,

e.g., in mobile and wide area settings [210]. Eventual consistency leaves to the application programmer the burden of dealing with *anomalies* — i.e., behaviors deviating from that of an ideal linearizable execution. Hence, quite a large body of recent work aims to achieve a better understanding of its subtle implications [47, 49, 33, 39]. Since eventual consistency constrains only the *convergence* of replicas, it does not entail any guarantees about recency or ordering of operations. Burckhardt [65] proposes the following formal definition of eventual consistency:

$$\textsc{EventualConsistency}(\mathcal{F}) \triangleq \\ \textsc{EventualVisibility} \wedge \textsc{NoCircularCausality} \wedge \textsc{RVal}(\mathcal{F}) \quad (2.14)$$

where:

$$\textsc{EventualVisibility} \triangleq \forall a \in H, \forall [f] \in H/\approx_{ss}: \\ |\{b \in [f] : (a \xrightarrow{rb} b) \wedge (a \xrightarrow{vis} b)\}| < \infty \quad (2.15)$$

and

$$\textsc{NoCircularCausality} \triangleq acyclic(hb) \quad (2.16)$$

that is, the acyclic projection of hb, which we defined in Eq. 2.2. EventualVisibility mandates that, eventually, operation op will be visible to another operation op' invoked after the completion of op.

In an alternative attempt at clarifying the definition of eventual consistency, Shapiro et al. [214] identify the following properties:

- *Eventual delivery*: if some correct replica applies a write operation op, op is eventually applied by all correct replicas;
- *Convergence*: all correct replicas that have applied the same write operations eventually reach an equivalent state;
- *Termination*: all operations complete.

To this definition of eventual consistency, Shapiro et al. [214] add the following constraint:

- *Strong convergence*: all correct replicas that have applied the same write operations *have* equivalent state.

In other words, this last property guarantees that any two replicas that have applied the same (possibly unordered) set of writes will hold the same data. A storage system enforcing both eventual consistency and strong convergence is said to implement **strong eventual** consistency.

We capture strong convergence from the perspective of read operations, by requiring that reads which have the identical sets of visible writes return the same values.

$$\textsc{StrongConvergence} \triangleq \forall a, b \in H|_{rd} : vis^{-1}(a)|_{wr} = vis^{-1}(b)|_{wr} \Rightarrow a.oval = b.oval \tag{2.17}$$

Then, strong eventual consistency can be defined as:

$$\textsc{StrongEventualConsistency}(\mathcal{F}) \triangleq \textsc{EventualConsistency}(\mathcal{F}) \wedge \textsc{StrongConvergence} \tag{2.18}$$

Quiescent consistency [137] requires that if an object stops receiving updates (i.e. becomes quiescent), then the execution is equivalent to some sequential execution containing only complete operations. Although this definition resembles eventual consistency, it does not guarantee termination: a system that does not stop receiving updates will not reach quiescence, thus replicas convergence. Following Burckhardt [65], we formally define quiescent consistency as:

$$\textsc{QuiescentConsistency}(\mathcal{F}) \triangleq |H|_{wr}| < \infty \Rightarrow \exists C \in \mathcal{C} : \forall [f] \in H/\approx_{ss} : |\{op \in [f] : op.oval \notin \mathcal{F}(op, C)\}| < \infty \tag{2.19}$$

2.3.3 PRAM and sequential consistency

Pipeline RAM (**PRAM** or FIFO) consistency [175] prescribes that all processes see write operations issued by a given process in the same order as they were invoked by that process, as if they were in a pipeline — hence the name. However, processes may observe writes issued by different processes in different orders. Thus, no global total ordering is required. We define PRAM consistency by requiring the visibility partial order to be a superset of session order:

$$\text{PRAM} \triangleq so \subseteq vis \tag{2.20}$$

As proved by Brzezinski et al. [63], PRAM consistency is ensured *iff* the system provides read-your-write, monotonic reads and monotonic writes guarantees, which we will introduce in Section 2.3.4.

In a storage system implementing **sequential** consistency all operations are serialized in the same order at all replicas, and the ordering of operations determined by each process is preserved. Formally:

$$\textsc{SequentialConsistency}(\mathcal{F}) \triangleq \textsc{SingleOrder} \wedge \text{PRAM} \wedge \textsc{RVal}(\mathcal{F}) \tag{2.21}$$

Thus, sequential consistency, first defined by Lamport [161], is a guarantee of ordering rather than recency. Like linearizability, sequential consistency enforces a common global order of operations. Unlike linearizability, sequential consistency does not require real-time ordering of operations across different sessions: only the real-time ordering of operations invoked by the same process is preserved (as in PRAM consistency). [6] A quantitative comparison of the power and costs involved in the implementation of sequential consistency and linearizability is presented by Attiya and Welch [28].

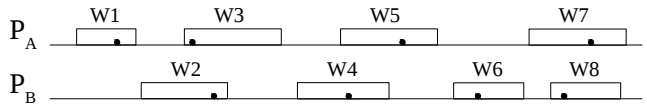


Figure 2.3 – An execution with processes issuing write operations on a shared object. Black spots are the chosen linearization points.

Figure 2.3 shows an execution with two processes issuing write operations on a shared object. Let us suppose that the two processes also continuously perform read operations. Each process will observe a certain serialization of the write operations. If we were to assume that the system respects PRAM consistency, those two processes might observe, for instance, the following two serializations:

$$S_{P_A}: \quad W1 \quad W2 \quad W3 \quad W5 \quad W4 \quad W7 \quad W6 \quad W8 \tag{S.1}$$

$$S_{P_B}: \quad W1 \quad W3 \quad W5 \quad W7 \quad W2 \quad W4 \quad W6 \quad W8 \tag{S.2}$$

If the system implemented sequential consistency, then S_{P_A} would be equal to S_{P_B} and it would respect the ordering of operations imposed by each writing process. Thus, any of (S.1) or (S.2) would be acceptable. On the other hand, assuming the system implements linearizability, and assigning linearization points as indicated by the points in Figure 2.3, (S.3) would be the only allowed serialization:

$$S_{Lin}: \quad W1 \quad W3 \quad W2 \quad W4 \quad W5 \quad W6 \quad W8 \quad W7 \tag{S.3}$$

2.3.4 Session guarantees

Although originally defined by Terry et al. [228] in connection to *client* sessions, *session guarantees* may as well apply to situations in which the concept of session is more loosely defined and it just refers to a specific process' point of view on the execution. We note that previous works in literature have classified session guarantees as *client-centric models* [226].

[6] In Section 2.3.10 we present *processor* consistency, a model whose semantic strength stands between those of PRAM and sequential consistency.

Monotonic reads states that successive reads must reflect a non-decreasing set of writes. Namely, if a process has read a certain value v from an object, any successive read operation will not return any value written before v. Intuitively, a read operation can be served only by those replicas that have executed all write operations whose effects have already been observed by the requesting process. In effect, we can represent this by saying that, given three operations $a, b, c \in H$, if $a \xrightarrow{\text{vis}} b$ and $b \xrightarrow{\text{so}} c$, where b and c are read operations, then $a \xrightarrow{\text{vis}} c$, i.e. the transitive closure of vis and so is included in vis.

$$\begin{aligned}\textsc{MonotonicReads} &\triangleq \forall a \in H, \forall b, c \in H|_{rd} : a \xrightarrow{\text{vis}} b \wedge b \xrightarrow{\text{so}} c \Rightarrow a \xrightarrow{\text{vis}} c \\ &\triangleq (vis; so|_{rd \to rd}) \subseteq vis \end{aligned} \quad (2.22)$$

The read-your-writes guarantee, later renamed **read-my-writes** [230, 65], requires that a read operation invoked by a process can only be carried out by replicas that have already applied all writes previously invoked by the same process.

$$\begin{aligned}\textsc{ReadMyWrites} &\triangleq \forall a \in H|_{wr}, \forall b \in H|_{rd} : a \xrightarrow{\text{so}} b \Rightarrow a \xrightarrow{\text{vis}} b \\ &\triangleq so|_{wr \to rd} \subseteq vis \end{aligned} \quad (2.23)$$

Let us assume that two processes issue read and write operations on a shared object as in Figure 2.4.

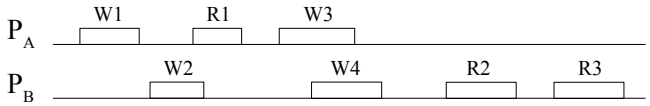


Figure 2.4 – An execution with processes issuing read and write operations on a shared object.

Given this execution, P_A and P_B could observe the following serializations, which satisfy the read-your-write guarantee but not PRAM consistency:

$$S_{P_A} : \quad W1 \quad W3 \quad W4 \quad W2 \quad \text{(S.4)}$$

$$S_{P_B} : \quad W2 \quad W4 \quad W3 \quad W1 \quad \text{(S.5)}$$

We note that some works in literature refer to *session consistency* as a special case of read-your-writes consistency that can be attained through *sticky* client sessions, i.e. those sessions in which the process always invokes operations on a given replica.

In a system that ensures **monotonic writes** a write is only performed on a replica if the replica has already applied all previous writes of the same session. In other words, replicas must

apply all writes belonging to the same session according to the order in which they were issued.

$$\textsc{MonotonicWrites} \triangleq \forall a, b \in H|_{wr} : a \xrightarrow{\text{so}} b \Rightarrow a \xrightarrow{\text{vis}} b \triangleq so|_{wr \to wr} \subseteq vis \quad (2.24)$$

Writes-follow-reads, sometimes called *session causality*, is somewhat the converse concept of read-your-write guarantee as it ensures that writes made during the session are ordered after any writes made by any process on any object whose effects were seen by previous reads in the same session.

$$\textsc{WritesFollowReads} \triangleq \forall a, c \in H|_{wr}, \forall b \in H|_{rd} : a \xrightarrow{\text{vis}} b \wedge b \xrightarrow{\text{so}} c \Rightarrow a \xrightarrow{\text{ar}} c \triangleq (vis; so|_{rd \to wr}) \subseteq ar \quad (2.25)$$

We note that some of the session guarantees embed specific notions of causality, and that in fact, as proved by Brzezinski et al. [64], causal consistency — which we describe next — requires and includes them all.

2.3.5 Causal models

The commonly accepted notion of potential causality in distributed systems is enclosed in the definition of *happened-before* relation introduced by Lamport [160]. According to this relation, two operations a and b are ordered if (a) they are both part of the same thread of execution, (b) b reads a value written by a, or (c) they are related by a transitive closure leveraging (a) and/or (b). This notion, originally defined in the context of message passing systems, has been translated to a consistency condition for shared-memory systems by Hutto and Ahamad [143]. The potential causality relation establishes a partial order over operations which we represent as hb in Eq. 2.2. Hence, while operations that are potentially causally [7] related must be seen by all processes in the same order, operations that are not causally related (i.e. causally concurrent) may be observed in different orders by different processes. In other words, **causal** consistency dictates that all replicas agree on the ordering of causally related operations [143, 11, 185]. This can be expressed as the conjunction of two predicates [65]:

- $\textsc{CausalVisibility} \triangleq hb \subseteq vis$
- $\textsc{CausalArbitration} \triangleq hb \subseteq ar$

Hence, causal consistency is defined as:

$$\textsc{Causality}(\mathcal{F}) \triangleq \textsc{CausalVisibility} \wedge \textsc{CausalArbitration} \wedge \textsc{RVal}(\mathcal{F}) \quad (2.26)$$

[7] While the most appropriate terminology would be "potential causality", for simplicity, hereafter we will use "causality".

Figure 2.5 represents an execution with two processes writing and reading the value of a shared object, with arrows indicating the causal relationships between operations.

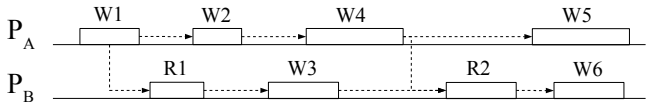


Figure 2.5 – An execution with two processes issuing operations on a shared object. Arrows express causal relationships between operations.

Assuming the execution respects PRAM but not causal consistency, we might have the following serializations:

$$S_{P_A}: \quad W1 \quad W2 \quad W4 \quad W5 \quad W3 \quad W6 \tag{S.6}$$
$$S_{P_B}: \quad W3 \quad W6 \quad W1 \quad W2 \quad W4 \quad W5 \tag{S.7}$$

With causal consistency (which implies PRAM), we could have obtained these serializations:

$$S_{P_A}: \quad W1 \quad W3 \quad W2 \quad W4 \quad W5 \quad W6 \tag{S.8}$$
$$S_{P_B}: \quad W1 \quad W2 \quad W3 \quad W4 \quad W6 \quad W5 \tag{S.9}$$

Recent work by Bailis et al. [34] promotes the use of explicit application-level causality, which is a subset of potential causality, [8] for building highly available distributed systems that would entail less overhead in terms of coordination and metadata maintenance. Furthermore, an increasing body of research has been drawing attention to causal consistency, considered an optimal tradeoff between user-perceived correctness and coordination overhead, especially in mobile or geo-replicated applications [177, 36, 252].

Causal+ (or convergent causal) consistency [177] mandates, in addition to causal consistency, that all replicas eventually and independently agree on the ordering of write operations. In fact, causally concurrent write operations may generate conflicting outcomes which in convergent causally consistent systems are handled in the same way by commutative and associative functions. Essentially, causal+ strengthens causal consistency with strong convergence (see Equation (2.17)), which mandates that all correct replicas that have applied the same write operations have equivalent state. In other words, causal+ consistency augments causal consistency with strong convergence in the same vein as strong eventual consistency [214] strengthens eventual consistency. Hence, causal+ consistency can be expressed as:

$$\textsc{Causal+}(\mathcal{F}) \triangleq \textsc{Causality}(\mathcal{F}) \wedge \textsc{StrongConvergence} \tag{2.27}$$

[8] As argued in [34], the application-level causality graph would be smaller in fanout and depth with respect to the traditional causal one, because it would only enclose relevant causal relationships, hinging on application-level knowledge and user-facing outcomes.

Real-time causal consistency has been defined in [185] as a stricter condition than causal consistency that enforces an additional condition: causally concurrent write operations that do not overlap in real-time must be applied according to their real-time order.

$$\textsc{RealTimeCausality}(\mathcal{F}) \triangleq \textsc{Causality}(\mathcal{F}) \wedge \textsc{RealTime} \tag{2.28}$$

where RealTime is defined as in Eq. E.1.

We note that although Lloyd et al. [177] classify real-time causal consistency as stronger than causal+ consistency, they are actually incomparable, as real-time causality — as defined by Mahajan et al. [185] — does not imply strong convergence. Of course, one can devise a variant of real-time causality that respects strong convergence as well.

Attiya et al. [30] define *observable causal* consistency as a strengthening of causal consistency for multi-value registers (MVR) that exposes the concurrency between operations when this concurrency may be inferred by processes from their observations. Observable causal consistency has also been proved to be the strongest consistency model satisfiable for a certain class of highly-available data stores implementing MVRs.

2.3.6 Staleness-based models

Intuitively, consistency models based on the notion of staleness allow reads to return old, *stale* written values. They provide stronger guarantees than eventually consistent semantics, but weak enough to allow for more efficient implementations than linearizability. In literature, two common metrics are employed to measure staleness: (real) time and data (object) versions.

To the best of our knowledge, the first definition of a consistency model explicitly dealing with time-based staleness is proposed by Singla et al. [218] as **delta** consistency. According to delta consistency, a write is guaranteed to become visible at most after $delta$ time units. Moreover, delta consistency is defined in conjunction with an ordering criterion (which is reminiscent of the *slow memory* consistency model, which we postpone to Section 2.3.9): writes to a given object by the same process are observed in the same order by all processes, but no global ordering is enforced for writes to a given object by different processes.

In an analogous way, *timed consistency* models, as defined by Torres-Rojas et al. [233], restrict the sets of values that read operations may return by the amount of time elapsed since the preceding writes. Specifically, in a *timed serialization* all reads occur *on time*, i.e. they do not return stale values when there are more recent ones that have been available for more than Δ units of time — Δ being a parameter of the execution. In other words, similarly to delta consistency, if a write operation is performed at time t, the value written by this operation must be visible by all processes by time $t + \Delta$.

Mahajan et al. [184] define a consistency condition named **bounded staleness** which is very similar to timed and delta semantics: a write operation of a given process becomes visible

to other processes no later than a fixed amount of time. However, this definition is also related to the use of a periodic message (i.e. a *beacon*) which allows each process to keep up with updates from other processes or to *suspect* of missing updates.

The differences among delta consistency, timed reads and bounded staleness are just a matter of subtle operational details that derive from the diverse contexts and practical purposes for which those models were developed. Thus, we can capture in formal terms the core semantics expressed by delta consistency, timed consistency models and bounded staleness as the following condition:

$$\textsc{TimedVisibility}(\Delta) \triangleq \forall a \in H|_{wr}, \forall b \in H, \forall t \in Time : \\ a.rtime = t \wedge b.stime \geq t + \Delta \Rightarrow a \xrightarrow{\text{vis}} b \quad (2.29)$$

Timed causal consistency [232] guarantees that each execution respects the partial ordering of causal consistency and that all reads are *on time*, with tolerance Δ:

$$\textsc{TimedCausality}(\mathcal{F}, \Delta) \triangleq \textsc{Causality}(\mathcal{F}) \wedge \textsc{TimedVisibility}(\Delta) \quad (2.30)$$

As depicted in Figure 2.1, due to the timed visibility term, timed causal is stronger than causal consistency.

Similarly, **timed serial** consistency [232] combines the real-time global ordering guarantee with the timed serialization constraint. A timed serially consistent execution with $\Delta = 0$ is linearizable. Golab et al. [122] describe Δ**-atomicity**, a condition which is equivalent to timed serial consistency. Indeed, according to Δ-atomicity, read operations may return either the value written by the last preceding write, or the value of a write operation returned up to Δ time units ago. In a follow-up work, Golab et al. [123] propose a novel metric called Γ, which entails fewer assumptions and is more robust than Δ against clock skew. The corresponding consistency semantics, Γ**-atomicity**, expresses, like Δ-atomicity, a deviation in time of a given execution from a linearizable one having the same operations' outcomes. We formalize the core notion of Δ-atomicity, Γ-atomicity and timed serial consistency with the following predicate:

$$\textsc{TimedLinearizability}(\mathcal{F}, \Delta) \triangleq \\ \textsc{SingleOrder} \wedge \textsc{TimedVisibility}(\Delta) \wedge \textsc{RVal}(\mathcal{F}) \quad (2.31)$$

Figure 2.6 illustrates an execution featuring read operations whose outcomes depend on a fixed timing parameter Δ.

If we were to assume that, despite the timing parameter, P_A and P_B observed the following serialization:

$$S_{P_{A,B}} : \quad W2 \quad W6 \quad W1 \quad W3 \quad W4 \quad W5 \quad (S.10)$$

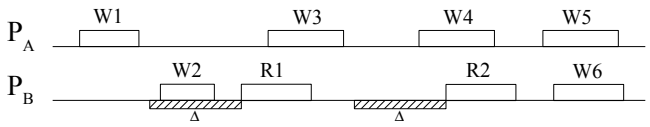


Figure 2.6 – An execution with processes issuing operations on a shared object. Hatched rectangles highlight the Δ parameter of staleness-based read operations.

then such execution would be sequentially consistent but it would not satisfy the timed serial consistency requirements. Thus, this execution serves as hint of the relative strengths of the sequential and timed serial consistency models, as illustrated in Fig. 2.1.

Prefix consistency [229, 227], also called **timeline** consistency [90], guarantees that readers observe an ordered sequence of writes, which may not be the most recent ones. It expresses an ordering rather than a recency constraint on writes: the read value is the result of a specific sequence of writes in an order agreed by all replicas. This pre-established order is supposedly reminiscent of that one imposed by sequential consistency. Thus, we could rename prefix consistency as *prefix sequential* consistency, whereas a version abiding real-time constraints would be called *prefix linearizable* consistency. Formally, we describe prefix sequential consistency as:

$$\textsc{PrefixSequential}(\mathcal{F}) \triangleq \textsc{SingleOrder} \wedge \textsc{MonotonicWrites} \wedge \textsc{RVal}(\mathcal{F}) \tag{2.32}$$

where the term named MonotonicWrites implies that the ordering of writes belonging to the same session is respected, as defined in Eq. 2.24. Similarly, we express prefix linearizable consistency as:

$$\textsc{PrefixLinearizable}(\mathcal{F}) \triangleq \textsc{SingleOrder} \wedge \textsc{RealTimeWW} \wedge \textsc{RVal}(\mathcal{F}) \tag{2.33}$$

where

$$\textsc{RealTimeWW} \triangleq rb|_{wr \to wr} \subseteq ar \tag{2.34}$$

In a study on quorum-based replicated systems with malicious faults, Aiyer et al. [13] formalize relaxed semantics that tolerate limited version-based staleness. Substantially, K-**safe**, K-**regular** and K-**atomic** (or K-**linearizability**) generalize the register consistency conditions previously introduced in [163] and described in Section 2.3.1, by permitting reads non-overlapping concurrent writes to return one of the latest K values written. For instance K-**linearizability** can be formalized as: [9]

$$\begin{aligned}&\text{K-}\textsc{Linearizable}(\mathcal{F}, K) \triangleq \\ &\qquad \textsc{SingleOrder} \wedge \textsc{RealTimeWW} \wedge \text{K-}\textsc{RealTimeReads}(K) \wedge \textsc{RVal}(\mathcal{F})\end{aligned} \tag{2.35}$$

[9] Strictly speaking, K-linearizability implicitly assumes K initial writes (i.e. writes with input value $\perp$) [13].

where

$$\text{K-RealTimeReads}(K) \triangleq \forall a \in H|_{wr}, \forall b \in H|_{rd}, \forall PW \subseteq H|_{wr}, \forall pw \in PW : \\ |PW| < K \wedge a \xrightarrow{ar} pw \wedge pw \xrightarrow{rb} b \wedge a \xrightarrow{rb} b \Rightarrow a \xrightarrow{ar} b \quad (2.36)$$

Upon these results, Bailis et al. [35] built a probabilistic model to predict the staleness of reads performed on eventually consistent quorum-based stores. They define Probabilistically Bounded Staleness (PBS) in terms of ***k*-staleness** and ***t*-visibility**. The former describes a probabilistic model that restricts the staleness of values returned by read operations. The latter probabilistically limits the time before a write becomes visible. The combination of these two models is named PBS $\langle k, t \rangle$**-staleness**. In a sense, PBS k-staleness is a probabilistic weakening of K-atomicity, i.e. the one that with probability equal to 1 becomes K-linearizability. Similarly, PBS t-visibility is a probabilistic weakening of timed visibility.

2.3.7 Fork-based models

The trust limitations that arise in the context of outsourced storage and computation [71, 238] have revamped the research on algorithms and protocols that deal with Byzantine faults [165]. In the Byzantine fault model, faulty processes and replicas may tamper with data or perform other arbitrary operations (within the limits of cryptography) in order to deliberately disrupt executions.

Consequently, new consistency models were defined to reshape the correctness goals in accordance to what is attainable when coping with such strong fault assumptions. When dealing with several untrusted storage repositories, Byzantine fault tolerance can be applied to mask certain fault patterns [238, 53] and even provide strong consistency semantics (e.g., [54] or see Chapter 3). However, when dealing with a single untrusted store, the situation is different and the consistency semantics needs to be relaxed [71]. Feasible consistency semantics for correct clients that interact with untrusted storage have been captured within the family of *fork-based* consistency models. In a nutshell, systems dealing with untrusted storage aim at providing linearizability when the storage is correct, while (gracefully) degrading to weaker consistency models — specifically, fork-based models — when the storage exhibits Byzantine faults.

The forefather of this family of models is **fork** (or fork-linearizable) consistency, introduced by Mazières and Shasha [189]. In short, a fork-linearizable system guarantees that if the storage system causes the visible histories of two processes to differ even for a single operation, they may never again observe each other's writes after that without the server being exposed as faulty. Any divergence in the histories observed by different groups of correct processes can be easily spotted by using any available communication protocol between them (e.g., out-of-band communication, gossip protocols, etc.). Fork-linearizability respects session order (PRAM

semantics) and real-time arbitration, thus it can be expressed as follows:

$$\textsc{ForkLinearizability}(\mathcal{F}) \triangleq \text{PRAM} \wedge \textsc{RealTime} \wedge \textsc{NoJoin} \wedge \textsc{RVal}(\mathcal{F}) \tag{2.37}$$

where the NoJoin predicate stipulates that clients whose sequences of visible operations (also called *views*) have been forked by an adversary, cannot be joined again:

$$\begin{aligned}\textsc{NoJoin} \triangleq \forall a_i, b_i, a_j, b_j \in H : a_i \not\approx_{ss} a_j \wedge (a_i, a_j) \in ar \setminus vis \wedge a_i \preceq_{so} b_i \wedge a_j \preceq_{so} b_j \\ \Rightarrow (b_i, b_j), (b_j, b_i) \notin vis\end{aligned} \tag{2.38}$$

The weaker **fork*** consistency model was defined by Li and Mazières [173] to allow for better performance and liveness guarantees. Fork* consistency allows forked groups of processes to observe at most one common operation issued by a certain correct process.

$$\textsc{Fork*}(\mathcal{F}) \triangleq \textsc{ReadYourWrites} \wedge \textsc{RealTime} \wedge \textsc{AtMostOneJoin} \wedge \textsc{RVal}(\mathcal{F}) \tag{2.39}$$

where

$$\begin{aligned}\textsc{AtMostOneJoin} \triangleq &\forall a_i, a_j \in H : a_i \not\approx_{ss} a_j \wedge (a_i, a_j) \in ar \setminus vis \Rightarrow \\ &\wedge |\{b_i \in H : a_i \preceq_{so} b_i \wedge (\exists b_j \in H : a_j \preceq_{so} b_j \wedge b_i \xrightarrow{vis} b_j\}| \leq 1 \\ &\wedge |\{b_j \in H : a_j \preceq_{so} b_j \wedge (\exists b_i \in H : a_i \preceq_{so} b_i \wedge b_j \xrightarrow{vis} b_i\}| \leq 1\end{aligned} \tag{2.40}$$

Notice that, unlike fork-linearizability, fork* does not respect the monotonicity of reads (and hence PRAM) [73].

Fork-sequential consistency [196, 72] requires that whenever an operation becomes visible to multiple processes, all these processes share the same history of operations occurring before that operation. Therefore, whenever a process reads a certain value written by another process, the reader is guaranteed to share with the writer process the set of visible operations that precede that write operation. Essentially, similarly to sequential consistency, a global order of operations is ensured, up to a common visible operation. Formally:

$$\textsc{ForkSequential}(\mathcal{F}) \triangleq \text{PRAM} \wedge \textsc{NoJoin} \wedge \textsc{RVal}(\mathcal{F}) \tag{2.41}$$

Mahajan et al. [184] define **fork-join causal** consistency (FJC) as a weaker variant of causal consistency that can preserve safeness and availability in spite of Byzantine faults. In a fork-join causal consistent storage system if a write operation op issued by a correct process depends on a write operation op' issued by any process, then, at every correct process, op' becomes visible before op. In other words, FJC enforces causal consistency among correct processes, and allows partitioned groups of processes to reconcile their histories through merging policies. Furthermore, inconsistent writes by a Byzantine process are treated as concurrent writes by

multiple *virtual processes*. **Bounded fork-join causal** [185] refines this by limiting the number of forks accepted from a faulty node, thus bounding the number of virtual nodes needed to represent a faulty node.

Finally, **weak fork-linearizability** [73] relaxes fork-linearizability in two ways: (1) after being partitioned, two processes may share the visibility of one more operation (i.e. *at-most-one-join*, as in fork* consistency), and (2) the real-time order of the last visible operation by each process might not be preserved (i.e. *weak real-time order*). This enables improved liveness guarantees (i.e. *wait freedom*). Weak fork-linearizability can be expressed as:

$$\textsc{WeakForkLin}(\mathcal{F}) \triangleq \text{PRAM} \wedge \text{K-}\textsc{RealTime}(2) \wedge \textsc{AtMostOneJoin} \wedge \textsc{RVal}(\mathcal{F}) \tag{2.42}$$

where K-RealTime(2) predicate is similar to K-RealTimeReads(2) defined in Equation 2.36, but generalized to all operations (i.e. with quantifier $\forall op \in H$). We note that weak fork-linearizability and fork* consistency are incomparable [73].

2.3.8 Composite and tunable semantics

To bridge the gap between strongly consistent and efficient implementations, several works propose consistency models that use different semantics in an adaptive fashion according to the tradeoffs between performance and correctness. [10]

The idea of distinguishing operations' consistency requirements by their semantics dates back to the shared-memory systems era. In that context, consistency models that employed different ordering constraints depending on operation type (e.g., *acquire* and *release*, rather than read/write data accesses) were called *hybrid*, whereas those that did not were referred to as *uniform* [193, 103, 116].

A similar approach was used by Attiya and Friedman [27] for consistency of shared-memory multiprocessors. **Hybrid** consistency is defined as a model requiring a concerted adoption of weak and strong consistency semantics. In a hybrid consistent system, *strong* operations are guaranteed to be seen in some sequential order by all processes (as in sequential consistency), whereas *weak* operations are designed to be fast, and they eventually become visible by all processes (much like in eventual consistency). Weak operations, however, are ordered with respect to strong operations: if two operations belong to the same session and one of them is strong, then their relative order is the same for all processes.

In a similar manner, Ladin et al. [156] assign an ordering type to each operation: *causal* operations respect causality ordering among them; *forced* operations are delivered in the same order at all relicas; *immediate* operations are performed as they return and they are delivered by each replica in same order with respect to all other operations.

[10] We do not formulate formal definitions for tunable semantics considering that they can be expressed by combining the logical predicates reported in the rest of the paper.

Eventual serializability [11] is described by Fekete et al. [109]. It requires a partial ordering of operations that is gradually strengthened, and eventually settles to a total order. A given operation is *strict* or *non-strict*. A strict operation is required to be *stable* as soon as it returns, whereas non-strict ones may be reordered afterwards. An operation is said to be *stable* if the prefix of operations preceding it has reached its final total order. Fekete et al. [109] envision an implementation in which a process attaches to an invoked operation the list of identifiers of operations that must be ordered before such operation, and a flag that indicates its type (i.e. strict or non-strict). The final global and total order of operations is similar to the sequential consistency ordering as no notion of real-time is involved.

Similarly, Serafini et al. [212] distinguish *strong* and *weak* operations. While strong operations are immediately linearized, weak ones are linearized only eventually. Weak operations are thus said to respect **eventual linearizability**. Weak operations are designed to terminate despite failures, and therefore they can be temporarily ordered in an arbitrary manner, thus violating linearizability. Ultimately, all operations gravitate towards a total order that satisfies real-time constraints.

Krishnamurthy et al. [155] propose a QoS model that allows client applications of a distributed storage system to express their consistency requirements. Accordingly, a client is directed towards a specific group of replicas implementing synchronous or lazy replication schemes, thus applying strong or weak consistency semantics. This system is said to provide **tunable** consistency.

In the same vein, Li et al. [170] propose **RedBlue** consistency. With RedBlue consistency operations are flagged as *blue* or *red* depending on several conditions such as their commutativity and the respect of invariants. According to such classification, operations are executed locally and lazily replicated, or serialized through synchronous coordination. Furthermore, all operations respect causal consistency and operations marked as red follow a total order among themselves. In a follow-up work, Li et al. [171] implement and evaluate a system that relieves the programmer from having to choose the right consistency for each operation, by exploiting static and dynamic code analysis.

Yu and Vahdat [249] propose a consistency spectrum based on three metrics: *staleness*, *order error* and *numerical error*. These metrics are embedded in a **conit** (portmanteau of "consistency unit"), which is a three-dimensional vector that quantifies the divergence from an ideal linearizable execution. Numerical error accounts for the number of writes that are already globally applied but not yet propagated to a given replica. Order error quantifies the number of writes at any replica that are subject to reordering, while staleness bounds the real-time delay of writes propagation. These metrics aim to capture the fundamental dimensions of consistency on the general requirement of agreement on state and update ordering. Note that, according

[11] We remark that, despite its name, eventual serializability is defined for non-transactional storage systems.

to this model, and unlike timed consistency (see Section 2.3.6), time-based staleness is defined from replicas' viewpoint rather than with respect to the timing of individual operations.

Similarly, Santos et al. [211] quantify the divergence of data object replicas by using a three-dimensional consistency vector. Originally designed for distributed multiplayer games, **vector-field** consistency mandates for each object a vector $\kappa = [\theta, \sigma, \nu]$ that bounds its staleness in a particular *view* of the virtual world. In particular, the vector establishes the maximum divergence of replicas in time (θ), number of updates (σ), and object value (ν). Unlike conit, this model brings about a notion of locality-awareness as it describes consistency as a vector field deployed throughout a virtual environment.

Later works put forward tunable consistency as a suitable model for cloud storage, since it enables more flexible quality of service (QoS) policies and service-level agreements (SLAs). Kraska et al. [154] propose consistency **rationing**, which entails adapting the consistency level at runtime by considering economic concerns. Similarly, Chihoub et al. [79] explore the possibility of a self-adaptive protocol that dynamically adjusts consistency to meet the application needs. In a sequent work, Chihoub et al. [80] add the monetary cost to the equation and study its tradeoffs with consistency in cloud settings. Terry et al. [230] advocate the use of declarative consistency-based SLAs that allows users of cloud key-value stores to attain an improved awareness of the inherent performance-correctness tensions. A similar approach has been subsequently implemented as a declarative programming model for tunable consistency by Sivaramakrishnan et al. [219].

In an attempt at proposing stronger semantics for geo-replicated storage, Balegas et al. [41] introduce *explicit* consistency. Besides providing eventual consistency, explicit consistency ensures that application-specific correctness rules (i.e. *invariants*) be respected during executions. In a follow-up work, Gotsman et al. [125] propose a proof rule to assign fine-grained restrictions on operations in order to respect data integrity invariants.

Finally, in the context of composite consistency models, it is worth mentioning systems that turn eventual consistency of data (offered by modern cloud storage services) into stronger semantics, by relying on small volumes of metadata kept in a separate linearizable store. In independent efforts, this technique was recently proposed under the names of *consistency anchoring* [54] and *consistency hardening* [98] — a full-fledged system implementing the latter is described in Chapter 3.

2.3.9 Per-object semantics

Per-object (or per-key) semantics have been defined to express consistency constraints on a per-object basis. Intuitively, per-object ordering semantics allow for more efficient implementations than global ordering semantics, by taking advantage of techniques such as sharding and state partitioning.

Slow memory, defined by Hutto and Ahamad [143], is a weaker variant of PRAM consistency. It requires that all processes observe the writes of a given process to a given object in the same order. In other words, slow memory is a per-object weakening of PRAM consistency:

$$\textsc{PerObjectPRAM} \triangleq (so \cap ob) \subseteq vis \tag{2.43}$$

An important concept in the family of per-object semantics is **coherence** [116] (or cache consistency [124]), which was first introduced as correctness condition of memory hierarchies in shared-memory multiprocessor systems [103]. Coherence ensures that what has been written to a specific memory location becomes visible in some sequential order by all processors, possibly through their local caches. In other words, coherence requires operations to be globally ordered on a per-object basis. A very similar concept has been adopted in recent work [90, 177] as **per-record timeline** consistency. This condition, described in relation to replicated storage, ensures that for each individual key (or object), all processes observe the same ordering of operations. Formally, we capture such condition with the following predicate:

$$\textsc{PerObjectSingleOrder} \triangleq \\ \exists H' \subseteq \{op \in H : op.oval = \nabla\} : ar \cap ob = vis \cap ob \setminus (H' \times H) \tag{2.44}$$

A system in which executions respect ordering of operations by a certain process on each object, and a global ordering of operations invoked on each object implements a semantics that we could name *per-object sequential* consistency:

$$\textsc{PerObjectSequential}(\mathcal{F}) \triangleq \\ \textsc{PerObjectSingleOrder} \wedge \textsc{PerObjectPRAM} \wedge \textsc{RVal}(\mathcal{F}) \tag{2.45}$$

Processor consistency, defined by Goodman [124] and formalized by Ahamad et al. [10], is expressed by two conditions: (a) writes issued by a process must be observed in the order in which they were issued, and (b) if there are two write operations to the same object, all processes observe these operations in the same order. These two conditions are in fact PRAM and per-record timeline consistency, thus:

$$\textsc{ProcessorConsistency}(\mathcal{F}) \triangleq \textsc{PerObjectSingleOrder} \wedge \textsc{PRAM} \wedge \textsc{RVal}(\mathcal{F}) \tag{2.46}$$

Finally, some works [192] mention *per-object linearizability*, which is in fact equivalent to linearizability on a per-object basis,due to its *locality* property [138].

We further note that one could compose other arbitrary consistency models by refining some of the predicates mentioned in this work to match only operations performed on individual

objects. As a case in point, Burckhardt et al. [68] describe **per-object causal** as: $hbo \triangleq ((so \cap ob) \cup vis)^+$.

2.3.10 Synchronized models

For completeness, in this section we overview semantics defined in the '80s and early '90s in order to establish correctness conditions of multiprocessor shared-memory systems. In order to exploit the computational parallelism of these systems, and, at the same time, to cope with the different performance of their components (e.g., memories, interconnections, processors, etc.), buffering and caching layers were adopted. A fundamental challenge of this kind of architecture is making sure that all memories reflect a common, consistent view of shared data. Thus, system designers employed *synchronization variables*, i.e. special shared objects that only expose two operations, named *acquire* and *release*. The synchronization variables are used as a generic abstraction to implement logical fences meant to coordinate concurrent accesses to shared data objects. In other words, synchronization variables protect the access to shared data through mutual exclusion by means of low level primitives (e.g., locks) or high-level language constructs (e.g., critical sections). While the burden of using these tools is left to the programmer, the system is supposed to distinguish the shared data accesses from those to the synchronization variables, possibly by implementing and exposing specific low level instructions. We note that some of the semantics defined in this section have inspired the models in use in modern CPUs to describe instruction reorderings that can be applied for throughput optimization.

Sequential consistency [161] (which we defined in Section 2.3.3) was originally adopted as ideal correctness condition for multiprocessors shared-memory systems. **Weak ordering** [12] as described by Dubois et al. [103], represents a convenient weakening of sequential consistency that introduces performance improvements. In a system that implements weak ordering: (a) all accesses to synchronization variables must be *strongly ordered*, (b) no access to a synchronization variable is allowed before all previous reads have been completed, and (c) processes cannot perform reads before issuing an access to a synchronization variable. In particular, Dubois et al. [103] define operations as *strongly ordered* if they comply with two specific criteria about session ordering and relatively to some special instructions supported by pipelined cache-based systems. Weak ordering has been subsequently redefined in terms of coordination requirements between software and hardware. Namely, Adve and Hill [5] define a *synchronization model* as a set of constraints on memory accesses that specify how and when synchronization needs to be enforced. Given this definition, "*a hardware is weakly ordered with respect to a given synchronization model if and only if it appears sequentially consistent to all software that obey the synchronization model.*"

[12] Some works in literature refer to weak ordering as to "weak consistency." We chose to avoid this equivocation by adopting its original nomenclature.

Release consistency [116] is a weaker extension of weak ordering that exploits detailed information about synchronization operations and non-synchronization accesses. Operations have to be labeled before execution by the programmer (or the compiler) as strong or weak. Hence, this widens the classification operated by weak ordering, which included just synchronization and non-synchronization labels. Similarly to hybrid consistency (see Section 2.3.8), strong operations are ordered according to processor or sequential consistency, whereas weak operations are restricted by the relative ordering of strong operations invoked by the same process.

Subsequently, several algorithms that slightly alter the original implementation of release consistency have been proposed. For instance, **lazy release** consistency [150] relaxes release consistency by postponing the enforcing of consistency from the release to the next acquire operation. The rationale of lazy release consistency is reducing the number of messages and the amount of data exchanged for coordination. Along the same lines, the protocol called *automatic update release consistency* [144] aims at improving performance over software-only implementations of lazy release consistency, by using a virtual memory mapped network interface.

Bershad and Zekauskas [51] define **entry** consistency by strengthening the relation between synchronization objects and the data which they guard. According to entry consistency, every object has to be guarded by a synchronization variable. Thus, in a sense, this model is a location-relative weakening of a consistency semantic, similarly to the models surveyed in Section 2.3.9. Moreover, entry consistency operates a further distinction of the synchronization operations in exclusive and non-exclusive. Thanks to these features, reads can occur with a greater degree of concurrency, thus enabling better performance.

Scope consistency [145] claims to offer most of the potential performance advantages of entry consistency, without requiring explicit binding of data to synchronization variables. The key intuition of scope consistency is the use of an abstraction called *scope* to implicitly capture the relationship between data and synchronization operations. Consistency scopes can be derived automatically from the use of synchronization variables in the program, thus easing the work of programmers.

With the definition of **location** consistency, Gao and Sarkar [115] forwent the basic assumption of *memory coherence* [116], i.e. the property that ensures that all writes to the same object are observed in the same order by all processes (see Section 2.3.9). Thus, they explored the possibility of executing multithreaded programs by just enforcing a partial order on writes to shared data. Similarly to entry consistency, in location consistency each object is associated to a synchronization variable. However, thanks to the relaxed ordering constraint, Gao and Sarkar [115] prove that location consistency is more efficient and equivalently strong when it is applied to settings with low data contention between processes.

2.4 Related work

Several works in literature have provided overviews of consistency models. In this section, we discuss these works and classify them according to their different perspectives.

Shared-memory systems Gharachorloo et al. [116] proposed a classification of shared memory access policies, specifically regarding their concurrency control semantics (e.g., the use of synchronization operations versus read/write accesses). Mosberger [193] adopted this classification to conduct a study on the memory consistency models popular at that time and their implementation tradeoffs. Adve and Gharachorloo [6] summarized in a practical tutorial the informal definitions and related issues of consistency models most commonly adopted in shared-memory multiprocessor systems.

Several subsequent works developed uniform frameworks and notations to represent consistency semantics defined in literature [7, 205, 43]. Most notably, Steinke and Nutt [222] provide a unified theory of consistency models for shared memory systems based on the composition of few fundamental declarative properties. In turn, this declarative and compositional approach outlines a partial ordering over consistency semantics. Similarly, a treatment of composability of consistency conditions has been proposed in [112]. On this subject, in Chapter 3 we illustrate the design and evaluation of Hybris, a storage system that takes advantage of a composition of different semantics.

While all these works proved to be valuable and formally sound, they represent only a limited portion of the consistency semantics relevant to modern non-transactional storage systems.

Distributed database systems In more recent years, researchers have proposed categorizations of the most influential consistency models for modern storage systems. Namely, Tanenbaum and van Steen [226] proposed the client-centric versus data-centric classification, while Bermbach and Kuhlenkamp [46], expanded such classification and provided descriptions for the most popular models. While practical and instrumental in attaining a good understanding of the consistency spectrum, these works propose informal treatments based on a simple dichotomous categorization which falls short of capturing some important consistency semantics. With the survey presented in this chapter, we aim at improving over these works, as we adopt a formal model based on first-order logic predicates and graph theory. We derive this model from the one proposed by Burckhardt [65], which we refined and expanded in order to enable the definition of a wider and richer range of consistency semantics. In particular, whereas Burckhardt [65] focuses mostly on session and eventual semantics, we cover a broader ground, including more than 50 different consistency semantics. We further note that some of our definitions — notably, those of sequential consistency, causal consistency and the session

guarantees — differ from those provided by Burckhardt, as we strived to provide a more accurate correspondence with the original definitions.

Measuring consistency In a concurrent trend, researchers have been straining to design uniform and rigorous frameworks to measure consistency in both shared memory systems and, more recently, in distributed database systems. Namely, while some works have proposed metrics to assess consistency [249, 123], others have devised methods to verify, given an execution, whether it satisfies a certain consistency model [191, 118, 23]. Finally, due to the loose definitions and opaque implementations of eventual consistency, recent research has tried to quantify its inherent anomalies as perceived from a client-side perspective [239, 198, 47, 203, 181]. In this regard, our work provides a more comprehensive and structured overview of the metrics that can be adopted to evaluate consistency. As an example, in Chapter 4 we describe a principled approach to consistency verification based on this framework.

Transactional systems Readers interested in pursuing a formal treatment of consistency models for transactional storage systems may refer to [8]. Similarly, other works by Harris et al. [134] and by Dziuma et al. [105] complement this survey with overviews on semantics specifically designed for transactional memory systems. Finally, some recent research [67, 77, 78] adopted variants of the same framework used in this chapter to propose axiomatic specifications of transactional consistency models.

2.5 Summary

In this chapter, we presented a comprehensive overview of the consistency models for non-transactional storage systems. Thanks to our principled approach, we were able to highlight subtle yet meaningful differences among consistency models, which will help scholars and practitioners attain a better understanding of the tradeoffs involved.

In order to describe consistency semantics, we adopted a mathematical framework based on graph theory and first-order logic. We developed such formal framework as an extension and refinement of the one proposed by Burckhardt [65]. The framework elements aptly capture the interplay of different factors involved in the executions of distributed storage systems.

We used this framework to propose formal definitions for the most popular of the over 50 consistency semantics we analyzed. For the rest of them, we presented informal descriptions which provide insights about their feature and relative strengths. Moreover, we clustered semantics according to criteria which account for their natures and common traits. In turn, both the clustering and the formal definitions helped us building a partial ordering of consistency models (see Figure 2.1). We believe this partial ordering of semantics will prove convenient both in designing more precise and coherent models, and in evaluating and comparing the correctness of systems already in place. In this regard, in Chapter 4 we will introduce a declarative approach to consistency verification based on the model just described.

As further contribution, we provide in Appendix C an ordered list of all semantics analyzed in this work, along with references to articles containing their definitions or describing their implementations in research literature. Finally, Appendix A conveniently lists all the logic predicates formulated in this section.

Chapter 3

Robust and Strongly Consistent Hybrid Cloud Storage

In this chapter, we present Hybris: a hybrid cloud storage system that improves over the reliability of modern cloud storage, and offers stronger consistency semantics (i.e. linearizability — which we formally defined in Sec. 2.3.1). We illustrate Hybris' design and evaluate its performance in different contexts and with respect to state-of-the-art prototypes. In Appendix D we detail the correctness proofs and algorithms of Hybris.

3.1 Introduction

Hybrid cloud storage entails storing data on private premises as well as on one (or more) remote, public cloud storage platforms. To enterprises, such hybrid design brings the best of both worlds: the benefits of public cloud storage (e.g., elasticity, flexible payment schemes and disaster-safe durability), in addition to a fine-grained control over confidential data. For example, an enterprise can keep private data on premises while storing less sensitive data at potentially untrusted public clouds. In a sense, the hybrid cloud approach eliminates to a large extent various security concerns that companies have with entrusting their data to commercial clouds [236]. As a result, enterprise-class hybrid cloud storage solutions are booming, with all leading storage providers, such as Dell EMC, [1] IBM, [2] NetApp, [3] Microsoft [4] and others, offering proprietary solutions.

Besides security and trust concerns, storing data on a single cloud presents issues related to reliability [128], performance, vendor lock-in [26, 4], as well as consistency, since cloud storage

[1] https://www.emc.com/en-us/cloud/hybrid-cloud-computing/index.htm.
[2] https://www.ibm.com/cloud-computing/bluemix/hybrid.
[3] http://www.netapp.com/us/solutions/cloud/hybrid-cloud/index.aspx.
[4] https://www.microsoft.com/en-us/cloud-platform/hybrid-cloud.

services are notorious for typically providing only eventual consistency [237, 48]. To address these concerns, several research works considered storing data *robustly* in public clouds, by leveraging *multiple* cloud providers [26, 238]. In short, these *multi-cloud* storage systems, such as DepSky [53], ICStore [42], SPANStore [245] and SCFS [52], leverage multiple public cloud providers to distribute trust, increase reliability, availability and consistency guarantees. A significant advantage of the multi-cloud approach is that it is implemented on the client side and as such, it demands no big investments into additional storage solutions.

However, existing robust multi-cloud storage systems suffer from serious limitations. Often, the robustness of these systems is limited to tolerating cloud outages, but not arbitrary or malicious behavior in clouds (e.g., data corruptions) [42, 245]. Other multi-cloud systems that do address arbitrary faults [53, 52] require prohibitive costs, as they rely on $3f + 1$ clouds to mask f faulty ones. This is a significant overhead with respect to tolerating only cloud outages, which makes these systems expensive to use in practice. Moreover, all existing multi-cloud storage systems scatter metadata across public clouds, increasing the difficulty of storage management, and impacting performance and costs.

In this chapter, we unify the hybrid and the multi-cloud approaches, and present Hybris, [5] the first robust hybrid cloud storage system. By combining the hybrid cloud with the multi-cloud, Hybris effectively brings together the benefits of both paradigms, thereby increasing security, reliability and consistency. Additionally, the design of Hybris allows to withstand arbitrary cloud faults at the same price of tolerating only outages. Hybris exposes the de facto standard key-value store API, and is designed to seamlessly replace popular storage services such as Amazon S3 as backend of modern cloud applications. The key idea behind Hybris is to keep the *metadata* on private premises, including metadata related to data outsourced to public clouds. This approach not only grants more control over the data scattered across different public clouds, but also allows Hybris to significantly outperform existing multi-cloud storage systems, both in terms of system performance (e.g., latency) and storage cost, while providing strong consistency guarantees.

In summary, the salient features of Hybris are the following:

Tolerating cloud malice at the price of outages Hybris puts no trust in any public cloud provider. Namely, Hybris can mask arbitrary (including malicious) faults of up to f public clouds by replicating data on as few as $f + 1$ clouds in the common case (when the system is synchronous and without faults). In the worst case, that is, to cope with network partitions, cloud inconsistencies and faults, Hybris uses up to f additional clouds. This is in sharp contrast with existing multi-cloud storage systems that require up to $3f + 1$ clouds to mask f malicious ones [53, 52]. Additionally, Hybris uses symmetric-key

[5] Hybris, sometimes also transliterated from ancient Greek as 'hubris', means extreme pride or arrogance. In Greek mythology, Hybris describes heroic mortals striving to surpass the boundaries of their mortal nature, and/or defy the authority of gods.

encryption to preserve the confidentiality of outsourced data. The required cryptographic keys are stored on trusted premises and shared through the metadata service.

Efficiency Hybris is efficient and incurs low cost. In the common case, a Hybris write involves as few as $f + 1$ public clouds, whereas a read involves only a single cloud, even though all clouds are untrusted. Hybris achieves this using cryptographic hashes, and without relying on expensive cryptographic primitives. By storing metadata on local premises, Hybris avoids the expensive round-trips for lightweight operations that plagued previous multi-cloud systems. Finally, Hybris optionally reduces storage requirements by supporting erasure coding [208], at the expense of increasing the number of clouds involved.

Scalability The potential pitfall of adopting such a compound architecture is that private resources may represent a scalability bottleneck. Hybris avoids this issue by keeping the metadata footprint very small. As an illustration, the replicated variant of Hybris maintains about 50 bytes of metadata per key, which is an order of magnitude smaller than comparable systems [53]. As a result, the Hybris metadata service, residing on trusted premises, can easily support up to 30k write ops/s and nearly 200k read ops/s, despite being fully replicated for fault tolerance. Moreover, Hybris offers per-key multi-writer multi-reader capabilities thanks to wait-free [136] concurrency control, further boosting the scalability of Hybris compared to lock-based systems [245, 53, 52].

Strong consistency Hybris guarantees linearizability (i.e. atomic consistency) [139] of reads and writes even though public clouds may guarantee no more than eventual consistency [237, 48]. Weak consistency is an artifact of the high availability requirements of cloud platforms [120, 60], and is often cited as a major impediment to cloud adoption, since eventually consistent stores are notoriously difficult to program and reason about [33]. Even though some cloud stores have recently started offering strongly consistent APIs, this offer usually comes with significantly higher monetary costs (for instance, Amazon charges twice the price for strong consistency compared to weak [20]). In contrast, Hybris is cost-effective as it relies on strongly consistent metadata within a private cloud, which is sufficient to mask inconsistencies of the public clouds. In fact, Hybris treats a cloud inconsistency simply as an arbitrary fault. In this regard, Hybris implements one of the few known ways of composing consistency semantics in a practical and meaningful fashion.

We implemented Hybris as a Java application library. [6] To maintain its code base small and facilitate adoption, we chose to reliably replicate metadata by layering Hybris on top of the Apache ZooKeeper coordination service [142]. Hybris clients act simply as ZooKeeper clients — our system does not entail any modifications to ZooKeeper, hence easing its deployment. In

[6] The Hybris prototype is released as open source software [106].

addition, we designed Hybris metadata service to be easily portable from ZooKeeper to any SQL-based replicated RDBMS as well as NoSQL data store that exports a conditional update operation. As an example, we implemented an alternative metadata service using the Consul coordination service [87]. We evaluated Hybris using both micro-benchmarks and the YCSB [91] benchmarking framework. Our evaluation shows that Hybris significantly outperforms state-of-the-art robust multi-cloud storage systems, with a fraction of the cost and stronger consistency guarantees.

The rest of this chapter is organized as follows. In Section 3.2, we present the Hybris architecture and system model. Then, in Section 3.3, we provide the algorithmic details of the Hybris protocol. In Section 3.4 we discuss Hybris implementation and optimizations, on whose performance we report in Section 3.5. We provide a discussion on related work in Section 3.6. Pseudocode of algorithms and correctness arguments are postponed to Appendix D.

3.2 Hybris overview

The high-level design of Hybris is presented in Figure 3.1. Hybris mixes two types of resources: 1) private, trusted resources that provide computation and limited storage capabilities and 2) virtually unlimited untrusted storage resources in outsourced clouds. We designed Hybris to leverage commodity cloud storage APIs that do not offer computation services, e.g., key-value stores like Amazon S3.

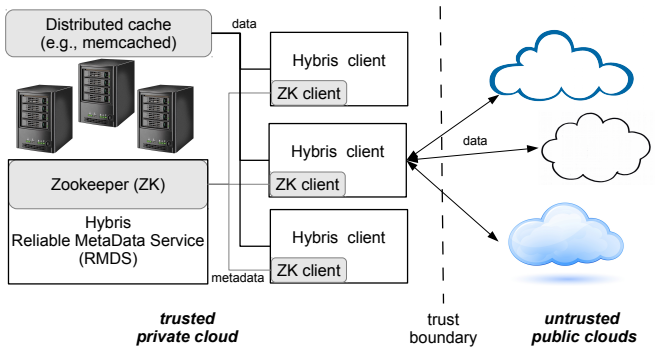


Figure 3.1 – Hybris architecture. Reused (open-source) components are depicted in grey.

Hybris stores data and metadata separately. Metadata is stored within the key component of Hybris called Reliable MetaData Service (RMDS). RMDS has no single point of failure and is assumed to reside on private premises. [7]

On the other hand, data is stored on untrusted public clouds. Hybris distributes data across multiple cloud storage providers for robustness, i.e. to mask cloud outages and malicious faults. In addition, Hybris caches data locally on private premises. While different caching solutions exist, our reference implementation uses Memcached [190], an open source distributed caching system. Finally, at the heart of the system is the Hybris client, whose library orchestrates the interactions with public clouds, RMDS and the caching service. The Hybris client is also responsible for encrypting and decrypting data, leveraging RMDS in order to share encryption keys (see Sec. 3.3.8).

In the following sections, we first specify our system model and assumptions. Then we define the Hybris data model and specify its consistency and liveness semantics.

3.2.1 System model

Fault model We assume a distributed system where any of the components might fail. In particular, we assume a dual fault model, where: (i) the processes on private premises (i.e. in

[7] We discuss and evaluate the deployment of RMDS across geographically distributed (yet trusted) data centers in Section 3.5.4.

the private cloud) can fail by crashing, [8] and (ii) we model public clouds as prone to arbitrary failures, including malicious faults [199]. Processes that do not fail are called *correct*.

Processes on private premises are clients and metadata servers. We assume that *any* number of clients and any minority of metadata servers can be (crash) faulty. Moreover, to guarantee availability despite up to f (arbitrary) faulty public clouds, Hybris requires at least $2f + 1$ public clouds in total. However, Hybris consistency (i.e. safety) is maintained regardless of the number of faulty public clouds.

For simplicity, we assume an adversary that can coordinate malicious processes as well as process crashes. However, the adversary cannot subvert the cryptographic hash (e.g., SHA-2), and it cannot spoof communication among non-malicious processes.

Timing assumptions Similarly to our fault model, our communication model is dual, with its boundary coinciding with the trust boundary (see Fig. 3.1). Namely, we assume communication within the private portion of the system as partially synchronous [104] (i.e. with arbitrary but finite periods of asynchrony), whereas communication between clients and public clouds is entirely asynchronous (i.e. does not rely on any timing assumption) yet reliable, with messages between correct clients and clouds being eventually delivered.

We believe that our dual fault and timing assumptions reasonably reflect typical hybrid cloud deployment scenarios. In particular, the accuracy of this model finds confirmations in recent studies about performance and faults of public clouds [128] and on-premise clusters [76].

Consistency Our consistency model is also dual. We model processes on private premises as classical state machines, with their computation proceeding in indivisible, atomic steps. On the other hand, we model clouds as eventually consistent stores [48] (see Sec. 2.3.2). Roughly speaking, eventual consistency guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value [237].

3.2.2 Hybris data model and semantics

Similarly to commodity public cloud storage services, Hybris exposes a key-value store (KVS) API. In particular, the Hybris address space consists of flat containers, each holding multiple keys. The KVS API consists of four main operations: *(i)* PUT$(cont, key, value)$, to put $value$ under key in container $cont$; *(ii)* GET$(cont, key)$, to retrieve the value associated with key; *(iii)* DELETE$(cont, key)$ to remove the key entry and *(iv)* LIST$(cont)$ to list the keys present in container $cont$. Moreover, Hybris supports transactional writes through the TPUT$(cont, \langle key_{lst} \rangle, \langle value_{lst} \rangle)$ API. We collectively refer to operations that modify storage

[8] We relax this assumption by discussing the suitability of the *cross fault tolerance* (XFT) model [176] in §3.4.3. In §3.5.3 we evaluate the performance of both crash fault and cross fault tolerant replication protocols.

state (e.g., PUT, TPUT and DELETE) as *write* operations, whereas the other operations (e.g., GET and LIST) are called *read* operations.

Hybris implements a multi-writer multi-reader key-value storage, and is strongly consistent, i.e. it implements *linearizable* [139] semantics (see Sec. 2.3.1). Linearizability (also known as atomic consistency) provides the illusion that the effect of a complete operation *op* takes place instantly at some point in time between its invocation and response. An operation invoked by a faulty client might appear either as complete or not invoked at all. Optionally, Hybris can be set to support weaker consistency semantics, which may enable better performance (see Sec. 3.3.10).

Although it provides strong consistency, Hybris is highly available. Hybris writes are *wait-free*, i.e. writes by a correct client are guaranteed to eventually complete [136]. On the other hand, a Hybris read operation by a correct client will always complete, except in the corner case where an infinite number of writes to the same key is concurrent with the read operation (this is called *finite-write termination* [3]). Hence, in Hybris, we trade read wait-freedom for finite-write termination and better performance. In fact, guaranteeing read wait-freedom reveals very costly in KVS-based multi-cloud storage systems [42] and significantly impacts storage complexity. We feel that our choice will not be limiting in practice, since FW-termination essentially offers the same guarantees as wait-freedom for a large number of workloads.

3.3 Hybris Protocol

In this section we present the Hybris protocol. We describe in detail how data and metadata are accessed by clients in the common case, and how consistency and availability are preserved despite failures, asynchrony and concurrency. We postpone the correctness proofs to Appendix D.

3.3.1 Overview

The key part of Hybris is the Reliable MetaData Store (RMDS), which maintains metadata associated with each key-value pair. Each metadata entry consists of the following elements: *(i)* a logical timestamp, *(ii)* a list of at least $f + 1$ pointers to clouds that store value v, *(iii)* a cryptographic hash of v ($H(v)$), and *(iv)* the size of value v.

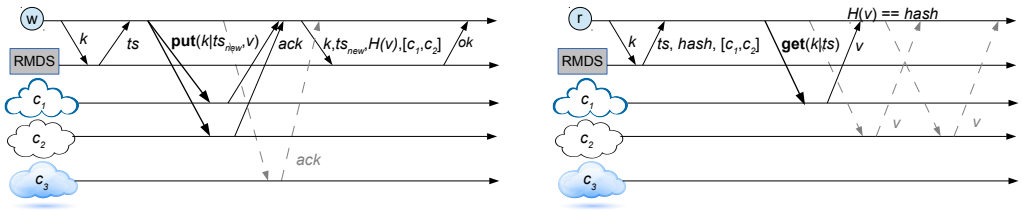
Despite being lightweight, the metadata is powerful enough to allow tolerating arbitrary cloud failures. Intuitively, the cryptographic hash within a trusted and consistent RMDS enables end-to-end integrity protection: neither corrupted nor stale data produced by malicious or inconsistent clouds are ever returned to the application. Additionally, the data size entry helps prevent certain denial-of-service attack vectors by a malicious cloud (see Sec. 3.4.4).

Furthermore, Hybris metadata acts as a directory pointing to $f + 1$ clouds, thus enabling a client to retrieve the correct value despite f of them being arbitrarily faulty. In fact, with Hybris, as few as $f + 1$ clouds are sufficient to ensure both consistency and availability of read operations (namely GET; see Sec. 3.3.3). Additional f clouds (totaling $2f + 1$ clouds) are only needed to guarantee that writes (i.e. PUT) are available as well in the presence of f cloud outages (see Sec. 3.3.2).

Finally, besides cryptographic hash and pointers to clouds, a metadata entry includes a timestamp that induces a total order on operations which captures their real-time precedence ordering, as required by linearizability. Timestamps are managed by the Hybris client, and consist of a classical multi-writer tag [182] comprising a monotonically increasing sequence number sn and a client id cid serving as tiebreaker.[9] The subtlety of Hybris lies in the way it combines timestamp-based lock-free multi-writer concurrency control within RMDS with garbage collection (Sec. 3.3.4) of stale values from public clouds (see Sec. 3.3.5 for details).

In the following we detail each Hybris operation. We assume that a given Hybris client never invokes multiple concurrent operations on the same key.

[9]We decided against leveraging server-managed timestamps (e.g., provided by ZooKeeper) to avoid constraining RMDS to a specific implementation. More details about RMDS implementations can be found in Sec. 3.4.



(a) PUT (k, v) (b) GET (k)

Figure 3.2 – Hybris PUT and GET protocol ($f = 1$).
Common-case is depicted in solid lines.

3.3.2 PUT protocol

Hybris PUT protocol consists of the steps illustrated in Figure 3.2(a). To write a value v under key k, the client first fetches the metadata associated with key k from RMDS. The metadata contains timestamp $ts = (sn, cid_i)$ of the latest authoritative write to k. The client computes a new timestamp $ts_{new} = (sn + 1, cid)$. Next, the client combines key k and timestamp ts_{new} to a new key $k_{new} = k|ts_{new}$ and invokes *put*(k_{new}, v) on $f + 1$ clouds in parallel. Concurrently, the clients starts a timer, set to the observed upload latency for an object of the same size. In the common case, the $f + 1$ clouds reply before the timer expires. Otherwise, the client invokes *put*(k_{new}, v) on up to f secondary clouds (dashed arrows in Fig. 3.2(a)). Once the client has received an ack from $f + 1$ different clouds, it is assured that the PUT is durable and can proceed to the final stage of the operation.

In the final step, the client attempts to store in RMDS the metadata associated with key k, consisting of timestamp ts_{new}, cryptographic hash $H(v)$, size of value v $size(v)$, and the list ($cloudList$) of pointers to those $f + 1$ clouds that have acknowledged storage of value v. This final step constitutes the *linearization point* of PUT, therefore it has to be performed in a specific way. Namely, if the client performs a straightforward update of metadata in RMDS, then this metadata might be overwritten by metadata with a *lower* timestamp (i.e. the so-called *old-new inversion* happens), breaking the timestamp ordering of operations and thus, violating linearizability. [10] In order to prevent this, we require RMDS to export an atomic conditional update operation. Hence, in the final step of Hybris PUT, the client issues a conditional update to RMDS, which updates the metadata for key k *only if* the written timestamp ts_{new} is *greater* than the one that RMDS already stores. In Section 3.4 we describe how we implemented this functionality over Apache ZooKeeper API and, alternatively, in the Consul-based RMDS instance. We note that any other NoSQL and SQL DBMS that supports conditional updates can be adopted to implement the RMDS functionality.

[10] Note that, since garbage collection (detailed in Sec. 3.3.4) relies on timestamp-based ordering to tell old values from new ones, old-new inversions could even lead to data loss.

3.3.3 GET in the common case

The Hybris GET protocol is illustrated in Figure 3.2(b). To read a value stored under key k, the client first obtains from RMDS the latest metadata for k, consisting of timestamp ts, cryptographic hash h, value size s, as well a list $cloudList$ of pointers to $f + 1$ clouds that store the corresponding value. The client selects the first cloud c_1 from $cloudList$ and invokes *get*($k|ts$) on c_1, where $k|ts$ denotes the key under which the value is stored. The client concurrently starts a timer set to the typically observed download latency from c_1 (given the value size s). In the common case, the client is able to download the value v from the first cloud c_1 before expiration of its timer. Once it receives value v, the client checks that v matches the hash h included in the metadata bundle (i.e. if $H(v) = h$). If the value passes this check, then the client returns it to the application and the GET completes.

In case the timer expires, or if the value downloaded from the first cloud does not pass the hash check, the client sequentially proceeds to downloading the data from another cloud from $cloudList$ (see dashed arrows in Fig. 3.2(b)) and so on, until it exhausts all $f + 1$ clouds from $cloudList$.[11] In some corner cases, caused by concurrent garbage collection (described in Sec. 3.3.4), failures, repeated timeouts (asynchrony), or clouds' inconsistency, the client must take additional actions, which we describe in Sec. 3.3.5.

3.3.4 Garbage collection

The purpose of garbage collection is to reclaim storage space by deleting obsolete versions of objects from clouds while allowing read and write operations to execute concurrently. Garbage collection in Hybris is performed by the client asynchronously in the background. Therefore, the PUT operation can return control to the application without waiting for the completion of garbage collection.

To perform garbage collection for key k, the client retrieves the list of keys prefixed by k from each cloud as well as the latest authoritative timestamp ts. This involves invoking *list*($k|*$) on every cloud and fetching the metadata associated with key k from RMDS. Then for each key k_{old}, where $k_{old} < k|ts$, the client invokes *delete*(k_{old}) on every cloud.

3.3.5 GET in the worst-case

In the context of cloud storage, there are known issues with weak (e.g., eventual [237]) consistency — see Sec. 2.3.2. With eventual consistency, even a correct, non-malicious cloud might deviate from linearizable semantics and return an unexpected value, typically a stale one.

[11] As we discuss in details in Sec. 3.4, in our implementation, clouds in $cloudList$ are ranked by the client by their typical latency in ascending order. Hence, when reading, the client will first read from the "fastest" cloud from $cloudList$ and then proceed to slower clouds.

In this case, the sequential common-case reading from $f + 1$ clouds as described in Section 3.3.3 might not return the correct value, since the hash verification might fail at all $f + 1$ clouds. In addition to the case of inconsistent clouds, this anomaly might also occur if: *(i)* the timers set by the client for otherwise non-faulty clouds expire (i.e. in case of asynchrony or network outages), and/or *(ii)* the values read by the client were concurrently garbage collected (see Sec. 3.3.4).

To address this issues, Hybris levereges strong metadata consistency to mask data inconsistencies in the clouds, effectively allowing availability to be traded off for consistency. To this end, the Hybris client indulgently reissues a *get* to all clouds in parallel, and waits to receive at least one value matching the required hash. However, due to possible concurrent garbage collection (Sec. 3.3.4), the client needs to make sure it always compares the values received from clouds to the most recent key's metadata. This can be achieved in two ways: *(i)* by simply iterating over the entire GET including metadata retrieval from RMDS, or *(ii)* by only repeating the *get* operations at $f + 1$ clouds while fetching metadata from RMDS only when it actually changes.

In Hybris, we adopt the latter approach. Notice that this implies that RMDS must be able to inform the client proactively about metadata changes. This can be achieved by having a RMDS that supports subscriptions to metadata updates, which is possible to achieve by using, e.g., Apache ZooKeeper and Consul (through the concept of *watch*, see Sec. 3.4 for details). This worst-case protocol is executed only if the common-case GET fails (Sec. 3.3.3), and it proceeds as follows:

1. The client first reads the metadata for key k from RMDS (i.e. timestamp ts, hash h, size s and cloud list $cloudList$) and subscribes for updates related to key k metadata.
2. The client issues a parallel *get*$(k|ts)$ to all $f + 1$ clouds from $cloudList$.
3. When a cloud $c \in$cloudList responds with value v_c, the client verifies $H(v_c)$ against h. [12]
   (a) If the hash verification succeeds, the GET returns v_c.
   (b) Otherwise, the client discards v_c and reissues *get*$(k|ts)$ to cloud c.

(*) At any point in time, if the client receives a metadata update notification for key k from RMDS, it cancels all pending downloads, and repeats the procedure from step 1.

The complete Hybris GET, as described above, ensures finite-write termination [3] in presence of eventually consistent clouds. Namely, a GET may fail to return a value only theoretically, i.e. in case of an infinite number of concurrent writes to the same key, in which case, garbage collection might systematically and indefinitely often remove every written value before the client manages to retrieve it. [13] We believe that this exceptional corner case is of marginal importance for the vast majority of applications.

[12] For simplicity, we model the absence of a value as a special NULL value that can be hashed.

[13] Notice that it is straightforward to modify Hybris to guarantee read availability even in case of an infinite number of concurrent writes, by switching off the garbage collection.

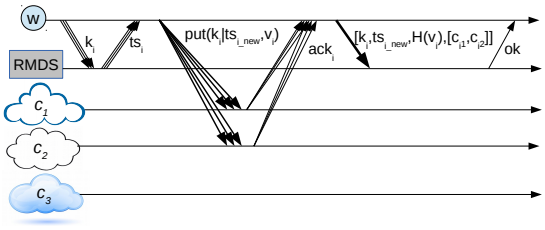


Figure 3.3 – Hybris transactional PUT protocol ($f = 1$).
Worst case communication patterns are omitted for clarity.

3.3.6 Transactional PUT

Hybris supports a transactional PUT operation that writes atomically to multiple keys. The steps associated with the transactional PUT operation are depicted in Figure 3.3.

Similarly to the normal PUT, the client first fetches the latest authoritative timestamps $[ts_0...ts_n]$ by issuing parallel requests to the RMDS for metadata of the concerned keys $[k_0...k_n]$. Each timestamp ts_i is a tuple consisting of a sequence number sn_i and a client id cid_i. Based on timestamp ts_i, the client computes a new timestamp ts_{i_new} for each key, whose value is $(sn_i + 1, cid_i)$. Next, the client combines each key k_i and timestamp ts_{i_new} to a new key $k_{i\_new} = k_i|ts_{i\_new}$ and invokes *put* (k_{i_new}, v_i) on $f + 1$ clouds in parallel. This operation is executed in parallel for each key to be written. Concurrently, the client starts a set of timers as for the normal PUT. In the common case, the $f + 1$ clouds reply to the client for each key in a timely fashion, before the timer expires. Otherwise, the client invokes *put* (k_{i_new}, v_i) to up to f secondary clouds. Once the client has received acknowledgments from $f + 1$ different clouds for each key, it is assured that the transactional PUT is durable and can thus proceed to the final stage of the operation.

In the final step, the client stores in RMDS the updated metadata associated with each key k_i, consisting of the timestamp ts_{i_new}, the cryptographic hash $H(v_i)$, and the list of pointers to the $f + 1$ clouds that have correctly stored v_i. As for the normal PUT operation, to avoid the so-called old-new inversion anomaly, we employ the conditional update exposed by RMDS. The metadata update succeeds only if, for each key k_i the written timestamp ts_{i_new} is greater than the timestamp currently stored for key k_i. In order to implement transactional atomicity, we wrap the metadata updates into an RMDS transaction. Specifically, we employ the MULTI API exposed by Apache ZooKeeper and the corresponding API in Consul. Thanks to this, if any of the single write to RMDS fails, the whole transactional PUT aborts. In this case, the objects written to the cloud stores are eventually erased by the normal garbage collection background task.

In summary, this approach implements an optimistic transactional concurrency control that, in line with the other parts of Hybris protocol, eschews locks to provide wait-freedom [136].

3.3.7 DELETE and LIST

The Hybris DELETE and LIST operations are local to RMDS, and do not access public clouds.

In order to delete a value, the client performs the PUT protocol with the special $cloudList$ value $\perp$ denoting the deletion. Deleting a value creates a metadata *tombstone* in RMDS, i.e. metadata that lack corresponding values in the cloud stores. Metadata tombstones are necessary to keep record of the latest authoritative timestamp associated with a given key, and to preserve per-key timestamp monotonicity. Deleted values are eventually removed from cloud stores by the normal garbage collection. On the other hand, the LIST operation simply retrieves from RMDS all the keys in the container $cont$ that are not associated with tombstone metadata.

3.3.8 Confidentiality

Ensuring data confidentiality [14] in Hybris is straightforward. During a PUT, just before uploading data to $f+1$ public clouds, the client encrypts the data with a symmetric cryptographic key k_{enc} which is then added to the metadata bundle. The hash is then computed on the ciphertext (rather than plaintext). The rest of PUT protocol remains unchanged. Notice that the client may generate a new encryption key at each PUT, or reuse the key stored in RMDS by previous PUT operations.

In order to decrypt data, a client uses the encryption key k_{enc} retrieved with the metadata bundle. Then, as the ciphertext downloaded from some cloud successfully passes the hash test, the client decrypts the data using k_{enc}.

3.3.9 Erasure coding

In the interest of minimizing bandwidth and storage space requirements, Hybris supports erasure coding. Erasure codes have been shown to provide resilience to failures through redundancy schemes which are significantly more efficient than replication [241]. Erasure codes entail partitioning data into $k > 1$ blocks with m additional parity blocks. Each of the $k + m$ blocks takes approximately $1/k$ of the original storage space. If the erasure code is *information-optimal*, the data can be reconstructed from any k blocks despite up to m erasures. In the context of cloud storage, blocks can be stored on different clouds and erasures correspond to arbitrary failures (e.g., network outages, data corruption, etc.). For simplicity, in Hybris we fix m to equal f.

Deriving an erasure coding variant of Hybris from its replicated counterpart is relatively straightforward. Namely, in a PUT operation, the client encodes original data into $f + k$ erasure-

[14] Oblivious RAM algorithms can provide further confidentiality guarantees by masking data access patterns [221]. However, we decided not to integrate those algorithms in Hybris since they require performing additional operations and using further storage space, which could hinder performance and significantly increase monetary costs.

coded blocks, and stores one block per cloud. Hence, with erasure coding, PUT involves $f + k$ clouds in the common case (instead of $f + 1$ with replication). Then, the client computes $f + k$ hashes (instead of a single hash as with replication) that are stored in the RMDS as part of the metadata. Finally, the erasure-coded GET fetches blocks from k clouds in the common case, with block hashes verified against those stored in RMDS. In the worst case, Hybris with erasure coding uses up to $2f + k$ (resp., $f + k$) clouds in PUT (resp., GET) operations.

Finally, it is worth noting that in Hybris the parameters f and k are independent. This offers more flexibility with respect to prior solutions which mandated $k \geq f + 1$.

3.3.10 Weaker consistency semantics

A number of today's cloud applications may benefit from improved performance in exchange for weaker consistency guarantees. Over the years, researchers and practitioners have defined these weaker consistency guarantees in a wide spectrum of semantics that we described in Chapter 2. Hybris exposes this consistency vs performance tradeoff to the application developers through an optional API. Specifically, Hybris implements two weaker consistency semantics: *read-my-writes* and *bounded staleness* consistency.

Read-my-writes In read-my-writes consistency [228] a read operation invoked by some client can be serviced only by replicas that have already applied all previous write operations by the same client. E-commerce shopping carts are typical examples of applications that would benefit from this consistency semantics. Indeed, customers only write and read their own cart object, and are generally sensitive to the latency of their operations [131].

This semantics is implemented in Hybris by leveraging caching. Essentially, a *write-through* caching policy is enabled in order to cache all the data written by each client. After a successful PUT, a client stores the written data in Memcached, under the compound key used for the clouds (i.e. $\langle k|ts_{new}\rangle$, see Sec. 3.3.2). Additionally, the client stores the compound key in a local in-memory hash table along with the original one (i.e. k). Later reads will fetch the data from the cache using the compound key cached locally. In this way, clients may obtaining previously written values without incurring the monetary and performance costs entailed by strongly consistent reads. In case of a cache miss, the client falls back to a normal read from the clouds as discussed in Sec. 3.3.3 and 3.3.5.

Bounded staleness According to the bounded staleness semantics, the data read from a storage system must be fresher than a certain threshold. This threshold can be defined in terms of data versions [122], or real-time [233]. Web search applications are a typical use case of this semantics, as they are latency-sensitive, yet they tolerate a certain bounded inconsistency.

Our bounded staleness protocol also makes use of the cache layer. In particular, to implement time-based bounded staleness we cache the written object on Memcached under the original

key k — instead of using, as for read-your-write, the compound key. Additionally, we instruct the caching layer to evict all objects older than a certain expiration period Δ.[15] Hence, all objects read from cache will abide the staleness restriction.

To implement version-based bounded staleness, we add a counter field to the metadata stored on RMDS, accounting for the number of versions written since the last caching operation. During a PUT, the client fetches the metadata from RMDS (as specified in Sec. 3.3.2) and reads this caching counter. In case of successful writes to the clouds, the client increments the counter. If the counter exceeds a predefined threshold η, the object is cached under its original key (i.e. k) and the counter is reset. When reading, clients will first try to read the value from the cache, thus obtaining, in the worst case, a value that is η versions older than the most recent one.

[15] Similarly to Memcached, most modern off-the-shelf caching systems implement this functionality.

3.4 Implementation

We implemented Hybris as an application library [106]. The implementation pertains solely to the Hybris client side since the entire functionality of the metadata service (RMDS) is layered on top of the Apache ZooKeeper client. Namely, Hybris does not entail any modification to the ZooKeeper server side. Our Hybris client is lightweight and consists of about 3800 lines of Java code. Hybris client interactions with public clouds are implemented by wrapping individual native Java SDK clients (drivers) for each cloud storage provider into a common lightweight interface that masks the small differences across the various storage APIs. [16]

In the following, we first discuss in detail our RMDS implementation with ZooKeeper and the alternative one using Consul; then we describe several Hybris optimizations that we implemented.

3.4.1 ZooKeeper-based RMDS

We layered our reference Hybris implementation over Apache ZooKeeper [142]. In particular, we durably store Hybris metadata as ZooKeeper *znodes*. In ZooKeeper, znodes are data objects addressed by *paths* in a hierarchical namespace. For each instance of Hybris we generate a root znode. Then, the metadata pertaining to Hybris container $cont$ is stored under ZooKeeper path $\langle root \rangle / cont$. In principle, for each Hybris key k in container $cont$, we store a znode with path $path_k = \langle root \rangle / cont / k$.

ZooKeeper offers a fairly modest API. The ZooKeeper API calls relevant to Hybris are the following:

- *create/setData*($p, data$) creates/updates a znode identified by path p with $data$.
- *getData*(p) is used to retrieve data stored under znode p.
- *sync()* synchronizes the ZooKeeper replica that maintains the client's session with the ZooKeeper leader, thus making sure that the read data contains the latest updates.
- *getChildren*(p) (only used in Hybris LIST) returns the list of znodes whose paths are prefixed by p.

Finally, ZooKeeper allows several operations to be wrapped into a transaction, which is then executed atomically. We used this API to implement the TPUT (transactional PUT) operation.

Besides data, znodes are associated to some specific ZooKeeper metadata (not be confused with Hybris metadata, which we store as znodes data). In particular, our implementation uses znode version number vn, that can be supplied as an additional parameter to the *setData*

[16] Initially, our implementation relied on the Apache JClouds library [25], which roughly serves the main purpose as our custom wrappers, yet covers dozens of cloud providers. However, JClouds introduces its own performance overhead that prompted us to implement the cloud driver library wrapper ourselves.

operation. In this way, *setData* becomes a *conditional update* operation, that updates a znode only if its version number exactly matches the one given as parameter.

ZooKeeper linearizable reads In ZooKeeper, only write operations are linearizable [142]. In order to get the latest updates through the *getData* calls, the recommended technique consists in performing a *sync* operation beforehand. While this normally results in a linearizable read, there exists a corner case scenario in which another quorum member takes over as leader, while the old leader, unaware of the new configuration due to a network partition, still services read operations with possibly stale data. In such case, the read data would still reflect the update order of the various clients but may fail to include recent completed updates. Hence, the "*sync+read*" schema would result in a *sequentially consistent* read [161]. This scenario would only occur in presence of network partitions (which are arguably rare on private premises), and in practice it is effectively avoided through the use of heartbeats and timeouts mechanisms between replicas [142]. Nonetheless, in principle, the correctness of a distributed algorithm should not depend on timing assumptions. Therefore we implemented an alternative, *linearizable* read operation through the use of a dummy write preceding the actual read. This dummy write, being a normal quorum-based operation, synchronizes the state among replicas and ensures that the following read operation reflects the latest updates seen by the current leader. With this approach, we trade performance for a stronger consistency semantics (i.e. linearizability [139]). We implemented this scheme as an alternative set of API calls for the ZooKeeper-based RMDS, and benchmarked it in a geo-replicated setting (see Sec. 3.5.4) — as it represents the typical scenario in which this kind of tradeoffs are most conspicuous. However, for simplicity of presentation, in the following we only refer to the *sync+read* schema for getting data from the ZooKeeper-based RMDS.

Hybris PUT At the beginning of PUT(k, v), when the client fetches the latest timestamp ts for k, the Hybris client issues a *sync*() followed by *getData*($path_k$). This *getData* call returns, besides Hybris timestamp ts, the internal version number vn of the znode $path_k$. In the final step of PUT, the client issues *setData*($path_k, md, vn$) which succeeds only if the version of znode $path_k$ is still vn. If the ZooKeeper version of $path_k$ has changed, the client retrieves the new authoritative Hybris timestamp ts_{last} and compares it to ts. If $ts_{last} > ts$, the client simply completes a PUT (which appears as immediately overwritten by a later PUT with ts_{last}). In case $ts_{last} < ts$, the client retries the last step of PUT with ZooKeeper version number vn_{last} that corresponds to ts_{last}. This scheme (inspired by [81]) is wait-free [136], thus always terminates, as only a finite number of concurrent PUT operations use a timestamp smaller than ts.

Hybris GET During GET, the Hybris client reads metadata from RMDS in a strongly consistent fashion. To this end, a client always issues a *sync*() followed by *getData*($path_k$), just like in the PUT protocol. In addition, to subscribe for metadata updates in GET we use ZooKeeper *watches*

(set by, e.g., *getData* calls). In particular, we make use of these notifications in the algorithm described in Section 3.3.5.

3.4.2 Consul-based RMDS

In order to further study Hybris performance, we implemented an alternative version of RMDS using Consul [87]. Like ZooKeeper, Consul is a distributed coordination service, which exposes a simple key-value API to store data addressed in a URL-like fashion. Consul is written in Go and implements the Raft consensus algorithm [195]. Unlike ZooKeeper, Consul offers a service discovery functionality and has been designed to support cross-data center deployments. [17]

The implementation of the Consul RMDS client is straightforward, as it closely mimics the logic described in Sec. 3.4.1 for ZooKeeper. Among the few relevant differences we note that the Consul client is stateless and uses HTTP rather than a binary protocol. Furthermore, Consul reads can be linearizable without the need for additional client operations to synchronize replicas.

3.4.3 Cross fault tolerant RMDS

The recent widespread adoption of portable connected devices has blurred the ideal security boundary between trusted and untrusted settings. Additionally, partial failures due to misconfigurations, software bugs and hardware failures in trusted premises have a record of causing major outages in productions systems [94]. Recent research by Ganesan et al. [113] has highlighted how in real-world crash fault tolerant stores even minimal data corruptions can go undetected or cause disastrous cluster-wide effects. For all these reasons, it is arguably sensible to adopt replication protocols robust enough to tolerate faults beyond crashes even in trusted premises. Byzantine fault tolerant (BFT) replication protocols are an attractive solution for dealing with these issues. However, BFT protocols are designed to handle failure modes which are unreasonable for systems running in trusted premises, as they assume active and even malicious adversaries. Besides, handling such powerful adversaries takes a high toll on performance. Hence, several recent research works have proposed fault models that stand somewhere in-between the crash and the Byzantine fault models. A prominent example of this line of research is *cross fault* tolerance (XFT) [176], which decouples faults due to network disruptions from arbitrary machine faults. Basically, this model excludes the possibility of an adversary that controls both the network and the faulty machines at the same time. Thus, it fittingly applies to systems deployed in private premises. Therefore, we implemented an instance of RMDS that guarantees cross fault tolerance. We omit implementation details because,

[17] Currently, the recommended way of deploying Consul across data centers is by using separate consensus instances through partitioning of application data (see https://www.consul.io/docs/internals/consensus.html).

as the server side code is based on ZooKeeper [176], the client side logic closely mimics the one implemented for the ZooKeeper-based RMDS.

3.4.4 Optimizations

Cloud latency ranking In our Hybris implementation, clients rank clouds by latency and prioritize those that present lower latency. Hybris client then uses these cloud latency rankings in common case to: *(i)* write to $f + 1$ clouds with the lowest latency in PUT, and *(ii)* to select from $cloudList$ the cloud with the lowest latency as *preferred* to retrieve objects in GET. Initially, we implemented the cloud latency ranking by reading once (i.e. upon initialization of the Hybris client) a default, fixed-size (e.g., 100kB) object from each of the public clouds. Interestingly, during our experiments, we observed that the cloud latency rank significantly varies with object size as well as the type of the operation (PUT vs. GET). Hence, our implementation establishes several cloud latency rankings depending on the file size and the type of operation. In addition, the Hybris client can be instructed to refresh these latency ranks when necessary.

Erasure coding Hybris integrates an optimally efficient Reed-Solomon codes implementation, using the Jerasure library [202], by means of its JNI bindings. The cloud latency ranking optimization remains in place with erasure coding. When performing a PUT, $f + k$ erasure coded blocks are stores in $f + k$ clouds with lowest latency, whereas with GET, $k > 1$ clouds with lowest latency are selected (out of $f + k$ clouds storing data chunks).

Preventing "Big File" DoS attacks A malicious preferred cloud might mount a DoS attack against an Hybris client during a read by sending, instead of the correct object, an object of arbitrary large size. In this way, a client would not detect a malicious fault until computing a hash of the received file. To cope with this attack, the Hybris client saves object size s as metadata on RMDS and cancels the downloads whose payload length exceeds s.

Caching Our Hybris implementation enables object caching on private portions of the system. We implemented simple write-through cache and caching-on-read policies. With write-through caching enabled, the Hybris client simply writes to cache in parallel to writing to the clouds. On the other hand, with caching-on-read enabled, the Hybris client asynchronously writes the GET object to cache, upon returning it to the application. In our implementation, we use Memcached distributed cache, which exports a key-value API just like public clouds. Hence, all Hybris writes to the cache use exactly the same addressing as writes to public clouds (i.e. using *put*$(k|ts, v)$). To leverage cache within a GET, the Hybris client, after fetching metadata from RMDS, always tries first to read data from the cache (i.e. by issuing *get*$(k|ts)$ to Memcached). Only in case of a cache miss, it proceeds normally with a GET, as described in Sections 3.3.3

and 3.3.5. Furthermore, Hybris can be instructed to use the caching layer to provide specific consistency semantics weaker than linearizability, as described in Sec. 3.3.10.

3.5 Evaluation

In this section we evaluate Hybris performance, costs and scalability in various settings. In detail, we present the following experiments:

1. An evaluation of common-case latency of Hybris compared to a a state-of-the-art multi-cloud storage system [53], as well as to the latency of individual cloud providers (§3.5.1).
2. An evaluation of the GET latency with one malicious fault in a public cloud (§3.5.2).
3. A scalability benchmark of the Hybris RMDS component in its crash fault and cross fault tolerant implementations (§3.5.3).
4. A benchmark of RMDS scalability in a wide area deployment (§3.5.4).
5. An evaluation of Hybris caching performance using YCSB cloud serving benchmark [91] (§3.5.5).
6. An assessment of Hybris as backend of a personal storage and synchronization application (§3.5.6).
7. An estimate of the monetary costs of Hybris compared to alternatives (§3.5.7).

In all the following experiments, unless specified otherwise, caching is disabled. We focus on the arguably most common and interesting case where $f = 1$ [92], i.e. where at most one public cloud may exhibit arbitrary faults. Furthermore, we set the erasure coding reconstruction threshold k to 2. Hybris clients interact with four cloud providers: Amazon S3, Rackspace CloudFiles, Microsoft Azure and Google Cloud Storage. For each provider, we only used cloud storage data centers located in Europe.

3.5.1 Experiment 1: common-case latency

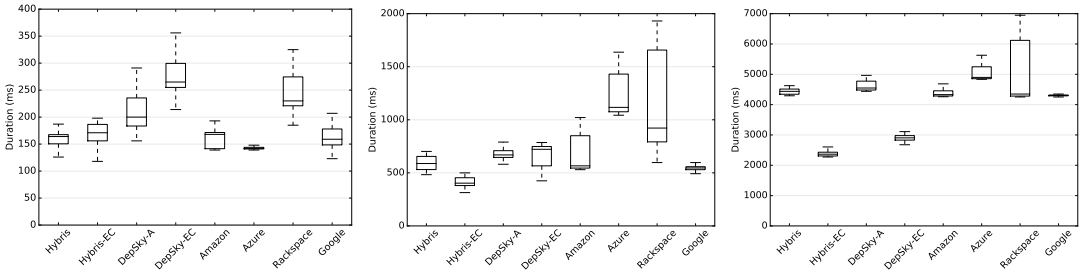
In this experiment, we benchmark the common-case latency of Hybris and Hybris-EC (i.e. Hybris using erasure coding instead of replication) with respect to those of DepSky-A, DepSky-EC (i.e. a version of DepSky featuring erasure codes support) [53], [18] and the four individual public clouds underlying both Hybris and DepSky.

Private cloud setup To perform this experiment and the next one (Sec. 3.5.2), we deployed Hybris "private" components (namely, Hybris client, metadata service (RMDS) and cache) on virtual machines (VMs) within an OpenStack [19] cluster that acts as our private cloud, located in Sophia Antipolis, France. Our OpenStack cluster consists of: two master nodes running on a dual quad-core Xeon L5320 server clocked at 1.86GHz, with 16GB of RAM, two 1TB RAID5 hard-drive volumes and two 1Gb/s network interfaces; nine worker nodes that execute on two

[18] We used the open-source DepSky implementation available at http://cloud-of-clouds.github.io/depsky/.
[19] http://www.openstack.org/

sixteen-core Intel Xeon CPU E5-2630 servers clocked at 2.4GHz, with 128GB of RAM, ten 1TB disks and four 1Gb/s network cards. [20] We use the KVM hypervisor, and each machine in the physical cluster runs the Juno release of OpenStack on top of a Ubuntu 14.04 Linux distribution. We collocate ZooKeeper and Memcached (in their off-the-shelf default configurations) using three VMs. Each VM has one quad-core virtual processor clocked at 2.40GHz, 8GB of RAM, one PATA virtual hard drive, and it is connected to the others through a gigabit Ethernet network. All VMs run the Ubuntu Linux 16.04 distribution images, updated with the most recent patches. In addition, several OpenStack VMs with similar features are used for running clients. Each VM has 100Mb/s internet connectivity for both upload and download bandwidths.

For this micro-benchmark we perform a set of independent PUT and GET operations for data sizes ranging from 100kB to 10MB. We repeated each experiment 30 times, and each set of GET and PUT operations has been performed one after the other in order to minimize side effects due to internet routing and traffic fluctuations. Figures 3.4 and 3.5 show the boxplots of client latencies, varying the size of the object to be written or read. In the boxplots, the central line shows the median, the box corresponds to the 1^{st} and 3^{rd} quartiles, and whiskers are drawn at the most extreme data points within 1.5 times the interquartile range from 1^{st} and 3^{rd} quartiles.



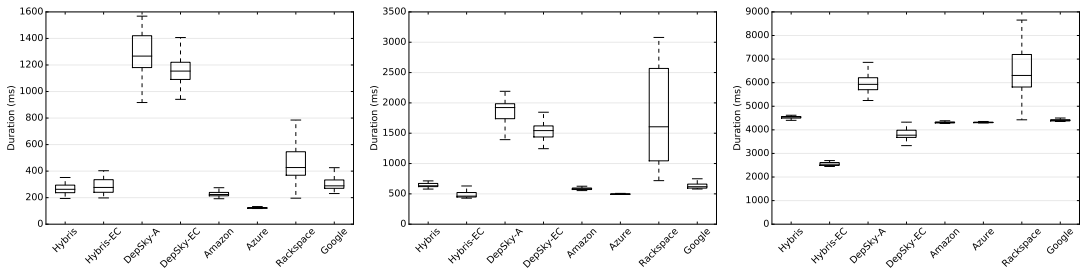
(a) 100kB GET (b) 1MB GET (c) 10MB GET

Figure 3.4 – Latencies of GET operations.

We observe that Hybris GET latency (Fig. 3.4) closely follows those of the fastest cloud storage provider, as in fact it downloads the object from that specific cloud, thanks to Hybris cloud latency ranking (see Sec. 3.4). We further observe (Fig. 3.5) that Hybris PUT roughly performs as fast as the second fastest cloud storage provider. This is expected since Hybris uploads to clouds are carried out in parallel to the first two cloud providers previously ranked by their latency.

Hybris-EC PUT uploads 3 chunks roughly half as large as the original payload, in parallel, to the three fastest clouds. Notice that the overhead of computing the coding information and of using a third cloud is amortized as the payload size increases. Similarly, Hybris-EC GET retrieves

[20] Our hardware and network configuration closely resembles the one recommended by commercial private cloud providers.



(a) 100kB PUT (b) 1MB PUT (c) 10MB PUT

Figure 3.5 – Latencies of PUT operations.

chunks of about half the original size from the two fastest clouds in parallel. As for the PUT, Hybris-EC GET performance advantage increases as the payload size increases.

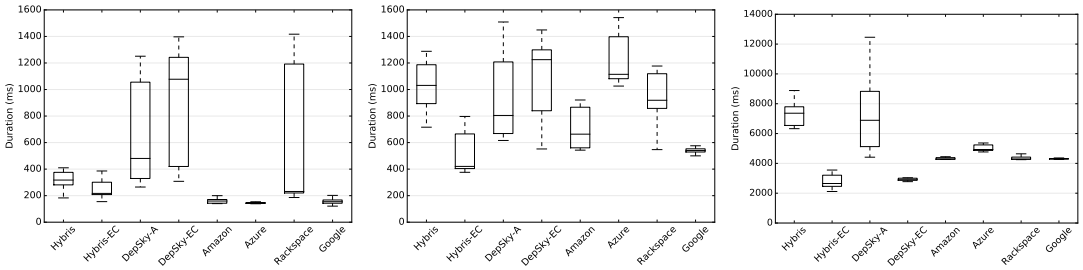
Notice that Hybris and Hybris-EC outperform the corresponding clients of DepSky in both PUT and GET operations. The difference is significant particularly for smaller to medium object sizes (e.g., 100kB and 1MB). This is explained by the fact that Hybris stores metadata locally, whereas DepSky needs to store and fetch metadata across clouds. With increased file sizes (e.g., 10MB) latency merely due to payload takes over and the difference becomes less pronounced.

We further note that we observed, throughout the tests, a significant variance of clouds performance, in particular for downloading large objects from Amazon and Rackspace. However, thanks to its latency ranking, Hybris manages to mitigate the backlashes of this phenomenon on the overall performance.

3.5.2 Experiment 2: latency under faults

In order to assess the impact of faulty clouds on Hybris GET performance, we repeated Experiment 1 with one cloud serving tampered objects. This experiment aims at stress testing the common-case optimization of Hybris to download objects from a single cloud. In particular, we focused on the worst case for Hybris, by injecting a fault on the closest cloud, i.e. the one most likely to be chosen for the download because of its low latency. We injected faults by manually tampering the data through an external client.

Figure 3.6 shows the download times of Hybris, Hybris-EC, DepSky-A and DepSky-EC for objects of different sizes, as well as those of individual clouds, for reference. Hybris performance is nearly the sum of the download times by the two fastest clouds, as the GET downloads happen, in this case, sequentially. However, despite its single cloud read optimization, Hybris performance under faults remains comparable to that of DepSky variants that download objects in parallel. We further note that both Hybris-EC and DepSky-EC are less sensitive to faulty clouds than the corresponding versions featuring plain replication, as they fetch fewer data in parallel from single clouds.



(a) 100kB GET (b) 1MB GET (c) 10MB GET

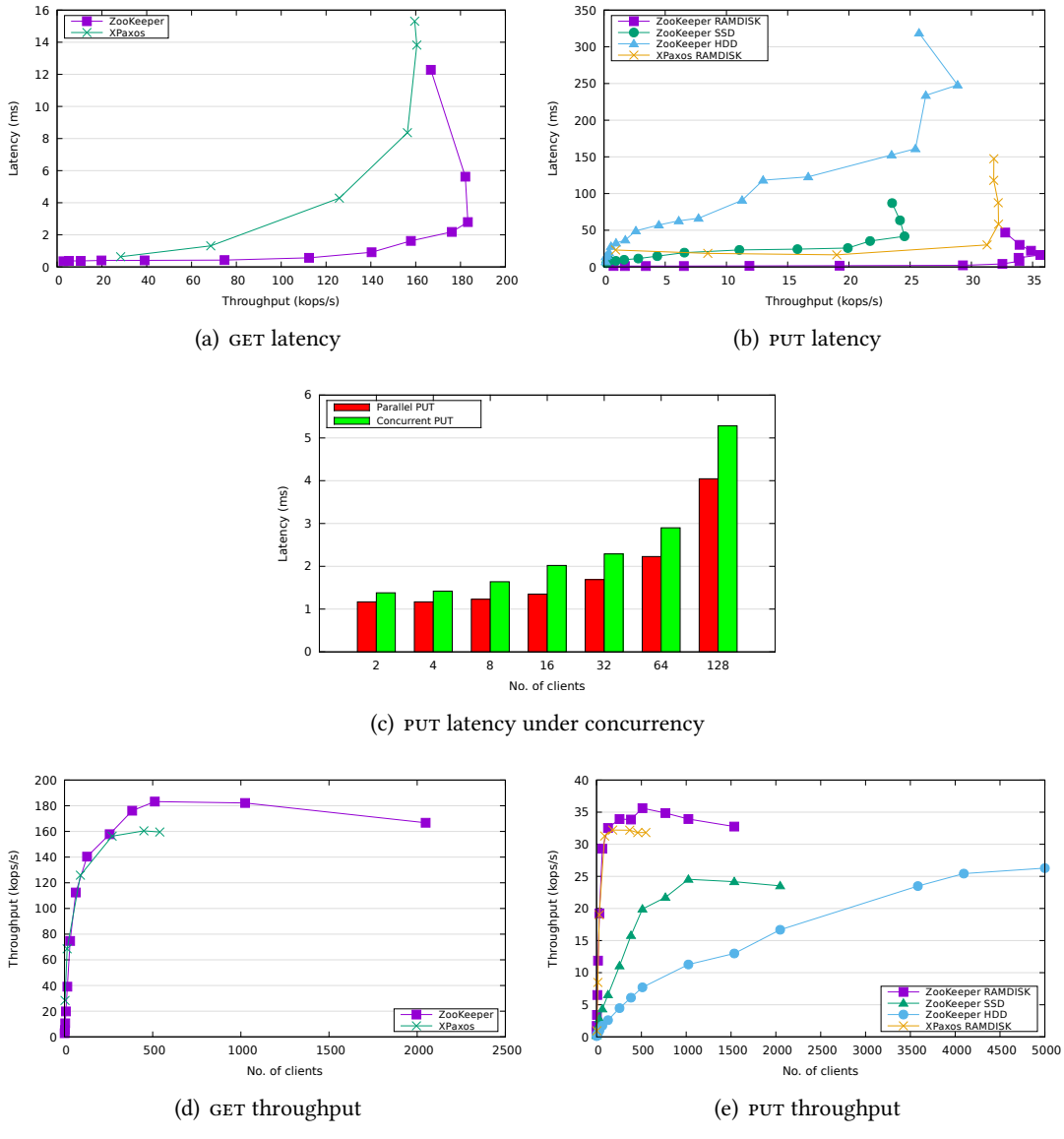
Figure 3.6 – Latencies of GET operations with one faulty cloud.

3.5.3 Experiment 3: RMDS performance

As we envision a typical deployment of Hybris in corporate settings, which generally present high Internet access bandwidth, we identify in RMDS the most likely bottleneck of the system. Therefore, in this experiment we aim to stress our crash fault tolerant (vanilla ZooKeeper) and cross fault tolerant (XPaxos [176]) RMDS implementations in order to assess their performance. For this purpose, we short-circuit public clouds and simulate uploads by writing a 100 byte payload to an in-memory hash map. To mitigate possible performance impact of the shared OpenStack private cloud, we perform (only) this experiment deploying RMDS on a dedicated cluster of three 8-core Xeon E3-1230 V2 machines (3.30GHz, 20GB ECC RAM, 1GB Ethernet, 128GB SATA SSD, 250GB SATA HDD 10000rpm). The obtained results are shown in Figure 3.7.

Figure 3.7(a) shows GET latency as we increase throughput. The observed peak throughput of roughly 180 kops/s achieved with latencies below 4 ms is due to the fact that syncing reads in ZooKeeper come with a modest overhead, and we take advantage of read locality in ZooKeeper to balance requests across nodes. Furthermore, since RMDS has a small footprint, all read requests are serviced directly from memory without incurring the cost of stable storage access. Using the XPaxos-based RMDS, Hybris GET achieves a peak of 160 kops/s with latencies of about 10 ms. For read operations, XPaxos message pattern is similar to ZooKeeper's and it uses lightweight cryptographic operations (e.g., message authentication codes).

In contrast, PUT operations incur the toll of atomic broadcast and stable storage accesses in the critical path. Figure 3.7(b) shows the latency-throughput curve for three different classes of stable storage backing ZooKeeper, namely conventional HDD, SSD and RAMDISK, which would be replaced by non-volatile RAM in a production-ready system. The observed differences suggest that the choice of stable storage for RMDS is crucial for overall system performance, with HDD-based RMDS incurring latencies nearly one order of magnitude higher than RAMDISK-based at peak throughput of 28 kops/s (resp. 35 kops/s). As expected, SSD-based RMDS is in the middle of the latency spectrum spanned by the other two storage types. XPaxos achieves a maximum throughput of about 32 kops/s using RAMDISK as storage. The difference between ZooKeeper



(a) GET latency

(b) PUT latency

(c) PUT latency under concurrency

(d) GET throughput

(e) PUT throughput

Figure 3.7 – Performance of metadata read and write operations with RMDS deployed as local cluster in private premises.

and XPaxos performance is due to the use of CPU-intensive cryptographic operations in XPaxos. Note that, unlike in [176] XPaxos does not outperform ZooKeeper because, in the cluster setting of a private cloud, CPU is the bottleneck of XPaxos, whereas in the WAN experiments by Liu et al. [176] the bottleneck, for both protocols, is the network. Nevertheless, the peak throughput of XPaxos-based RMDS is within 10% of ZooKeeper peak throughput, which seems an acceptable overhead for the additional guarantees of XPaxos and the XFT model. [21] To understand the impact of concurrency on RMDS performance, we evaluated the latency of PUT under heavy contention to a single key. Figure 3.7(c) shows that despite 128 clients writing concurrently to the same key, the latency overhead is only 30% over clients writing to separate keys.

Finally, Figures 3.7(d) and 3.7(e) depict throughput curves as more clients invoking operations in closed-loop are added to the system. Specifically, Fig. 3.7(d) suggests that ZooKeeper-based RMDS is able to service read requests coming from $2K$ clients near peak throughput, while XPaxos can service up to 600 clients on the same client machines due to its substantial use of cryptographic operations. On the other hand, Figure 3.7(e) shows again the performance discrepancy in ZooKeeper when using different stable storage types, with RAMDISK and HDD at opposite ends of the spectrum. Observe that HDD peak throughput, despite being below that of RAMDISK, slightly overtakes SSD throughput with $5K$ clients.

3.5.4 Experiment 4: RMDS geo-replication

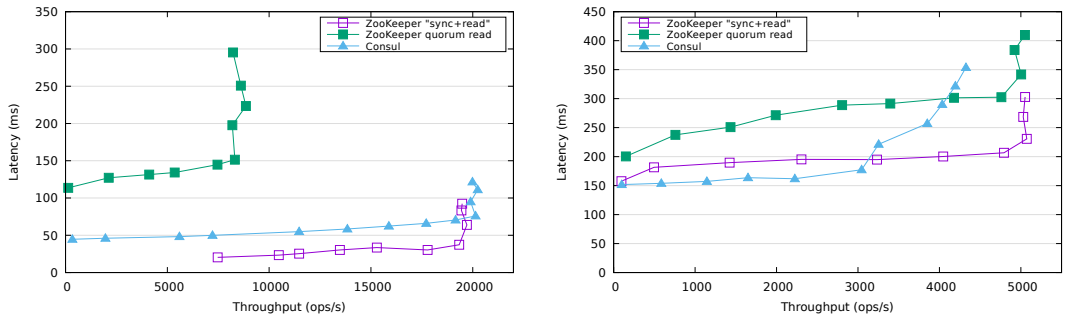
Modern applications are being deployed increasingly often across wide area networks, in so-called *geo-replicated* settings [92], to improve latency and/or fault tolerance. To evaluate Hybris performance in this context, we placed each of the servers composing the RMDS cluster in a different data center, and replicated the measurements of Sec. 3.5.3.

For this experiment, we used virtual machines and network infrastracture by IBM SoftLayer [220]. Specifically, three virtual machines make up the RMDS cluster, while three others host processes emulating concurrent clients invoking GET and PUT operations. We placed two machines (one for the clients and one as a server) in each of three data centers located in San Jose (US), Washington D.C. (US) and London (UK). All the machines run Ubuntu 16.04 and dispose of 4GB RAM, one quad-core 2.6GHz CPU, a 24GB SSD virtual hard-drive, and 100Mbps public VLAN. Figure 3.8 illustrates the latencies between data centers.

Each client machine ran up to 800 processes sequentially reading or writing 100kB objects to an in-memory hash map, as in Sec. 3.5.3. We remark that in Hybris both RMDS and clouds can be configured separately on a client basis, thus making it possible to exploit the most favorable settings in terms of deployment location. However, finding the best combination of client settings for wide area deployment requires more specific assumptions that depend on the application domain and is therefore out of the scope of this experiment. Research literature

[21] An optimized implementation of XPaxos could achieve better performance by offloading cryptographic operations to a dedicated cryptoprocessor.

[55, 130] and deployment guidelines [84, 88] suggest mitigating the performance cost of strongly consistent wide area coordination by means of read-only servers and state partitioning. We acknowledge these approaches as beneficial and practical, despite lacking generality and a genuine cross-data center consensus primitive. However, in this experiment we aim at assessing how far we can stretch the performance of a single consensus instance based on off-the-shelf coordination systems available as of September 2016, i.e. ZooKeeper 3.4.9 and Consul 0.7. In particular, in addition to the standard "*sync+read*" sequentially consistent ZooKeeper read, as mentioned in Sec. 3.4.1, we implemented a linearizable metadata read operation by prepending a quorum-based dummy write. We benchmark both the quorum-based and the *sync+read* ZooKeeper schemes along with the Consul RMDS. Figure 3.9 shows the results of this wide area benchmark with respect to throughput and latency performance.



(a) GET performance

(b) PUT performance

Figure 3.9 – Performance of metadata operations for Hybris PUT and GET in wide area settings using Consul or ZooKeeper as RMDS. Each RMDS cluster is composed of three servers deployed in San Jose, Washington D.C. and London.

During the write experiment, multiple Hybris clients performed PUT operations to different keys, while in the read experiments all clients read data associated to a single key. Both coordination systems reach peak throughput when using about 1600 concurrent clients. Note how this simple wide area deployment strategy easily reduces read throughput to $1/9$ and write throughput to $1/6$ of the corresponding figures for local clusters. Nonetheless, the throughputs and latencies recorded are arguably acceptable for a wide range of applications, especially in contexts of low write concurrency or asynchronous storage without direct user interaction.

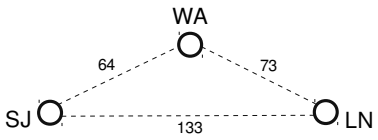


Figure 3.8 – Round trip time latency (ms) between the data centers of Fig. 3.9.

The performance difference between the two coordination systems derive from the consensus protocol they implement (i.e. Zab [206] vs Raft [195]) and the specific API they expose.

Besides, as expected, we recorded a substantial performance loss when using the quorum-based ZooKeeper reads, as they require a wider agreement among replicas. We further note that the low average latencies of the ZooKeeper *read+sync* instance are due to the presence of reads performed by clients located in the same data center of the cluster leader. Ultimately, the choice of the coordination system depends on various needs and in practice it often hinges on infrastructure already in place. Hybris accommodates these needs through its modular design that eases the implementations of different RMDS instances.

3.5.5 Experiment 5: caching

In this experiment, we measure the performance of Hybris with and without caching (both write-through and caching-on-read simultaneously enabled). We deploy Memcached with 128MB cache limit and 10MB single object limit. We vary object sizes from 1kB to 10 MB and measure average latency using the YCSB benchmarking suite with workload B (95% reads, 5% writes). The results for GET are presented in Figure 3.10.

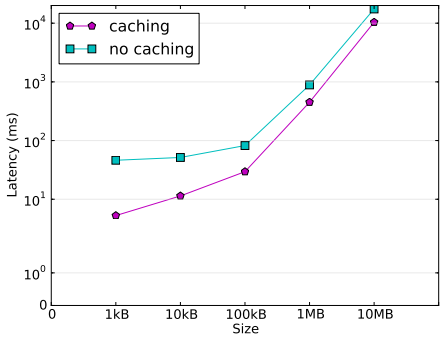


Figure 3.10 – Hybris GET latency with YCSB workload B (95% reads, 5% writes) varying data size.

Observe that caching decreases latency by an order of magnitude when the cache is large enough compared to object size. As expected, the benefits of cache decrease with increase in object size, and the resulting cache misses. This experiment shows that Hybris can simply benefit from caching, unlike other multi-cloud storage protocols (see also Table 3.2).

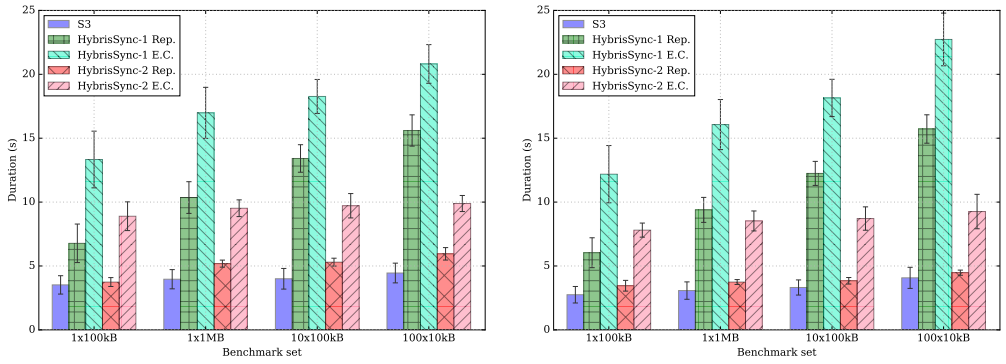
3.5.6 Experiment 6: Hybris as personal storage backend

A thorough evaluation of a storage protocol also depends on the type of application that makes use of it. For this reason, we decided to quantify the practical benefits and overhead of using Hybris as backend of a personal storage and synchronization application. Over the last decade, this kind of application has gained a significant adoption both in household and corporate contexts. Several products have been developed and commercialized, usually as

freeware in conjunction with storage pay-per-use schemes. At the same time, researchers have started studying their performance [99, 100].

We decided to integrate Hybris as storage backend of Syncany, a popular open source storage synchronization application written in Java [225]. The integration entailed the development of a storage plugin in two different versions. [22] The first version, which we call *HybrisSync-1*, uses Hybris for storage of Syncany data and metadata indifferently, while the second version (*HybrisSync-2*) exposes an API to exploit Hybris RMDS also for Syncany's own metadata management. In addition, we instrumented the Syncany command line client to measure the upload and download latencies of synchronization operations. We chose to compare Hybris — in its two versions, and using replication or erasure coding — with the baseline performance of an existing storage plugin integrating Amazon S3 as remote repository. [23]

For this experiment, we hosted the RMDS on a cluster of three virtual machines as in Sec. 3.5.1. We employed two other similar virtual machines to simulate two clients on the same local network, mutually synchronizing the content of local folders using Syncany. During the experiment, we employed only cloud storage accounts referring to data centers located in Europe, as the client machines. Considering the statistics about workloads in personal cloud storage [99], we designed a set of benchmarks varying the number of files to be synchronized along with their sizes.



(a) Syncany upload

(b) Syncany download

Figure 3.11 – Performance of data synchronization between hosts using different Syncany storage plugins: Amazon S3 ("S3"), and different Hybris versions. "HybrisSync-1" uses Hybris cloud storage for both Syncany data and metadata, while "HybrisSync-2" keeps Syncany's metadata in the RMDS. Both Hybris versions are evaluated in their replicated and erasure coded versions. Each bar represents the average of 30 repetitions, with whiskers marking the standard deviation interval.

[22] https://github.com/pviotti/syncany-plugin-hybris.
[23] https://github.com/syncany/syncany-plugin-s3

From the results shown in Fig. 3.11 we draw the following considerations. First, as highlighted in Sec. 3.5.1, erasure coding is beneficial only for object sizes that exceed a certain threshold (e.g., about 1MB in this experimental setting). For smaller objects the computational overhead and the additional latency introduced by the use of a third cloud outplays the reduced payload size. Therefore, given the kind of workload involved, cross-remote storage erasure coding is not a good match for personal cloud storage. Second, the version of the Hybris plugin exposing an API for metadata management performs significantly better than the one handling in the same way both Syncany data and metadata. This is due to the high latency cost of using clouds even for lightweight operations on metadata, which, in *HybrisSync-1* are in fact stored on clouds. Finally, *HybrisSync-2* with replication perform similarly to the Amazon S3 plugin, while offering further substantial guarantees in terms of consistency and fault tolerance.

In addition to this experiment, and in the scope of the CloudSpaces project [85], we integrated Hybris with StackSync [180], a prototype that provides, like Syncany, personal storage synchronization. [24] While exploratory, this integration demonstrates the feasibility of adopting Hybris also as backend of more scalable personal storage solutions.

3.5.7 Cost comparison

Table 3.1 shows an estimate of the monetary costs incurred by several cloud storage systems in the common case (i.e. in case of synchrony and without failures), including Amazon S3 as baseline. We set $f = 1$ and assume a symmetric workload that involves 10^6 write and 10^6 read operations accessing 1MB objects totaling to 1TB of storage over a period of 1 month. This corresponds to a modest workload of roughly 40 hourly operations. We further assume a metadata payload of 500B for each object. The reference costs per transaction, storage, and outbound traffic are those of the Amazon S3 US-East region, [25] as of September 5th, 2016. The cost comparison is based on the protocols' features reported in Table 3.2, and takes into account all applicable read and write cost optimizations (e.g., preferred quorums and sequential reads). Our figures exclude the cost of the private cloud in Hybris, which we assume to be part of an already existing infrastructure.

We observe that the overhead of Hybris is twice the baseline both for PUT and storage because Hybris stores data in two clouds in the common case. Since Hybris uses a single cloud once for each GET operation, the cost of GET equals that of the baseline, and hence is optimal. On the other hand, Hybris-EC incurs for $k = 2$ a moderate storage overhead of 1.5 times the baseline at the cost of increased overhead for PUT, as data needs to be dispersed onto three clouds. We further note that Hybris is the most cost-effective of the multi-cloud systems considered, as it requires, in its erasure coding version, an additional expense of only 19% more than the cost of a single cloud.

[24] https://github.com/pviotti/stacksync-desktop
[25] AWS Simple Monthly Calculator: https://calculator.s3.amazonaws.com/index.html.

Table 3.1 – Cost of cloud storage systems in USD for 2 x 10^6 transactions involving 10^6 objects of 1MB, totaling 1TB of storage.

System	PUT	GET	Storage Cost / Month	Total
ICStore [42]	60	276	180	516
DepSky-A [53]	30	93	91	214
DepSky-EC [53]	30	93	45	168
Hybris	10	92	60	162
Hybris-EC	15	92	45	152
Amazon S3	5	92	30	127

3.6 Related Work

Table 3.2 – Comparison of existing robust multi-writer cloud storage protocols. We distinguish cloud data operations (D) from cloud metadata operations (m). Unless indicated differently, properties pertain to replication-based variants.

Protocol	Semantics		Common case performance	
	Cloud faults	Consistency	No. of cloud operations	Blowup
ICStore [42]	crash-only	linearizable[a]	$(4f+2)(D+m)$ (writes) $(2f+1)(D+m)$ (reads)	$4f+2$
DepSky [53]	**arbitrary**	regular[a]	$(2f+1)(D+m)$ (writes) $(2f+1)(D+m)$ (reads)	$2f+1$[b]
Hybris	**arbitrary**	**linearizable**	$(\mathbf{f+1})\mathbf{D}$ (writes) $\mathbf{1D}$ (reads)	$\mathbf{f+1}$[c]

[a]Unlike Hybris, to achieve linearizable (resp., regular) semantics, ICStore (resp., DepSky) requires public clouds to be linearizable (resp., regular).

[b]The erasure coded variant of DepSky features $\frac{2f+1}{f+1}$ storage blowup.

[c]The erasure coded variant of Hybris features $\frac{f+k}{k}$ storage blowup, for any $k > 1$.

Multi-cloud storage systems Several storage systems have been designed to use multiple clouds to boost data robustness, notably in its reliability and availability. SafeStore [153] erasure-codes data across multiple storage platforms (clouds) and guarantees data integrity, confidentiality and auditing. It uses a non-replicated local server as encryption proxy, and to cache data and metadata, both stored on clouds. Furthermore, SafeStore requires from cloud providers to disclose information about their internal redundancy schemes, and to expose an API that is not available in any of nowadays' cloud storage services. SPANStore [245] seeks to minimize the cost of use of multi-cloud storage, leveraging a centralized cloud placement manager. However, SafeStore and SPANStore are not robust in the Hybris sense, as their centralized components (local proxy and placement manager, respectively) are single points of failure. RACS [4] and HAIL [58] assume immutable data, hence they do not address any concurrency aspects. The Depot key-value store [186] tolerates any number of untrusted clouds, but does not offer strong consistency and requires computational resources on clouds.

The multi-cloud storage systems most similar to Hybris are DepSky [53] and ICStore [42]. For clarity, we summarize the main aspects of these systems in Table 3.2. ICStore models cloud faults as outages and implements robust access to shared data. Hybris advantages over ICStore include tolerating malicious clouds and smaller storage blowup. [26] On the other hand, DepSky considers malicious clouds, yet requires $3f+1$ replicas, unlike Hybris. Furthermore, DepSky consistency guarantees are weaker than those of Hybris, even when clouds are strongly

[26]The blowup of a given redundancy scheme is defined as the ratio between the total storage size needed to store redundant copies of a file, over the original file size.

consistent. Finally, Hybris guarantees linearizability even in presence of eventually consistent clouds, which may harm the consistency guarantees of both ICStore and DepSky. Recently, and concurrently with this work, SCFS [52] augmented DepSky to a full-fledged file system by applying a similar idea of turning eventual consistency to strong consistency by separating cloud file system metadata from payload data. Nevertheless, SCFS still requires $3f + 1$ clouds to tolerate f malicious ones, i.e. the overhead it inherits from DepSky.

Latency-consistency tradeoffs for cloud storage Numerous recent works have proposed storage systems that leverage cloud resources to implement tunable latency-consistency tradeoffs. In particular, some of these works focus on providing tunable consistency semantics expressed by static declarative contracts (e.g., [230, 219]) while others offer dynamic adaptive mechanisms (e.g., [253, 79]). In alternative to strong consistency, Hybris provides tunable consistency semantics as well, through a static configuration of caching mechanisms implemented in trusted, private premises. Unlike previous works proposing latency-consistency tradeoffs, Hybris explicitly addresses resiliency concerns, and does not entail modification to standard cloud storage interfaces nor it requires cloud computing resources: its RMDS component has a small footprint which can be conveniently supplied by on-premises resources often already in place in corporate settings.

Separating data from metadata The idea of separating metadata management from data storage and retrieval has been proposed in previous literature. Notably, it has been adopted in the design of parallel file systems, with the main goal of maximizing throughput [119, 242]. Farsite [9] is an early protocol similarly proposing this design choice: it tolerates malicious faults by replicating metadata (e.g., cryptographic hashes and directory) separately from data. Hybris builds upon these techniques yet, unlike Farsite, it implements multi-writer/multi-reader semantics and is robust against timing failures as it relies on lock-free concurrency control. Furthermore, unlike Farsite, Hybris supports ephemeral clients and has no server code, targeting commodity cloud storage APIs.

Separation of data from metadata is intensively used in crash-tolerant protocols. For example, in the Hadoop Distributed File System (HDFS), modeled after the Google File System [117], HDFS NameNode is responsible for maintaining metadata, while data is stored on HDFS DataNodes. Other notable crash-tolerant storage systems that separate metadata from data include LDR [108] and BookKeeper [148]. LDR [108] implements asynchronous multi-writer multi-reader read/write storage and, like Hybris, uses pointers to data storage nodes within its metadata and requires $2f + 1$ data storage nodes. However, unlike Hybris, LDR considers full-fledged servers as data storage nodes and tolerates only their crash faults. BookKeeper [148] implements reliable single-writer multi-reader shared storage for logs. It stores metadata on servers (bookies) and data (i.e. log entries) in log files (ledgers). Like in Hybris RMDS, bookies point to ledgers, facilitating writes to $f + 1$ replicas and reads from a single ledger in the common-case. Unlike

BookKeeper, Hybris supports multiple writers and tolerates malicious faults of data repositories. Interestingly, all robust crash-tolerant protocols that separate metadata from data (e.g., [108, 148], but also Gnothi [240]), need $2f + 1$ data repositories in the worst case, just like Hybris, which additionally tolerates arbitrary faults.

After the publication of the preliminary, conference version of this work, several follow-up storage protocols that separate metadata from data and tolerate arbitrary faults have appeared. Notably, MDStore [74] and AWE [24] follow the footsteps of Hybris and use optimal number of metadata and data nodes, and implement read/write storage using replication (MDStore) and erasure coding (AWE). Unlike Hybris, MDStore and AWE are fully asynchronous and replace the eventually synchronous state-machine replication based metadata service used in Hybris with asynchronous read-write metadata service. This, however, results in increased complexity of MDStore and AWE protocols over Hybris, notably manifested in the higher latency values. Furthermore, MDStore and AWE implementations are not available, unlike that of Hybris.

More recently, Zhang et al. [256] described the design of Cocytus, an in-memory data store that applies erasure coding to bulk data while replicating metadata and keys through a primary-backup scheme. While in Hybris we exploit data and metadata separation for fault tolerance and correctness, Cocytus adopts this hybrid scheme to enable fast data recovery. In fact, while in Cocytus data and metadata are only logically separated, Hybris store them on separate systems offering different guarantees in matter of reliability and consistency. We further note that, like Cocytus, Hybris can optionally apply erasure coding to bulk data stored on clouds.

Finally, the idea of separating control and data planes in systems tolerating arbitrary faults was used also by Yin et al. [248] in the context of replicated state machines (RSM). While such approach could obviously be used for implementing storage as well, Hybris proposes a far more scalable and practical solution, while also tolerating pure asynchrony across data communication links.

Systems based on trusted components Several systems in research literature use trusted hardware to reduce the overhead of replication despite malicious faults from $3f + 1$ to $2f + 1$ replicas, typically in the context of RSM (e.g., [93, 82, 149, 235]). Some of these systems, like CheapBFT [149], employ only $f + 1$ replicas in the common case.

Conceptually, Hybris is similar to these systems in that it uses $2f + 1$ trusted metadata replicas (needed for RMDS) and $f + 1$ (untrusted) clouds. However, compared to these systems, Hybris is novel in several ways. Most importantly, existing systems entail placing trusted hardware within an untrusted process, which raises concerns over practicality of such approach. In contrast, Hybris trusted hardware (private cloud) exists separately from untrusted processes (public clouds), with this hybrid cloud model being in fact inspired by practical system deployments.

3.7 Summary

In this chapter, we presented Hybris, a robust hybrid cloud storage system. Hybris scatters data (using replication or erasure coding) across multiple untrusted and possibly inconsistent public clouds, and it replicates metadata within trusted premises. Hybris is very efficient: in the common-case, using data replication, writes involve only $f + 1$ clouds to tolerate up to f arbitrary public cloud faults, whereas reads access a single cloud. Hence, Hybris is the first multi-cloud storage protocol that makes it possible to tolerate potentially malicious clouds at the price of coping with simple cloud outages. Furthermore, Hybris offers strong consistency, as it leverages strongly consistent metadata stored off-clouds to mask the inconsistencies of cloud stores. Hybris is designed to seamlessly replace commodity key-value cloud stores (e.g., Amazon S3) in existing applications, and it can be used for storage of both archival and mutable data, due to its strong multi-writer consistency.

We also presented an extensive evaluation of the Hybris protocol. All experiments we conducted show that our system is practical, and demonstrate that it significantly outperforms comparable multi-cloud storage systems. Its performance approaches that of individual clouds.

Chapter 4

Automated Declarative Consistency Verification

In this chapter, we describe a novel approach to tackle the general issue of verifying the correctness of real world database systems. The key idea of this approach is defining consistency semantics as declarative invariants of executions. Specifically, we adopt the formal definitions of consistency semantics that we provided in Chapter 2 to verify what guarantees are actually respected by distributed stores, despite non-deterministic concurrency and partial failures. We developed a preliminary implementation that allows us to assess benefits and limitations of this approach.

4.1 Introduction

Consistency is a key correctness criterion of distributed storage systems. It plays a central role in the design and development of real world data streaming and data storage systems. The importance of consistency becomes even more evident when dealing with composite distributed systems, which entail composing different assumptions and correctness semantics. Nonetheless, in spite of recent efforts proposing *consistency-by-construction* through formal methods [243, 168], most real world storage systems are still developed in an ad-hoc manner: practitioners start with an implementation and proceed with correctness verification through limited testing afterwards (e.g., using unit and integration tests). As a result, those implementations are fraught with bugs that may prevent them from respecting their intended semantics or even lead to data loss [151, 32].

In an attempt to bridge the gap between traditional testing and formal techniques, several approaches to consistency verification have been devised: we summarize them in the following paragraphs.

Strong consistency checkers Several works focused on devising efficient techniques to determine whether executions of storage systems are strongly consistent [191, 244, 118, 140, 181]. This binary decision problem has then been extended to support other comparably strong semantics [23], and on-the-fly, incremental verification [122, 107].

Read-write staleness benchmarking A recent approach proposes client-side staleness measurements as a method to assess consistency. Specifically, data store clients perform write and read operations in a coordinated fashion in order to detect anomalies related to data staleness. This technique has been applied both to open source NoSQL databases [198] and, in a black-box testing manner, to commercial geo-replicated cloud stores [169, 47, 239], typically in the context of highly available, eventually consistent systems. Other works have focused on modeling and simulation of eventually consistent stores based on quorum systems by means of Monte Carlo simulation or by computing probability convolutions [35, 45].

Precedence graph A number of works related to transactional systems adopted a graph-based approach [8, 254]. According to this approach, transactions are represented by vertices of a precedence (or serialization) graph, while the edges connecting them denote their mutual read/write dependencies. Identifying inconsistencies (e.g., isolation anomalies) amounts to finding cycles in the precedence graph.

Application-level invariant checkers Finally, in recent years, some research efforts have focused on devising proof rules and efficient techniques to establish whether application-level invariants — rather than storage read/write semantics — are respected by the underlying storage system [219, 125, 194].

All these approaches, however, lack generality, as they target only a limited subset of consistency models. They also lack a comprehensive, structured view of the entire consistency spectrum, which makes them suboptimal in verifying the core semantics, or composition thereof, of different consistency models.

We believe that the first step towards building an effective and comprehensive consistency testing framework should be the adoption of a theoretically sound model of consistency. To this end, we advocate the use of a *declarative* approach to define a set of core semantics applicable to all consistency models. In particular, we aim at expressing both client-side visibility of read/write operations and server-side replicas state. By using logic predicates that encompass these two perspectives, we define consistency semantics that capture, in form of graph entities, the salient aspects of system executions, i.e. *ordering* and *visibility* of events. In this way, verifying an implementation of a given consistency semantics amounts to finding, for a given execution, the global state configurations that validate a logic predicate, taking into account client-side events.

In summary, we introduce a comprehensive and principled approach to verify the read-/write level correctness of non-transactional databases, by adopting the formal definitions of consistency semantics that we introduced in Chapter 2. Effectively, we propose a declarative, *property-based* approach to consistency verification, in the same vein of previous proposals made in the context of generic software testing [83]. We experiment this approach by implementing Conver, an early prototype of a property-based consistency verification framework that we developed in Scala.

4.2 A declarative semantic model

Several works in the literature [179, 15] illustrate the benefits of a declarative approach in the context of database, networking and distributed programming. Essentially, the declarative, axiomatic approach offers a better match to application-level semantics than the traditional imperative, operational approach. Therefore, the declarative approach allows for a more expressive, clear and compact way to describe the logic of distributed applications. Additionally, the declarative approach is amenable to static checking of correctness conditions, allowing distributed systems problems to be naturally cast into SAT/SMT problems [17, 219, 125], which in turn allows to leverage the efficiency of related state-of-the-art tools (e.g., [251, 1]).

We found a model supporting the declarative paradigm for consistency semantics in the work by Burckhardt [65], which we extended and refined as described in Chapter 2. This model supports expressing declarative, composable consistency semantics as first-order logic predicates over graph entities which describe visibility and ordering of operations.

In order to verify the consistency models listed in Chapter 2, we would need to build the entities composing abstract executions. In particular, we would need to work out the arbitration order of operations (ar) established by the storage system being verified. This requirement presents us with a design choice: should we build a verification framework that requires to instrument the code of each system being verified, or should we adopt a black-box approach? We decided to adopt a black-box approach, as it makes the framework easier to use and more practical to customize to different storage systems. However, the black-box testing approach makes it difficult to work out the arbitration order, as it can only make use of client processes. A seemingly viable approach could be building all possible total orders ar as a *linear extensions* of the returns-before partial order rb, and find among those total orders any one matching operations outcomes. Unfortunately, in spite of previous works describing similar efforts in the context of main memory models for concurrency [247, 231], this proved to be computationally intensive, and ultimately impractical, due to modern SAT/SMT tools being generally ill-suited to solve ordering constraints problems. [1]

[1] See, as an illustration: http://stackoverflow.com/questions/35558556/linear-extension-of-partial-orders-with-z3.

Considering this, we dismissed the global arbitration order to verify consistency semantics and focused only on information that could be collected at the client side. As a result, the spectrum of semantics verifiable by our current implementation is reduced to those listed in Figure 4.1. As can be noticed, we joined some semantics into *macro-semantics*. In particular, inter-session monotonicity embeds the notion of monotonic reads and monotonic writes, as it is in fact not possible to distinguish them from clients' perspective. Moreover, to check executions for linearizability, we implemented a graph-based algorithm which is based on the work by Lu et al. [181]. We postpone further details and examples of the checks we implemented to the next section.

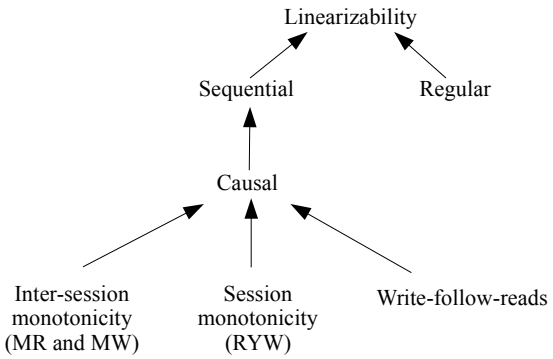


Figure 4.1 – Consistency semantics verifiable by our preliminary Conver implementation.

4.3 Property-based consistency verification

In this section we present Conver, a tool to verify the consistency semantics implemented by storage systems in a black-box manner. Conver verifies semantics as invariants of storage systems executions, thus effectively implementing a *property-based testing* approach.

Property-based testing (PBT) [83] (also called generative testing) is an approach to generic software testing that alleviates the burden of test case generation from the user, allowing the user to focus on specifying application-level properties that should hold for all executions. A PBT tool, when supplied with these properties along with information about the generic format of a valid input, generates random inputs, and then applies these inputs to the program while constantly verifying the validity of the supplied properties throughout the execution. In a sense, PBT combines the two old ideas of specification-based testing [166] and random testing [132]. Additionally, to make up for the possible "noise" induced by the random test case generation, modern PBT tools automatically reduce the complexity of failing tests to a minimal test case [255], thus proving useful in debugging tasks.

As an example, given a function `lsort` that sorts a list of integers, writing a property that states that the function should not change the list length, would just require the following lines code in Erlang:

```
prop_same_length() ->
    ?FORALL(L,list(integer()), length(L)=:=length(lsort(L))).
```

In principle, the PBT approach of expressing and testing consistency as a set of predicates allows for a testing methodology focused on correctness properties rather than operational semantics.

We embed this idea in the design of Conver, a prototype of a consistency verification framework that we developed in Scala. [2] Conver generates test cases consisting of executions of concurrent operations invoked on the data store under test. After each execution, Conver collects all client-side information and builds a graph describing operations' outcomes and relations (e.g., the returns-before relation rb, the session-order relation so, operation timings and results, etc.). Given client-side outcomes, Conver builds graph entities about ordering and visibility of operations. Then, Conver verifies the compliance of the execution to a given consistency model by checking the graph entities against the logic predicates composing the consistency model. First, it checks whether the execution respects linearizability (§ 2.3.1) by running a slightly modified version of the algorithm reported in [181]. If the linearizability check fails and a total order of operations cannot be determined, Conver runs a set of checks on the anomalies found. Specifically, by means of the graph entities described in Chapter 2, it looks for violation of write ordering across and within sessions. Table 4.1 lists the kind of anomalies Conver can detect, along with illustrations of minimal sample executions as drawn by Conver. Indeed, as a result of the verification process, Conver not only outputs a textual report of the execution, but it also provides a visualization of each failing test case, i.e. all executions that did not comply with a given consistency semantics. Additionally, the visualization highlights the operations that caused the test failure.

By default, Conver tests are run against clusters deployed on the local machine using Docker [3] containers; this greatly improves their portability, and eases their integration within existing test suites. As an example, test executions of ZooKeeper [4] verified that, as expected, it provides sequential consistency or linearizability, depending on the read API used (see § 3.4.1). Similarly, Riak's [5] consistency ranges from regular to session guarantees depending on its replication settings. Furthermore, Conver can programmatically emulate WAN latencies between containers and inject network faults by using the `netem` [6] Linux kernel module. Thanks to this feature, Conver can exercise the intrinsic nondeterminism of distributed systems further, and potentially discover subtle bugs [114]. Besides, Conver can be easily extended to

[2] Conver's source code is available at https://github.com/pviotti/conver.
[3] https://www.docker.com/
[4] https://zookeeper.apache.org/
[5] http://basho.com/products/riak-kv/
[6] https://wiki.linuxfoundation.org/networking/netem

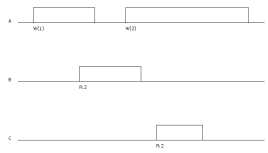
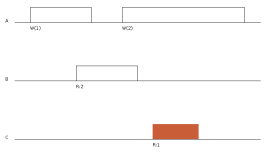
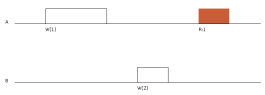
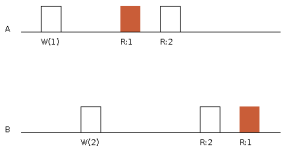
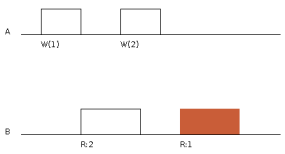
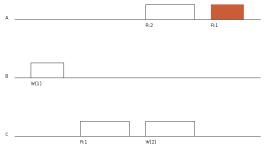
Consistency semantics	Anomaly	Sample execution
Linearizability	-	
Regular	New-old inversion	
Sequential	Stale read	
Causal	No global total order	
-	Inter-session monotonicity (Monotonic reads/writes)	
-	Session monotonicity (Read-your-writes)	
-	Inter-session causality (Writes-follow-reads)	

Table 4.1 – Consistency semantics with corresponding anomalies and minimal sample executions. All visualizations have been produced running Conver's own unit tests suite.

support the generation of test cases targeted to specific consistency semantics or corresponding to predefined workloads. In this way, Conver would lend itself to explore the consistency *vs.* latency tradeoffs, as in the vein of recent work by Fan et al. [107]. Figure 4.2 provides an overview of the functional architecture of Conver, including these extensions.

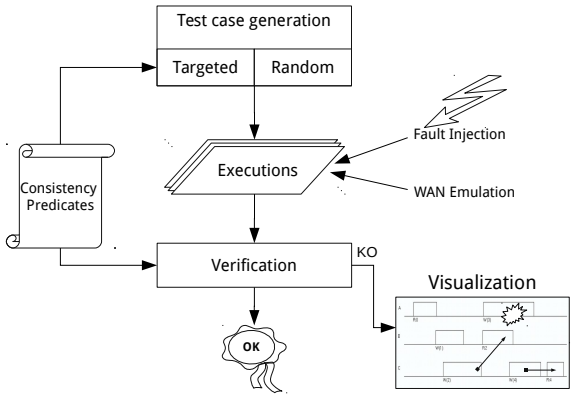


Figure 4.2 – Functional architecture of the Conver verification framework.

4.4 Discussion

With Conver, we made clear and specific design decisions, which we discuss in this section.

As previously mentioned, the black-box verification approach precludes the assessment of semantics that require server-side knowledge. Moreover, Conver trades the confidence on correctness typical of formal methods for better usability and performance, as it does not require access to source code of the data store under test, nor it needs support in form of code annotations, onerous proof tools or deep packet inspection. Hence, it can easily be used in association with existing testing tools. Moreover, since we designed Conver as a modular framework [91] adding the support for a new data store entails implementing a simple read/write API, which usually amounts to writing less than 50 lines of Scala code.

Another potential limitation of this approach concerns the scalability of the algorithms building and checking graph entities representing executions. In this regard, we remark that the most onerous algorithm we implemented in Conver (i.e. the linearizability checker) has been devised and used at Facebook on massive datasets [181]. Furthermore, in light of recent related research [250], we argue that simple setups of few hosts in combination with network fault injection can be sufficient to uncover most correctness bugs. Thus, the graph structures Conver needs to work out would be fairly limited in size. In our experience, an execution with about 10 clients and a cluster of 3 hosts takes less than 15 seconds, including the time required to set up the Docker cluster.

We further note that the property-based testing approach has already been applied by practitioners to verify the correctness of distributed applications. Specifically, modern PBT tools model the state of the system as a set of variables that are verified through postconditions [2]. This state model can only support the verification of consistency models that presume single-copy semantics, i.e. strong consistency models. Conver differs substantially from common PBT tools in the way the system state is represented and verified. In particular, the semantic model implemented by Conver describes the system state as a graph of operations [65]. Hence, Conver can verify a broader set of consistency models that apply to generic replicated storage systems.

4.5 Future Work

In the following, we discuss several possible extensions of our work.

Transactional consistency models Conver can be extended to support the verification of transactional consistency models. The state model supported by Conver has already been adapted to express transactional semantics [77]. Hence, this extension would entail the support of additional semantic entities for expressing transactional features within the model, and the implementation of an algorithm to detect transactional anomalies [8].

Mapping to application-level invariants A current trend in research advocates the use of application-level invariants to enable a fine-grain (but less portable) approach to consistency enforcement and verification [34, 125]. In this regard, we think it would be interesting to study how those invariants map to low-level I/O semantics within Conver. In order to do so, an additional layer in the Conver architecture could emulate common application use cases and interaction patterns, and match them with read-write database semantics.

Consistency-latency measurement Conver can be extended to measure latency of operations and throughput. In this way, it would serve as tool to explore consistency-performance tradeoffs in databases, both offline and in an incremental manner, as shown in recent work by Fan et al. [107].

Simulation and model checking The execution traces generated by Conver can be input into simulation frameworks. Moreover, Conver executions' data may help build mathematical models of the data store under test that could allow the development of formal specifications [141] that in turn would enable model checking or formal proofs of correctness.

4.6 Summary

In this chapter, we described a novel approach to the verification of consistency models implemented in distributed storage systems. We based our work on a semantic model defining consistency conditions by means of logic predicates over graph entities describing ordering and mutual visibility of operations. By leveraging the expressiveness of this model, we designed Conver, a consistency verification framework that follows the principles of property-based testing: Conver generates random test cases and run them in the attempt to falsify the consistency semantics defined as execution invariants. We believe that this contribution will prove instrumental to improve over the state-of-the-art on correctness verification of real world storage systems. In the next chapter, we further analyze possible research directions related to this work.

Chapter 5

Conclusion

In this thesis, we presented a set of contributions towards the goals of defining, composing and testing consistency semantics for distributed storage systems. In the following, we review those contributions and reflect on future research directions.

5.1 Towards a cross-stack principled approach to consistency

In Chapter 2, we provided a formal taxonomy of more than fifty consistency semantics defined in four decades of research. Such survey represents a preliminary step of normalization that will prove useful when sifting through the past literature, or evaluating and comparing claims about consistency semantics [78]. Indeed, by dismissing imprecise or inaccurate definitions, and by relating similar semantics to each other, we helped reducing the clutter and occasional equivocations on the matter. In this way, hopefully, future scholars will find easier the ongoing quest for the sweet spot between performance, correctness and fault tolerance.

Moreover, our work may allow for more clear and rigorous agreements on quality of service guarantees (SLA) between users and storage providers/vendors, in the vein of what has been proposed by Terry et al. [230]. The declarative approach we adopted could also promote more structured development and testing techniques, as we described in Chapter 4 or as in the recent work by Sivaramakrishnan et al. [219], which concretizes the paradigm of *design-by-contract* programming.

Finally, our survey may serve as a first step in a bottom-up strategy to work out the mapping between high-level application invariants and storage semantics. Indeed, while the focus of this thesis has been chiefly on low-level read/write semantics, a current avenue of research is about establishing the operational and semantic links between storage-level semantics and application invariants [34, 41]. In turn, this mapping may uncover potential opportunity for cross-stack optimizations and co-design [18], and enable further automation in the choice and enforcement of performance vs consistency tradeoffs [170, 125].

5.2 Planet-scale consistency and the challenges of composition

In Chapter 3, we described a storage system that reaps the benefits of cloud storage without incurring the cost of wide area cross-cloud coordination, unlike previous approaches [53]. In a sense, Hybris is yet another attempt at dispelling the dichotomy between safety and scalability, and it serves as an example of how we can circumvent theoretical impossibilities through practical and considerate tradeoffs. Other instances of pragmatic tradeoffs include recent works that propose techniques to minimize coordination [37, 172] or alleviate its cost [192, 102].

Moreover, Hybris makes use of two kinds of resources, different in their fault and timing assumptions, and providing different consistency guarantees. Hence, it demonstrates that, in certain settings, it is possible to leverage composition of models and semantics to conceal shortcomings of individual components while retaining the overall benefits. In Hybris' case we cover up faulty and weakly consistent clouds with a negligible amount of metadata and some coordination in limited, private settings. We believe that there may be unexplored potential benefits in the composition of systems that provide different semantics, or in the decomposition of established protocols into their constituent parts — as showed, for instance, in the work by Zhang et al. [257], which decouples fault tolerance from ordering requirements across the transactional protocol stack. Clearly, the requisite of this endeavors is a thorough understanding of the individual semantics and models. In this regard, the taxonomy provided in Chapter 2 represents a solid formal base, while Hybris, presented in Chapter 3, serves as a practical instance of a beneficial composition.

5.3 Verification of distributed systems correctness

Testing distributed systems is a notoriously difficult endeavor. Numerous approaches have been devised and implemented that provide a range of tradeoffs in matter of confidence and usability. While most of nowadays distributed software is still being tested with traditional unit and integration tests, some companies that operate large deployments have implemented sophisticated monitoring and tracing tools, that allow for thorough inspection and troubleshooting of production systems [216, 234]. Formal methods represent a valid alternative to ensure the correctness of distributed systems, as they are used to formally model a system and prove its correctness. They offer a strong degree of confidence but require non-negligible prior knowledge of languages and tools to develop faithful models or write formal specifications. Recently, the increased efficiency of solvers has enabled the adoption of what are sometimes referred to as "lightweight formal methods," i.e. the use of SAT/SMT solvers to test specific high-level invariants of software systems [125]. Another convenient middle ground is represented by the "smart testing" techniques, which include fault injection, directed random testing, deterministic simulations and property-based testing. The quest for a sweet spot in the usability vs confidence spectrum is particularly sensible for verification of semantic properties such as consistency.

Indeed, according to Rice's Theorem, the problem of deciding whether a program satisfies a semantic property is undecidable. Hence the need for practical solutions that trade certainty of correctness for usability.

In Chapter 4 we introduced a principled approach to consistency verification that follows the tenets of property-based testing. We based this approach on the declarative, graph-based framework that we described in Chapter 2: consistency is regarded as logic predicates that must be respected throughout executions, i.e. a set of invariants. We stress-test the preservation of these invariants by applying the techniques of random testing and fault injection. These ideas are implemented in Conver: a utility to perform confidence/sanity tests (also called "smoke tests") while developing database applications. Running a distributed test with Conver takes just a few seconds, thus it can be conveniently integrated and run with existing test suites. Besides, Conver is easily portable to any non-transactional store, thanks also to the increasing support for container-based clustered systems. Thus, Conver can become a standard tool to verify that off-the-shelf systems actually provide the semantics that they claim in their documentation. In turn, this may yield to a better comprehension and a more rigorous documentation of the software, which would be especially beneficial for commercial systems subject to quality agreements.

Finally, Conver represents also an effort to *materialize* the subtle yet perilous side effects of inconsistency in data management systems. Our aim in designing Conver was twofold: we wanted both to raise awareness about consistency issues among developers, and to concretize the outcomes entailed by the theoretical definitions we introduced in Chapter 2. Thus, Conver can not only be another tool in developers' tool belt, but it can also serve as an instrument to further the study of data storage semantics.

Bibliography

[1] "Alloy solver," http://alloy.mit.edu/alloy; visited February 2016.

[2] "PropEr testing of generic servers," http://proper.softlab.ntua.gr/Tutorials/PropEr_testing_of_generic_servers.html; visited February 2016.

[3] I. Abraham, G. Chockler, I. Keidar, and D. Malkhi, "Byzantine Disk Paxos: Optimal Resilience with Byzantine Shared Memory," *Distributed Computing*, vol. 18, no. 5, 2006.

[4] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon, "RACS: a case for cloud storage diversity," in *SoCC*, 2010. Online: http://doi.acm.org/10.1145/1807128.1807165

[5] S. Adve and M. D. Hill, "Weak ordering - a new definition," in *International Symposium on Computer Architecture*, 1990.

[6] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29, no. 12, 1996.

[7] S. V. Adve and M. D. Hill, "A unified formalization of four shared-memory models," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 6, 1993. Online: http://doi.ieeecomputersociety.org/10.1109/71.242161

[8] A. Adya, "Weak consistency: A generalized theory and optimistic implementations for distributed transactions," Ph.D. Dissertation, MIT, Mar. 1999, also as Technical Report MIT/LCS/TR-786.

[9] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer, "FARSITE: federated, available, and reliable storage for an incompletely trusted environment," in *OSDI*, 2002. Online: http://www.usenix.org/events/osdi02/tech/adya.html

[10] M. Ahamad, R. A. Bazzi, R. John, P. Kohli, and G. Neiger, "The power of processor consistency," in *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 1993. Online: http://doi.acm.org/10.1145/165231.165264

[11] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, "Causal memory: Definitions, implementation, and programming," *Distributed Computing*, vol. 9, no. 1, 1995.

[12] AIimpacts.org, "A history of storage cost," http://www.mkomo.com/cost-per-gigabyte-update, 2014.

[13] A. S. Aiyer, L. Alvisi, and R. A. Bazzi, "On the availability of non-strict quorum systems," in *DISC*, 2005. Online: http://dx.doi.org/10.1007/11561927_6

[14] S. Almeida, J. Leitão, and L. Rodrigues, "Chainreaction: a causal+ consistent datastore based on chain replication," in *EuroSys*, 2013.

[15] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears, "Boom analytics: exploring data-centric, declarative programming for the cloud," in *EuroSys*, 2010. Online: http://doi.acm.org/10.1145/1755913.1755937

[16] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak, "Consistency analysis in bloom: a CALM and collected approach," in *Conference on Innovative Data Systems Research (CIDR), 2011*, 2011.

[17] P. Alvaro, A. Hutchinson, N. Conway, W. R. Marczak, and J. M. Hellerstein, "Bloomunit: declarative testing for distributed programs," in *DBTest*, 2012. Online: http://doi.acm.org/10.1145/2304510.2304512

[18] P. Alvaro, P. Bailis, N. Conway, and J. M. Hellerstein, "Consistency without borders," in *SOCC*, 2013. Online: http://doi.acm.org/10.1145/2523616.2523632

[19] P. Alvaro, N. Conway, J. M. Hellerstein, and D. Maier, "Blazes: Coordination analysis for distributed programs," in *IEEE Conference on Data Engineering (ICDE), 2014*, 2014.

[20] Amazon, "Amazon DynamoDB Pricing," 2016, online: https://aws.amazon.com/dynamodb/pricing/; visited September 2016.

[21] American National Standard for Information Systems, "ANSI x3. 135-1992, Database Language – SQL," 1992.

[22] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "FAWN: a fast array of wimpy nodes," in *SOSP*, 2009. Online: http://doi.acm.org/10.1145/1629575.1629577

[23] E. Anderson, X. Li, M. A. Shah, J. Tucek, and J. J. Wylie, "What consistency does your key-value store actually provide?" in *Hot Topics in System Dependability*, ser. HotDep'10, 2010. Online: http://dl.acm.org/citation.cfm?id=1924908.1924919

[24] E. Androulaki, C. Cachin, D. Dobre, and M. Vukolic, "Erasure-coded byzantine storage with separate metadata," in *OPODIS*, 2014. Online: http://dx.doi.org/10.1007/978-3-319-14472-6_6

[25] Apache, "Apache JClouds," 2016, online: http://jclouds.apache.org/; visited September 2016.

[26] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, 2010. Online: http://doi.acm.org/10.1145/1721654.1721672

[27] H. Attiya and R. Friedman, "A correctness condition for high-performance multiprocessors (extended abstract)," in *ACM Symposium on Theory of Computing*, 1992. Online: http://doi.acm.org/10.1145/129712.129778

[28] H. Attiya and J. Welch, "Sequential consistency versus linearizability," *ACM Transactions on Computer Systems (TOCS)*, vol. 12, no. 2, May 1994.

[29] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," *Journal of the ACM*, vol. 42, no. 1, 1995. Online: http://doi.acm.org/10.1145/200836.200869

[30] H. Attiya, F. Ellen, and A. Morrison, "Limitations of highly-available eventually-consistent data stores," in *PODC*, Jul. 2015. Online: http://doi.acm.org/10.1145/2767386.2767419

[31] S. Babu and H. Herodotou, "Massively parallel databases and mapreduce systems," *Foundations and Trends in Databases*, vol. 5, no. 1, 2013. Online: http://dx.doi.org/10.1561/1900000036

[32] P. Bailis, "When is "ACID" ACID? rarely." http://www.bailis.org/blog/when-is-acid-acid-rarely/; visited January 2017.

[33] P. Bailis and A. Ghodsi, "Eventual consistency today: Limitations, extensions, and beyond," *Queue*, vol. 11, no. 3, Mar. 2013. Online: http://doi.acm.org/10.1145/2460276.2462076

[34] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "The potential dangers of causal consistency and an explicit solution," in *SOCC*, 2012. Online: http://doi.acm.org/10.1145/2391229.2391251

[35] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica, "Probabilistically bounded staleness for practical partial quorums," *VLDB*, vol. 5, no. 8, 2012.

[36] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Bolt-on causal consistency," in *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2013.

[37] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Coordination avoidance in database systems," *PVLDB*, 2014. Online: http://www.vldb.org/pvldb/vol8/p185-bailis.pdf

[38] P. Bailis, A. Fekete, J. M. Hellerstein, A. Ghodsi, and I. Stoica, "Scalable atomic visibility with RAMP transactions," in *ACM International Conference on Management of Data (SIGMOD)*, 2014. Online: http://doi.acm.org/10.1145/2588555.2588562

[39] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica, "Quantifying eventual consistency with PBS," *VLDB Journal*, vol. 23, no. 2, 2014. Online: http://dx.doi.org/10.1007/s00778-013-0330-1

[40] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *Conference on Innovative Data Systems Research (CIDR), 2011*, 2011.

[41] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. M. Preguiça, M. Najafzadeh, and M. Shapiro, "Putting consistency back into eventual consistency," in *EuroSys*, 2015. Online: http://doi.acm.org/10.1145/2741948.2741972

[42] C. Basescu, C. Cachin, I. Eyal, R. Haas, A. Sorniotti, M. Vukolić, and I. Zachevsky, "Robust data sharing with key-value stores," in *IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*, 2012. Online: http://dx.doi.org/10.1109/DSN.2012.6263920

[43] J. Bataller and J. M. Bernabéu-Aubán, "Synchronized DSM models," in *Parallel Processing (Euro-Par)*, 1997. Online: http://dx.doi.org/10.1007/BFb0002771

[44] N. M. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng, "PRACTI replication," in *NSDI*, 2006. Online: http://www.usenix.org/events/nsdi06/tech/belaramani.html

[45] D. Bermbach, *Benchmarking eventually consistent distributed storage systems*. KIT Scientific Publishing, 2014. Online: http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/3143625

[46] D. Bermbach and J. Kuhlenkamp, "Consistency in distributed storage systems - an overview of models, metrics and measurement approaches," in *Networked Systems (NETYS), 2013*, 2013.

[47] D. Bermbach and S. Tai, "Eventual consistency: How soon is eventual? An evaluation of amazon s3's consistency behavior," in *ACM Workshop on Middleware for Service Oriented Computing*, 2011. Online: http://doi.acm.org/10.1145/2093185.2093186

[48] ——, "Benchmarking eventual consistency: Lessons learned from long-term experimental studies," in *IEEE International Conference on Cloud Engineering (IC2E)*, 2014. Online: http://dx.doi.org/10.1109/IC2E.2014.37

[49] P. A. Bernstein and S. Das, "Rethinking eventual consistency," in *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2013. Online: http://doi.acm.org/10.1145/2463676.2465339

[50] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. Online: http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx

[51] B. N. Bershad and M. J. Zekauskas, "Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors," Tech. Rep., 1991.

[52] A. Bessani, R. Mendes, T. Oliveira, N. Neves, M. Correia, M. Pasin, and P. Verissimo, "SCFS: A shared cloud-backed file system," in *USENIX Annual Technical Conference, USENIX ATC*, 2014.

[53] A. N. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "DepSky: Dependable and secure storage in a cloud-of-clouds," *ACM Transactions on Storage (TOS)*, vol. 9, no. 4, 2013. Online: http://doi.acm.org/10.1145/2535929

[54] A. N. Bessani, R. Mendes, T. Oliveira, N. F. Neves, M. Correia, M. Pasin, and P. Veríssimo, "SCFS: A shared cloud-backed file system," in *USENIX Annual Technical Conference (ATC), 2014*, 2014. Online: https://www.usenix.org/conference/atc14/technical-sessions/presentation/bessani

[55] C. E. B. Bezerra, F. Pedone, and R. van Renesse, "Scalable state-machine replication," in *IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*, 2014. Online: http://dx.doi.org/10.1109/DSN.2014.41

[56] K. Birman, A. Schiper, and P. Stephenson, "Lightweight causal and atomic group multicast," *ACM Transactions on Computer Systems (TOCS)*, vol. 9, no. 3, 1991.

[57] V. Bortnikov, G. Chockler, A. Roytman, and M. Spreitzer, "Bulletin board: A scalable and robust eventually consistent shared memory over a peer-to-peer overlay," *Operating Systems Review*, vol. 44, no. 2, Apr. 2010. Online: http://doi.acm.org/10.1145/1773912.1773929

[58] K. D. Bowers, A. Juels, and A. Oprea, "HAIL: a high-availability and integrity layer for cloud storage," in *ACM CCS*, 2009. Online: http://doi.acm.org/10.1145/1653662.1653686

[59] M. Brandenburger, C. Cachin, and N. Knezevic, "Don't trust the cloud, verify: integrity and consistency for cloud object stores," in *ACM International Systems and Storage Conference (SYSTOR)*, 2015. Online: http://doi.acm.org/10.1145/2757667.2757681

[60] E. A. Brewer, "Pushing the CAP: Strategies for consistency and availability," *IEEE Computer*, vol. 45, no. 2, 2012. Online: http://doi.ieeecomputersociety.org/10.1109/MC.2012.37

[61] ——, "Towards robust distributed systems (abstract)," in *PODC*, 2000.

[62] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "TAO: facebook's distributed data store for the social graph," in *USENIX Annual Technical Conference (ATC)*, 2013. Online: https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson

[63] J. Brzezinski, C. Sobaniec, and D. Wawrzyniak, "Session guarantees to achieve pram consistency of replicated shared objects," in *Parallel Processing and Applied Mathematics (PPAM)*, 2003.

[64] ——, "From session causality to causal consistency," in *Workshop on Parallel, Distributed and Network-Based Processing (PDP)*, 2004. Online: http://dx.doi.org/10.1109/EMPDP.2004.1271440

[65] S. Burckhardt, *Principles of Eventual Consistency*, ser. Foundations and Trends in Programming Languages. now publishers, October 2014, vol. 1, no. 1-2. Online: http://research.microsoft.com/apps/pubs/default.aspx?id=230852

[66] S. Burckhardt, M. Fähndrich, D. Leijen, and B. P. Wood, "Cloud types for eventual consistency," in *Object-Oriented Programming (ECOOP), 2012*, 2012. Online: http://dx.doi.org/10.1007/978-3-642-31057-7_14

[67] S. Burckhardt, D. Leijen, M. Fähndrich, and M. Sagiv, "Eventually consistent transactions," in *European Symposium on Programming (ESOP), Held as Part of ETAPS 2012*, 2012. Online: http://dx.doi.org/10.1007/978-3-642-28869-2_4

[68] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski, "Replicated data types: specification, verification, optimality," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2014. Online: http://doi.acm.org/10.1145/2535838.2535848

[69] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *OSDI*, 2006. Online: http://www.usenix.org/events/osdi06/tech/burrows.html

[70] C. Cachin, A. Shelat, and A. Shraer, "Efficient fork-linearizable access to untrusted shared memory," in *PODC*, 2007.

[71] C. Cachin, I. Keidar, and A. Shraer, "Trusting the cloud," *SIGACT News*, vol. 40, no. 2, 2009. Online: http://doi.acm.org/10.1145/1556154.1556173

[72] ——, "Fork Sequential Consistency is Blocking," *Information Processing Letters*, vol. 109, no. 7, 2009.

[73] ——, "Fail-aware untrusted storage," *SIAM Journal on Computing (SICOMP)*, vol. 40, no. 2, 2011. Online: http://dx.doi.org/10.1137/090751062

[74] C. Cachin, D. Dobre, and M. Vukolic, "Separating data and control: Asynchronous BFT storage with 2t + 1 data replicas," in *Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2014. Online: http://dx.doi.org/10.1007/978-3-319-11764-5_1

[75] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. ul Haq, M. I. ul Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas, "Windows azure storage: a highly available cloud storage service with strong consistency," in *SOSP*, 2011.

[76] I. Cano, S. Aiyar, and A. Krishnamurthy, "Characterizing private clouds: A large-scale empirical analysis of enterprise clusters," in *SoCC*, 2016. Online: http://doi.acm.org/10.1145/2987550.2987584

[77] A. Cerone, G. Bernardi, and A. Gotsman, "A framework for transactional consistency models with atomic visibility," in *Conference on Concurrency Theory, (CONCUR), 2015*, 2015. Online: http://dx.doi.org/10.4230/LIPIcs.CONCUR.2015.58

[78] A. Cerone, A. Gotsman, and H. Yang, "Algebraic laws for weak consistency," in *CONCUR*, 2017.

[79] H.-E. Chihoub, S. Ibrahim, G. Antoniu, and M. S. Pérez-Hernández, "Harmony: Towards automated self-adaptive consistency in cloud storage," in *IEEE International Conference on Cluster Computing (CLUSTER)*, 2012.

[80] ——, "Consistency in the cloud: When money does matter!" in *Symposium on Cluster, Cloud, and Grid Computing (CCGrid), 2013*, 2013.

[81] G. Chockler, D. Dobre, and A. Shraer, "Brief announcement: Consistency and complexity tradeoffs for highly-available multi-cloud store," in *DISC*, 2013.

[82] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, "Attested append-only memory: making adversaries stick to their word," in *SOSP*, 2007. Online: http://doi.acm.org/10.1145/1294261.1294280

[83] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," in *ACM SIGPLAN ICFP*, 2000. Online: http://doi.acm.org/10.1145/351240.351266

[84] Cloudera, "Observers - making ZooKeeper scale even further," 2009, online: https://blog.cloudera.com/blog/2009/12/observers-making-zookeeper-scale-even-further/; visited September 2016.

[85] CloudSpaces, "CloudSpaces EU FP7 project." 2015, online: http://cloudspaces.eu/; visited September 2016.

[86] B. A. Coan, B. M. Oki, and E. K. Kolodner, "Limitations on database availability when networks partition," in *ACM Symposium on Principles of Distributed Computing, 1986*, 1986. Online: http://doi.acm.org/10.1145/10590.10606

[87] Consul, "Consul - distributed service discovery and configuration," 2016, online: https://www.consul.io/; visited September 2016.

[88] ——, "Consul - consensus protocol," 2016, online: https://www.consul.io/docs/internals/consensus.html; visited September 2016.

[89] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier, "Logic and lattices for distributed programming," in *SOCC*, 2012. Online: http://doi.acm.org/10.1145/2391229.2391230

[90] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *VLDB*, vol. 1, no. 2, 2008.

[91] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *SoCC*, 2010. Online: http://doi.acm.org/10.1145/1807128.1807152

[92] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, 2013. Online: http://doi.acm.org/10.1145/2491245

[93] M. Correia, N. F. Neves, and P. Veríssimo, "How to tolerate half less one Byzantine nodes in practical distributed systems," in *SRDS*, 2004. Online: http://doi.ieeecomputersociety.org/10.1109/RELDIS.2004.1353018

[94] M. Correia, D. G. Ferro, F. P. Junqueira, and M. Serafini, "Practical hardening of crash-tolerant systems," in *USENIX Annual Technical Conference*, 2012. Online: https://www.usenix.org/conference/atc12/technical-sessions/presentation/correia

[95] S. B. Davidson, H. Garcia-Molina, and D. Skeen, "Consistency in partitioned networks," *ACM Computing Surveys*, vol. 17, no. 3, 1985.

[96] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *SOSP*, 2007.

[97] P. Deutsch, "The eight fallacies of distributed computing," https://blogs.oracle.com/jag/resource/Fallacies.html, 1994.

[98] D. Dobre, P. Viotti, and M. Vukolić, "Hybris: Robust hybrid cloud storage," in *SOCC*, 2014. Online: http://doi.acm.org/10.1145/2670979.2670991

[99] I. Drago, M. Mellia, M. M. Munafò, A. Sperotto, R. Sadre, and A. Pras, "Inside dropbox: understanding personal cloud storage services," in *ACM SIGCOMM Internet Measurement Conference, IMC '12*, 2012. Online: http://doi.acm.org/10.1145/2398776.2398827

[100] I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras, "Benchmarking personal cloud storage," in *ACM SIGCOMM Internet Measurement Conference, IMC 2013*, 2013. Online: http://doi.acm.org/10.1145/2504730.2504762

[101] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, "Gentlerain: Cheap and scalable causal consistency with physical clocks," in *SOCC*, 2014. Online: http://doi.acm.org/10.1145/2670979.2670983

[102] J. Du, D. Sciascia, S. Elnikety, W. Zwaenepoel, and F. Pedone, "Clock-rsm: Low-latency inter-datacenter state machine replication using loosely synchronized physical clocks," in *IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*, 2014. Online: http://dx.doi.org/10.1109/DSN.2014.42

[103] M. Dubois, C. Scheurich, and F. A. Briggs, "Memory access buffering in multiprocessors," in *International Symposium on Computer Architecture (ISCA)*, 1986.

[104] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, Apr. 1988.

[105] D. Dziuma, P. Fatourou, and E. Kanellou, "Consistency for transactional memory computing," *Bulletin of the EATCS*, vol. 113, 2014. Online: http://eatcs.org/beatcs/index.php/beatcs/article/view/288

[106] Eurecom, "Hybris - robust hybrid cloud storage library," 2016, online: https://github.com/pviotti/hybris; visited September 2016.

[107] H. Fan, S. Chatterjee, and W. M. Golab, "Watca: The waterloo consistency analyzer," in *IEEE International Conference on Data Engineering (ICDE)*, 2016. Online: http://dx.doi.org/10.1109/ICDE.2016.7498354

[108] R. Fan and N. Lynch, "Efficient Replication of Large Data Objects," in *DISC*, 2003.

[109] A. Fekete, D. Gupta, V. Luchangco, N. A. Lynch, and A. A. Shvartsman, "Eventually-serializable data services," in *PODC*, 1996. Online: http://doi.acm.org/10.1145/248052.248113

[110] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten, "SPORC: group collaboration using untrusted cloud resources," in *OSDI*, 2010. Online: http://www.usenix.org/events/osdi10/tech/full_papers/Feldman.pdf

[111] M. J. Fischer, N. A. Lynch, and M. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2, 1985. Online: http://doi.acm.org/10.1145/3149.214121

[112] R. Friedman, R. Vitenberg, and G. Chockler, "On the composability of consistency conditions," *Information Processing Letters*, vol. 86, no. 4, 2003. Online: http://dx.doi.org/10.1016/S0020-0190(02)00498-2

[113] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions," in *USENIX Conference on File and Storage Technologies (FAST 17)*, 2017. Online: https://www.usenix.org/conference/fast17/technical-sessions/presentation/ganesan

[114] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions," in *FAST*, 2017. Online: https://www.usenix.org/conference/fast17/technical-sessions/presentation/ganesan

[115] G. R. Gao and V. Sarkar, "Location consistency-a new memory model and cache consistency protocol," *IEEE Transactions on Computers*, vol. 49, no. 8, 2000. Online: http://doi.ieeecomputersociety.org/10.1109/12.868026

[116] K. Gharachorloo, D. Lenoski, J. Laudon, P. B. Gibbons, A. Gupta, and J. L. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *International Symposium on Computer Architecture*, 1990.

[117] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *SOSP*, 2003. Online: http://doi.acm.org/10.1145/945445.945450

[118] P. B. Gibbons and E. Korach, "Testing shared memories," *SIAM J. Comput.*, vol. 26, no. 4, 1997. Online: http://dx.doi.org/10.1137/S0097539794279614

[119] G. A. Gibson, D. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka, "A cost-effective, high-bandwidth storage architecture," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998. Online: http://doi.acm.org/10.1145/291069.291029

[120] S. Gilbert and N. A. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, 2002.

[121] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. E. Anderson, "Scalable consistency in scatter," in *SOSP*, 2011.

[122] W. M. Golab, X. Li, and M. A. Shah, "Analyzing consistency properties for fun and profit," in *PODC*, 2011.

[123] W. M. Golab, M. R. Rahman, A. AuYoung, K. Keeton, and I. Gupta, "Client-centric benchmarking of eventual consistency for cloud storage systems," in *IEEE Conference on Distributed Computing Systems (ICDCS), 2014*, 2014. Online: http://dx.doi.org/10.1109/ICDCS.2014.57

[124] J. R. Goodman, "Cache consistency and sequential consistency," SCI Commitee, Tech. Rep. no. 61, March 1989.

[125] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro, "'Cause I'm strong enough: Reasoning about consistency choices in distributed systems," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2016. Online: http://lip6.fr/Marc.Shapiro/papers/CISE-POPL-2016.pdf

[126] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[127] R. Guerraoui and M. Vukolic, "How fast can a very robust read be?" in *PODC*, 2006. Online: http://doi.acm.org/10.1145/1146381.1146419

[128] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar, "Why does the cloud stop computing?: Lessons from hundreds of service outages," in *SoCC*, 2016. Online: http://doi.acm.org/10.1145/2987550.2987583

[129] V. Hadzilacos and S. Toueg, "A modular approach to fault-tolerant broadcasts and related problems," Cornell University, Department of Computer Science, Tech. Rep., 1994.

[130] R. Halalai, P. Sutra, E. Riviere, and P. Felber, "Zoofence: Principled service partitioning and application to the zookeeper coordination service," in *IEEE International Symposium on Reliable Distributed Systems*, 2014. Online: http://dx.doi.org/10.1109/SRDS.2014.41

[131] J. Hamilton, "The cost of latency," 2009, online: http://perspectives.mvdirona.com/2009/10/the-cost-of-latency/; visited September 2016.

[132] R. Hamlet, "Random testing," *Encyclopedia of Software Engineering*, 1994.

[133] S. Han, H. Shen, T. Kim, A. Krishnamurthy, T. E. Anderson, and D. Wetherall, "Metasync: File synchronization across multiple untrusted storage services," in *USENIX Annual Technical Conference (ATC)*, 2015. Online: https://www.usenix.org/conference/atc15/technical-session/presentation/han

[134] T. Harris, J. R. Larus, and R. Rajwar, *Transactional Memory, 2nd edition*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010. Online: http://dx.doi.org/10.2200/S00272ED1V01Y201006CAC011

[135] P. Helland, "Life beyond distributed transactions: an apostate's opinion," in *Conference on Innovative Data Systems Research (CIDR), 2007*, 2007.

[136] M. Herlihy, "Wait-Free Synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, 1991.

[137] M. Herlihy and N. Shavit, *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[138] M. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, 1990.

[139] M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, 1990.

[140] A. Horn and D. Kroening, "Faster linearizability checking via p-compositionality," in *IFIP International Conference on Formal Techniques for Distributed Objects, Components, and Systems, FORTE*, 2015.

[141] J. Hughes, B. C. Pierce, T. Arts, and U. Norell, "Mysteries of dropbox: Property-based testing of a distributed synchronization service," in *IEEE International Conference on Software Testing, Verification and Validation, ICST*, 2016. Online: http://dx.doi.org/10.1109/ICST.2016.37

[142] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: wait-free coordination for internet-scale systems," in *USENIX Annual Technical Conference, USENIX ATC*, 2010.

[143] P. W. Hutto and M. Ahamad, "Slow memory: Weakening consistency to enchance concurrency in distributed shared memories," in *International Conference on Distributed Computing Systems (ICDCS)*, 1990.

[144] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li, "Improving release-consistent shared virtual memory using automatic update," in *IEEE International Symposium on High-Performance Computer Architecture (HPCA), 1996*, 1996. Online: http://dx.doi.org/10.1109/HPCA.1996.501170

[145] L. Iftode, J. P. Singh, and K. Li, "Scope consistency: A bridge between release consistency and entry consistency," in *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 1996.

[146] P. R. Johnson and R. H. Thomas, "Maintenance of duplicate databases," Internet Requests for Comments, RFC Editor, RFC 677, January 1975, http://www.rfc-editor.org/rfc/rfc677.txt. Online: http://www.rfc-editor.org/rfc/rfc677.txt

[147] E. Junqué de Fortuny, D. Martens, and F. Provost, "Predictive modeling with big data: is bigger really better?" *Big Data*, vol. 1, no. 4, 2013.

[148] F. P. Junqueira, I. Kelly, and B. Reed, "Durability with bookkeeper," *Operating Systems Review*, vol. 47, no. 1, 2013. Online: http://doi.acm.org/10.1145/2433140.2433144

[149] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel, "CheapBFT: resource-efficient Byzantine fault tolerance," in *EuroSys*, 2012. Online: http://doi.acm.org/10.1145/2168836.2168866

[150] P. J. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy release consistency for software distributed shared memory," in *International Symposium on Computer Architecture, 1992*, 1992. Online: http://doi.acm.org/10.1145/139669.139676

[151] K. Kingsbury, "Jepsen - distributed systems safety analysis," visited February 2016. Online: http://jepsen.io/

[152] M. Komorowski, "Trends in the cost of computing," http://aiimpacts.org/trends-in-the-cost-of-computing/, 2015.

[153] R. Kotla, L. Alvisi, and M. Dahlin, "Safestore: A durable and practical storage system," in *USENIX Annual Technical Conference, USENIX ATC*, 2007.

[154] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann, "Consistency rationing in the cloud: Pay only when it matters," *VLDB*, vol. 2, no. 1, 2009.

[155] S. Krishnamurthy, W. H. Sanders, and M. Cukier, "An adaptive framework for tunable consistency and timeliness using replication," in *Dependable Systems and Networks (DSN)*, 2002.

[156] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat, "Providing high availability using lazy replication," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 4, 1992. Online: http://doi.acm.org/10.1145/138873.138877

[157] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *Operating Systems Review*, vol. 44, no. 2, 2010. Online: http://doi.acm.org/10.1145/1773912.1773922

[158] S. Lakshmanan, M. Ahamad, and H. Venkateswaran, "A secure and highly available distributed store for meeting diverse data storage needs," in *Dependable Systems and Networks (DSN)*, 2001. Online: http://doi.ieeecomputersociety.org/10.1109/DSN.2001.941410

[159] L. Lamport, "Paxos made simple," *SIGACT News*, vol. 32, no. 4, 2001. Online: http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf

[160] ——, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM (CACM)*, vol. 21, no. 7, 1978.

[161] ——, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. 28, no. 9, 1979.

[162] ——, "Specifying concurrent program modules," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 5, no. 2, 1983. Online: http://doi.acm.org/10.1145/69624.357207

[163] ——, "On interprocess communication. part i: Basic formalism," *Distributed Computing*, vol. 1, no. 2, 1986.

[164] ——, "On interprocess communication. part ii: Algorithms," *Distributed Computing*, vol. 1, no. 2, 1986.

[165] L. Lamport, R. E. Shostak, and M. C. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, 1982. Online: http://doi.acm.org/10.1145/357172.357176

[166] G. T. Laycock, "The theory and practice of specification based software testing," Ph.D. dissertation, University of Sheffield, 1993.

[167] C. Lee, S. J. Park, A. Kejriwal, S. Matsushita, and J. K. Ousterhout, "Implementing linearizability at large scale and low latency," in *SOSP*, 2015. Online: http://doi.acm.org/10.1145/2815400.2815416

[168] M. Lesani, C. J. Bell, and A. Chlipala, "Chapar: certified causally consistent distributed key-value stores," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2016. Online: http://doi.acm.org/10.1145/2837614.2837622

[169] A. Li, X. Yang, S. Kandula, and M. Zhang, "Cloudcmp: comparing public cloud providers," in *ACM SIGCOMM IMC*, 2010. Online: http://doi.acm.org/10.1145/1879141.1879143

[170] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," in *OSDI*, 2012. Online: http://dl.acm.org/citation.cfm?id=2387880.2387906

[171] C. Li, J. Leitão, A. Clement, N. M. Preguiça, R. Rodrigues, and V. Vafeiadis, "Automating the choice of consistency levels in replicated systems," in *USENIX Annual Technical Conference (ATC), 2014.*, 2014. Online: https://www.usenix.org/conference/atc14/technical-sessions/presentation/li_cheng_2

[172] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports, "Just say NO to paxos overhead: Replacing consensus with network ordering," in *OSDI*, 2016. Online: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/li

[173] J. Li and D. Mazières, "Beyond one-third faulty replicas in byzantine fault tolerant systems," in *NSDI*, 2007.

[174] J. Li, M. N. Krohn, D. Mazières, and D. Shasha, "Secure untrusted data repository (sundr)," in *OSDI*, 2004.

[175] R. J. Lipton and J. S. Sandberg, "Pram: A scalable shared memory," Princeton University, Tech. Rep. CS-TR-180-88, 1988.

[176] S. Liu, P. Viotti, C. Cachin, V. Quéma, and M. Vukolic, "XFT: practical fault tolerance beyond crashes," in *USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2016. Online: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/liu

[177] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: scalable causal consistency for wide-area storage with cops," in *SOSP*, 2011.

[178] ——, "Stronger semantics for low-latency geo-replicated storage," in *NSDI*, 2013. Online: http://dl.acm.org/citation.cfm?id=2482626.2482657

[179] B. T. Loo, T. Condie, M. N. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica, "Declarative networking: language, execution and optimization," in *ACM SIGMOD International Conference on Management of Data*, 2006. Online: http://doi.acm.org/10.1145/1142473.1142485

[180] P. G. Lopez, S. Toda, C. Cotes, M. Sanchez-Artigas, and J. Lenton, "Stacksync: Bringing elasticity to dropbox-like file synchronization," in *ACM/IFIP/USENIX Middleware*, 2014.

[181] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd, "Existential consistency: measuring and understanding consistency at facebook," in *SOSP*, 2015. Online: http://doi.acm.org/10.1145/2815400.2815426

[182] N. A. Lynch and A. A. Shvartsman, "RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks," in *DISC*, 2002.

[183] N. A. Lynch and M. R. Tuttle, "An introduction to input/output automata," *CWI Quarterly*, vol. 2, 1989.

[184] P. Mahajan, S. T. V. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: Cloud storage with minimal trust," in *OSDI*, 2010.

[185] P. Mahajan, L. Alvisi, and M. Dahlin, "Consistency, availability, and convergence," Computer Science Department, University of Texas at Austin, Tech. Rep. TR-11-22, 2011.

[186] P. Mahajan, S. T. V. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: Cloud storage with minimal trust," *ACM Trans. Comput. Syst.*, vol. 29, no. 4, 2011.

[187] D. Malkhi and M. K. Reiter, "Secure and scalable replication in phalanx," in *Symposium on Reliable Distributed Systems (SRDS), 1998*, 1998. Online: http://dx.doi.org/10.1109/RELDIS.1998.740474

[188] ——, "Byzantine quorum systems," *Distributed Computing*, vol. 11, no. 4, 1998. Online: http://dx.doi.org/10.1007/s004460050050

[189] D. Mazières and D. Shasha, "Building Secure File Systems out of Byzantine Storage," in *PODC*, 2002.

[190] Memcached, "Memcached," 2016, online: http://memcached.org/; visited September 2016.

[191] J. Misra, "Axioms for memory access in asynchronous hardware systems," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 8, no. 1, 1986.

[192] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *SOSP*, 2013. Online: http://doi.acm.org/10.1145/2517349.2517350

[193] D. Mosberger, "Memory consistency models," *Operating Systems Review*, vol. 27, no. 1, 1993. Online: http://doi.acm.org/10.1145/160551.160553

[194] M. Najafzadeh, A. Gotsman, H. Yang, C. Ferreira, and M. Shapiro, "The CISE tool: proving weakly-consistent applications correct," in *Workshop on the Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys*, 2016. Online: http://doi.acm.org/10.1145/2911151.2911160

[195] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm," in *USENIX Annual Technical Conference, USENIX ATC*, 2014. Online: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[196] A. Oprea and M. K. Reiter, "On consistency of encrypted files," in *DISC*, 2006.

[197] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems, Third Edition*. Springer, 2011. Online: http://dx.doi.org/10.1007/978-1-4419-8834-8

[198] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi, "YCSB++: benchmarking and performance debugging advanced features in scalable table stores," in *SOCC*, 2011. Online: http://doi.acm.org/10.1145/2038916.2038925

[199] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults," *J. ACM*, vol. 27, no. 2, 1980.

[200] D. Perkins, N. Agrawal, A. Aranya, C. Yu, Y. Go, H. V. Madhyastha, and C. Ungureanu, "Simba: tunable end-to-end data consistency for mobile apps," in *EuroSys*, 2015. Online: http://doi.acm.org/10.1145/2741948.2741974

[201] K. Petersen, M. Spreitzer, D. B. Terry, M. Theimer, and A. J. Demers, "Flexible update propagation for weakly consistent replication," in *SOSP*, 1997. Online: http://doi.acm.org/10.1145/268998.266711

[202] J. S. Plank, S. Simmerman, and C. D. Schuman, "Jerasure: A library in C/C++ facilitating erasure coding for storage applications - Version 1.2," University of Tennessee, Tech. Rep. CS-08-627, August 2008.

[203] M. R. Rahman, W. M. Golab, A. AuYoung, K. Keeton, and J. J. Wylie, "Toward a principled framework for benchmarking consistency," *Computing Research Repository*, vol. abs/1211.4290, 2012.

[204] J. Rao, E. J. Shekita, and S. Tata, "Using paxos to build a scalable, consistent, and highly available datastore," *VLDB*, vol. 4, no. 4, 2011.

[205] M. Raynal and A. Schiper, "A suite of definitions for consistency criteria in distributed shared memories," *Annales des Télécommunications*, vol. 52, no. 11-12, 1997. Online: http://dx.doi.org/10.1007/BF02997620

[206] B. Reed and F. P. Junqueira, "A simple totally ordered broadcast protocol," in *Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, 2008. Online: http://doi.acm.org/10.1145/1529974.1529978

[207] P. L. Reiher, J. S. Heidemann, D. Ratner, G. Skinner, and G. J. Popek, "Resolving file conflicts in the ficus file system," in *USENIX Summer 1994 Technical Conference, 1994*, 1994.

Online: https://www.usenix.org/conference/usenix-summer-1994-technical-conference/resolving-file-conflicts-ficus-file-system

[208] R. Rodrigues and B. Liskov, "High availability in dhts: Erasure coding vs. replication," in *IPTPS*, 2005. Online: http://dx.doi.org/10.1007/11558989_21

[209] H. Roh, M. Jeon, J. Kim, and J. Lee, "Replicated abstract data types: Building blocks for collaborative applications," *Journal of Parallel and Distributed Computing*, vol. 71, no. 3, 2011. Online: http://dx.doi.org/10.1016/j.jpdc.2010.12.006

[210] Y. Saito and M. Shapiro, "Optimistic replication," *ACM Computing Surveys*, vol. 37, no. 1, 2005.

[211] N. Santos, L. Veiga, and P. Ferreira, "Vector-field consistency for ad-hoc gaming," in *ACM/IFIP/USENIX Middleware Conference, 2007*, 2007.

[212] M. Serafini, D. Dobre, M. Majuntke, P. Bokor, and N. Suri, "Eventually Linearizable Shared Objects," in *PODC*, 2010.

[213] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski, "Convergent and commutative replicated data types," *Bulletin of the EATCS*, vol. 104, 2011.

[214] ——, "Conflict-free replicated data types," in *Stabilization, Safety, and Security of Distributed Systems (SSS), 2011*, 2011. Online: http://dx.doi.org/10.1007/978-3-642-24550-3_29

[215] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket, "Venus: verification for untrusted cloud storage," in *ACM Cloud Computing Security Workshop (CCSW)*, 2010.

[216] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," Google, Inc., Tech. Rep., 2010. Online: https://research.google.com/archive/papers/dapper-2010-1.pdf

[217] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis, "Zeno: Eventually consistent byzantine-fault tolerance," in *NSDI*, 2009. Online: http://www.usenix.org/events/nsdi09/tech/full_papers/singh/singh.pdf

[218] A. Singla, U. Ramachandran, and J. K. Hodgins, "Temporal notions of synchronization and consistency in beehive," in *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 1997. Online: http://doi.acm.org/10.1145/258492.258513

[219] K. C. Sivaramakrishnan, G. Kaki, and S. Jagannathan, "Declarative programming over eventually consistent data stores," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, 2015*, 2015. Online: http://doi.acm.org/10.1145/2737924.2737981

[220] SoftLayer, "IBM SoftLayer," 2016, online: http://www.softlayer.com/; visited September 2016.

[221] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: an extremely simple oblivious RAM protocol," in *ACM SIGSAC Conference on Computer and Communications Security, (CCS'13)*, 2013. Online: http://doi.acm.org/10.1145/2508859.2516660

[222] R. C. Steinke and G. J. Nutt, "A unified theory of shared memory consistency," *Journal of the ACM*, vol. 51, no. 5, 2004. Online: http://doi.acm.org/10.1145/1017460.1017464

[223] M. Stonebraker, "The case for shared nothing," *IEEE Database Eng. Bull.*, vol. 9, no. 1, 1986. Online: http://sites.computer.org/debull/86MAR-CD.pdf

[224] H. Sutter, "The free lunch is over: a fundamental turn toward concurrency in software," http://www.gotw.ca/publications/concurrency-ddj.htm, 2005.

[225] Syncany, "Secure file synchronization software for arbitrary storage backends," 2016, online: https://www.syncany.org/; visited September 2016.

[226] A. S. Tanenbaum and M. van Steen, *Distributed systems - principles and paradigms (2. ed.)*. Pearson Education, 2007.

[227] D. Terry, "Replicated data consistency explained through baseball," *Communications of the ACM (CACM)*, vol. 56, no. 12, 2013.

[228] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. B. Welch, "Session guarantees for weakly consistent replicated data," in *Parallel and Distributed Information Systems (PDIS)*, 1994.

[229] D. B. Terry, M. Theimer, K. Petersen, A. J. Demers, M. Spreitzer, and C. Hauser, "Managing update conflicts in bayou, a weakly connected replicated storage system," in *SOSP*, 1995. Online: http://doi.acm.org/10.1145/224056.224070

[230] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh, "Consistency-based service level agreements for cloud storage," in *SOSP*, 2013. Online: http://doi.acm.org/10.1145/2517349.2522731

[231] E. Torlak, M. Vaziri, and J. Dolby, "Memsat: checking axiomatic specifications of memory models," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010. Online: http://doi.acm.org/10.1145/1806596.1806635

[232] F. J. Torres-Rojas and E. Meneses, "Convergence through a weak consistency model: Timed causal consistency," *CLEI electronic journal*, vol. 8, no. 2, 2005.

[233] F. J. Torres-Rojas, M. Ahamad, and M. Raynal, "Timed consistency for shared distributed objects," in *PODC*, 1999.

[234] Twitter, "Distributed systems tracing with zipkin," https://blog.twitter.com/2012/distributed-systems-tracing-with-zipkin, 2012.

[235] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Veríssimo, "Efficient byzantine fault-tolerance," *IEEE Trans. Computers*, vol. 62, no. 1, 2013.

[236] VMware, "The Snowden Leak: A Windfall for Hybrid Cloud?" 2013, online: http://blogs.vmware.com/consulting/2013/09/the-snowden-leak-a-windfall-for-hybrid-cloud.html; visited September 2016.

[237] W. Vogels, "Eventually consistent," *Queue*, vol. 6, no. 6, Oct. 2008. Online: http://doi.acm.org/10.1145/1466443.1466448

[238] M. Vukolić, "The Byzantine empire in the intercloud," *SIGACT News*, vol. 41, no. 3, 2010. Online: http://doi.acm.org/10.1145/1855118.1855137

[239] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu, "Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective," in *Conference on Innovative Data Systems Research (CIDR), 2011*, 2011.

[240] Y. Wang, L. Alvisi, and M. Dahlin, "Gnothi: separating data and metadata for efficient and available storage replication," in *USENIX Annual Technical Conference, USENIX ATC*, 2012.

[241] H. Weatherspoon and J. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *IPTPS*, 2002.

[242] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *OSDI*, 2006. Online: http://www.usenix.org/events/osdi06/tech/weil.html

[243] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. E. Anderson, "Verdi: a framework for implementing and formally verifying distributed systems," in *ACM SIGPLAN PLDI*, 2015. Online: http://doi.acm.org/10.1145/2737924.2737958

[244] J. M. Wing and C. Gong, "Testing and verifying concurrent objects," *J. Parallel Distrib. Comput.*, vol. 17, no. 1-2, 1993. Online: http://dx.doi.org/10.1006/jpdc.1993.1015

[245] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha, "Spanstore: cost-effective geo-replicated storage spanning multiple cloud services," in *SOSP*, 2013.

[246] C. Xie, C. Su, C. Littley, L. Alvisi, M. Kapritsos, and Y. Wang, "High-performance ACID via modular concurrency control," in *SOSP*, 2015. Online: http://doi.acm.org/10.1145/2815400.2815430

[247] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind, "Nemos: A framework for axiomatic and executable specifications of memory consistency models," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2004. Online: http://dx.doi.org/10.1109/IPDPS.2004.1302944

[248] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for byzantine fault tolerant services," in *SOSP*, 2003. Online: http://doi.acm.org/10.1145/945445.945470

[249] H. Yu and A. Vahdat, "Design and evaluation of a conit-based continuous consistency model for replicated services," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 3, 2002.

[250] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. Jain, and M. Stumm, "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems," in *(OSDI)*, 2014. Online: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan

[251] Z3, "High-performance theorem prover," https://github.com/Z3Prover/z3; visited February 2016.

[252] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro, "Write fast, read in the past: Causal consistency for client-side applications." ACM/IFIP/USENIX Middleware Conference, 2015, Dec. 2015.

[253] H. Zeineddine and W. Bazzi, "Rationing data updates with consistency considerations in distributed systems," in *IEEE International Conference on Networks, ICON*, 2011. Online: http://dx.doi.org/10.1109/ICON.2011.6168469

[254] K. Zellag and B. Kemme, "Consistency anomalies in multi-tier architectures: automatic detection and prevention," *VLDB Journal*, vol. 23, no. 1, 2014. Online: http://dx.doi.org/10.1007/s00778-013-0318-x

[255] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Software Eng.*, vol. 28, no. 2, 2002. Online: http://dx.doi.org/10.1109/32.988498

[256] H. Zhang, M. Dong, and H. Chen, "Efficient and available in-memory kv-store with hybrid erasure coding and replication," in *USENIX Conference on File and Storage Technologies, FAST*, 2016. Online: https://www.usenix.org/conference/fast16/technical-sessions/presentation/zhang-heng

[257] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports, "Building consistent transactions with inconsistent replication," in *SOSP*, 2015. Online: http://doi.acm.org/10.1145/2815400.2815404

Appendix A

Summary of Consistency Predicates

Linearizability($\mathcal{F}$)	SingleOrder $\wedge$ RealTime $\wedge$ RVal($\mathcal{F}$)
SingleOrder	$\exists H' \subseteq \{op \in H : op.oval = \nabla\} : vis = ar \setminus (H' \times H)$
RealTime	$rb \subseteq ar$
Regular($\mathcal{F}$)	SingleOrder $\wedge$ RealTimeWrites $\wedge$ RVal($\mathcal{F}$)
Safe($\mathcal{F}$)	SingleOrder $\wedge$ RealTimeWrites $\wedge$ SeqRVal($\mathcal{F}$)
RealTimeWrites	$rb\|_{wr \to op} \subseteq ar$
SeqRVal($\mathcal{F}$)	$\forall op \in H : Concur(op) = \emptyset \Rightarrow op.oval \in \mathcal{F}(op, cxt(A, op))$
EventualConsistency($\mathcal{F}$)	EventualVisibility $\wedge$ NoCircularCausality $\wedge$ RVal($\mathcal{F}$)
EventualVisibility	$\forall a \in H, \forall [f] \in H/\approx_{ss}: \|\{b \in [f] : (a \xrightarrow{rb} b) \wedge (a \not\xrightarrow{vis} b)\}\| < \infty$
NoCircularCausality	$acyclic(hb)$
StrongConvergence	$\forall a, b \in H\|_{rd} : vis^{-1}(a)\|_{wr} = vis^{-1}(b)\|_{wr} \Rightarrow a.oval = b.oval$
StrongEventualCons.($\mathcal{F}$)	EventualConsistency($\mathcal{F}$) $\wedge$ StrongConvergence
QuiescentConsistency($\mathcal{F}$)	$\|H\|_{wr}\| < \infty \Rightarrow \exists C \in \mathcal{C} : \forall [f] \in H/\approx_{ss}: \|\{op \in [f] : op.oval \notin \mathcal{F}(op, C)\}\| < \infty$
PRAM	$so \subseteq vis$
SequentialConsistency($\mathcal{F}$)	SingleOrder $\wedge$ PRAMConsistency $\wedge$ RVal($\mathcal{F}$)
MonotonicReads	$\forall a \in H, \forall b, c \in H\|_{rd} : a \xrightarrow{\text{vis}} b \wedge b \xrightarrow{\text{so}} c \Rightarrow a \xrightarrow{\text{vis}} c \triangleq (vis; so\|_{rd \to rd}) \subseteq vis$
ReadMyWrites	$\forall a \in H\|_{wr}, \forall b \in H\|_{rd} : a \xrightarrow{\text{so}} b \Rightarrow a \xrightarrow{\text{vis}} b \triangleq so\|_{wr \to rd} \subseteq vis$
MonotonicWrites	$\forall a, b \in H\|_{wr} : a \xrightarrow{\text{so}} b \Rightarrow a \xrightarrow{\text{vis}} b \triangleq so\|_{wr \to wr} \subseteq vis$

WritesFollowReads	$\forall a, c \in H\|_{wr}, \forall b \in H\|_{rd} : a \xrightarrow{\text{vis}} b \wedge b \xrightarrow{\text{so}} c \Rightarrow a \xrightarrow{\text{ar}} c \triangleq (vis; so\|_{rd \to wr}) \subseteq ar$
CausalVisibility	$hb \subseteq vis$
CausalArbitration	$hb \subseteq ar$
Causality($\mathcal{F}$)	CausalVisibility $\wedge$ CausalArbitration $\wedge$ RVal($\mathcal{F}$)
Causal+($\mathcal{F}$)	Causality($\mathcal{F}$) $\wedge$ StrongConvergence
RealTimeCausality($\mathcal{F}$)	Causality($\mathcal{F}$) $\wedge$ RealTime
TimedVisibility(Δ)	$\forall a \in H\|_{wr}, \forall b \in H, \forall t \in Time : a.rtime = t \wedge b.stime \geq t + \Delta \Rightarrow a \xrightarrow{\text{vis}} b$
TimedCausality($\mathcal{F}, \Delta$)	Causality($\mathcal{F}$) $\wedge$ TimedVisibility(Δ)
TimedLinearizability($\mathcal{F}, \Delta$)	SingleOrder $\wedge$ TimedVisibility(Δ) $\wedge$ RVal($\mathcal{F}$)
PrefixSequential($\mathcal{F}$)	SingleOrder $\wedge$ MonotonicWrites $\wedge$ RVal($\mathcal{F}$)
PrefixLinearizable($\mathcal{F}$)	SingleOrder $\wedge$ RealTimeWW $\wedge$ RVal($\mathcal{F}$)
RealTimeWW	$rb\|_{wr \to wr} \subseteq ar$
K-Linearizable($\mathcal{F}, K$)	SingleOrder $\wedge$ RealTimeWW $\wedge$ K-RealTimeReads(K) $\wedge$ RVal($\mathcal{F}$)
K-RealTimeReads(K)	$\forall a \in H\|_{wr}, \forall b \in H\|_{rd}, \forall PW \subseteq H\|_{wr}, \forall pw \in PW : \|PW\| < K \wedge a \xrightarrow{ar} pw \wedge pw \xrightarrow{rb} b \wedge a \xrightarrow{rb} b \Rightarrow a \xrightarrow{ar} b$
ForkLinearizability($\mathcal{F}$)	PRAM $\wedge$ RealTime $\wedge$ NoJoin $\wedge$ RVal($\mathcal{F}$)
NoJoin	$\forall a_i, b_i, a_j, b_j \in H : a_i \not\approx_{ss} a_j \wedge (a_i, a_j) \in ar \setminus vis \wedge a_i \preceq_{so} b_i \wedge a_j \preceq_{so} b_j \Rightarrow (b_i, b_j), (b_j, b_i) \notin vis$
Fork*($\mathcal{F}$)	ReadMyWrites $\wedge$ RealTime $\wedge$ AtMostOneJoin $\wedge$ RVal($\mathcal{F}$)
AtMostOneJoin	$\forall a_i, a_j \in H : a_i \not\approx_{ss} a_j \wedge (a_i, a_j) \in ar \setminus vis \Rightarrow \|\{b_i \in H : a_i \preceq_{so} b_i \wedge (\exists b_j \in H : a_j \preceq_{so} b_j \wedge b_i \xrightarrow{vis} b_j\}\| \leq 1 \wedge \|\{b_j \in H : a_j \preceq_{so} b_j \wedge (\exists b_i \in H : a_i \preceq_{so} b_i \wedge b_j \xrightarrow{vis} b_i\}\| \leq 1$
ForkSequential($\mathcal{F}$)	PRAM $\wedge$ NoJoin $\wedge$ RVal($\mathcal{F}$)
WeakForkLin($\mathcal{F}$)	PRAM $\wedge$ K-RealTime(2) $\wedge$ AtMostOneJoin $\wedge$ RVal($\mathcal{F}$)
PerObjectPRAM	$(so \cap ob) \subseteq vis$
PerObjectSingleOrder	$\exists H' \subseteq \{op \in H : op.oval = \nabla\} : ar \cap ob = vis \cap ob \setminus (H' \times H)$
PerObjectSequential($\mathcal{F}$)	PerObjectSingleOrder $\wedge$ PerObjectPRAM $\wedge$ RVal($\mathcal{F}$)
ProcessorConsistency($\mathcal{F}$)	PerObjectSingleOrder $\wedge$ PRAM $\wedge$ RVal($\mathcal{F}$)
PerObjectHappensBefore	$hbo \triangleq ((so \cap ob) \cup vis)^+$

Table A.1 – Summary of consistency predicates formulated in Chapter 2.

Appendix B

Proofs of Strength Relations between Consistency Semantics

In this section, we prove some of the strength relations highlighted in Fig 2.1, using the formal definitions provided in Chapter 2 and listed, for convenience, in Appendix A. All proofs we provide hold regardless of the replicated data type implemented by the storage system. Specific semantics referring to the implemented replicated data type are enclosed in the return value consistency term (Eq. 2.4).

Preliminaries In the following, let H be an history and $A = (H, ar, vis)$ a corresponding abstract execution. Unless stated otherwise, we assume that all operations complete. Furthermore, each process can have at most one pending operation at any given time.

Proposition B.1. *Linearizability* $>$ *SequentialConsistency*

Proof. We show that (SingleOrder $\wedge$ RealTime) $\Rightarrow$ PRAM. Let $o, o' \in H$ and $o \xrightarrow{\text{so}} o'$. Then, by definition of so, $o \xrightarrow{\text{rb}} o'$. By RealTime, this implies $o \xrightarrow{\text{ar}} o'$. Finally, by SingleOrder, $o \xrightarrow{\text{vis}} o'$.
Furthermore, it follows trivially from the definitions of PRAM, SingleOrder and RealTime that PRAM $\not\Rightarrow$ RealTime, PRAM $\not\Rightarrow$ SingleOrder and PRAM $\not\Rightarrow$ (SingleOrder $\wedge$ RealTime). It follows that (SingleOrder $\wedge$ PRAM) $\not\Rightarrow$ (SingleOrder $\wedge$ RealTime). □

Proposition B.2. *Linearizability* $>$ *Regular*

Proof. It follows from RealTime $>$ RealTimeWrites, since $rb|_{wr \to op} \subseteq rb \subseteq ar$, and by the definition of rb. □

Proposition B.3. *Regular* $>$ *Safe*

Proof. It follows trivially from RVal($\mathcal{F}$) > SeqRVal($\mathcal{F}$). □

Proposition B.4. *SequentialConsistency > Causality*

Proof. First, we proceed to show that (SingleOrder ∧ PRAM) ⇒ CausalArbitration. By SingleOrder and PRAM we have that $vis \subseteq ar$ and $so \subseteq vis$. Thus, $hb = (so \cup vis)^+ \subseteq (vis \cup vis)^+ = vis^+ \subseteq ar^+ = ar$.

Now we prove that (SingleOrder ∧ PRAM) ⇒ CausalVisibility. As in the previous case, we find $hb \subseteq vis^+$. It remains to show that $vis^+ \subseteq vis$. Let $a, b, c \in H$, such that $a \xrightarrow{\text{vis}} b \xrightarrow{\text{vis}} c$. Then, by SingleOrder, $a \xrightarrow{\text{ar}} b \xrightarrow{\text{ar}} c$. Since ar is transitive, $a \xrightarrow{\text{ar}} c$. Thus, by SingleOrder, $a \xrightarrow{\text{vis}} c$.

It is easy to show that (CausalVisibility ∧ CausalArbitration) $\not\Longrightarrow$ (SingleOrder ∧ PRAM) from the definitions of the predicates in question. □

Proposition B.5. *Causality > WritesFollowReads*

Proof. Let $a, b, c \in H$ such that $a \xrightarrow{\text{vis}} b \xrightarrow{\text{so}} c$. Then, by CausalArbitration, $a \xrightarrow{\text{ar}} c$. □

Proposition B.6. *PRAM > ReadMyWrites*

Proof. It follows from the definition of so and from PRAM that $so|_{wr \to rd} \subseteq so \subseteq vis$. □

Proposition B.7. *PRAM > MonotonicWrites*

Proof. It follows from the definition of so and from PRAM that $so|_{wr \to wr} \subseteq so \subseteq vis$. □

Proposition B.8. *Safe > ReadMyWrites*

Proof. Let $a, b, c \in H$ such that $a \xrightarrow{\text{so}} b \xrightarrow{\text{vis}} c$. Then, by definition of so, $a \xrightarrow{\text{rb}} b \xrightarrow{\text{vis}} c$. By RealTime, $a \xrightarrow{\text{ar}} b \xrightarrow{\text{vis}} c$. Finally, by SingleOrder and by transitivity of ar, $a \xrightarrow{\text{vis}} c$. □

Proposition B.9. *Causality > PRAM*

Proof. Causality ⇒ PRAM follows from the definition of CausalVisibility, namely: $so \subseteq (so \cup vis)^+ \subseteq vis$. PRAM $\not\Longrightarrow$ CausalArbitration follows trivially from the definitions of the predicates in question. □

Proposition B.10. *Fork* > ReadMyWrites*

Proof. Given that the Fork* predicate includes ReadMyWrites, to prove that it is strictly stronger than ReadMyWrites we have to show that ReadMyWrites does not imply the other terms of its predicate. Formally: ReadMyWrites $\not\Longrightarrow$ RealTime ∧ AtMostOneJoin, which trivially follows from the predicates in question. □

Appendix C

Consistency Semantics and Implementations

Models	Definitions	Implementations [1]
Atomicity	Lamport [164]	Attiya et al. [29]
Bounded fork-join causal	Mahajan et al. [185]	-
Bounded staleness	Mahajan et al. [184]	-
Causal	Lamport [160], Hutto and Ahamad [143], Ahamad et al. [11], Mahajan et al. [185]	Ladin et al. [156], Birman et al. [56], Lakshmanan et al. [158], Lloyd et al. [178], Du et al. [101], Zawirski et al. [252], Lesani et al. [168]
Causal+	Lloyd et al. [177]	Petersen et al. [201], Belaramani et al. [44], Almeida et al. [14]
Coherence	Dubois et al. [103]	-
Conit	Yu and Vahdat [249]	-
Γ-atomicity	Golab et al. [123]	-
Δ-atomicity	Golab et al. [122]	-
Delta	Singla et al. [218]	-
Entry	Bershad and Zekauskas [51]	-

[1] In case of very popular consistency semantics (e.g., causal consistency, atomicity/linearizability), we only cite a subset of known implementations.

Eventual	Terry et al. [228], Vogels [237]	Reiher et al. [207], DeCandia et al. [96], Singh et al. [217], Bortnikov et al. [57], Bronson et al. [62]
Eventual linearizability	Serafini et al. [212]	-
Eventual serializability	Fekete et al. [109]	-
Fork*	Li and Mazières [173]	Feldman et al. [110]
Fork	Mazières and Shasha [189], Cachin et al. [70]	Li et al. [174], Brandenburger et al. [59]
Fork-join causal	Mahajan et al. [184]	-
Fork-sequential	Oprea and Reiter [196]	-
Hybrid	Attiya and Friedman [27]	-
K-atomic	Aiyer et al. [13]	-
K-regular	Aiyer et al. [13]	-
K-safe	Aiyer et al. [13]	-
k-staleness	Bailis et al. [35]	-
Lazy release	Keleher et al. [150]	-
Linearizability	Herlihy and Wing [138]	Burrows [69], Baker et al. [40], Glendenning et al. [121], Calder et al. [75], Corbett et al. [92], Han et al. [133], Lee et al. [167]
Location	Gao and Sarkar [115]	-
Monotonic reads	Terry et al. [228]	Terry et al. [229]
Monotonic writes	Terry et al. [228]	Terry et al. [229]
Observable causal	Attiya et al. [30]	-
PBS $\langle k, t \rangle$-staleness	Bailis et al. [35]	-
Per-object causal	Burckhardt et al. [68]	-
Per-record timeline	Cooper et al. [90], Lloyd et al. [177]	Andersen et al. [22]
PRAM	Lipton and Sandberg [175]	-
Prefix	Terry et al. [229], Terry [227]	-
Processor	Goodman [124]	-
Quiescent	Herlihy and Shavit [137]	-
Rationing	Kraska et al. [154]	-
Read-my-writes	Terry et al. [228]	Terry et al. [229]

Real-time causal	Mahajan et al. [185]	-
RedBlue	Li et al. [170]	-
Regular	Lamport [164]	Malkhi and Reiter [188], Guerraoui and Vukolic [127]
Release	Gharachorloo et al. [116]	-
Safe	Lamport [164]	Malkhi and Reiter [187], Guerraoui and Vukolic [127]
Scope	Iftode et al. [145]	-
Sequential	Lamport [161]	Rao et al. [204]
Slow	Hutto and Ahamad [143]	-
Strong eventual	Shapiro et al. [214]	Shapiro et al. [213], Conway et al. [89], Roh et al. [209]
Timed causal	Torres-Rojas and Meneses [232]	-
Timed serial	Torres-Rojas et al. [233]	-
Timeline	Cooper et al. [90]	Rao et al. [204]
Tunable	Krishnamurthy et al. [155]	Lakshman and Malik [157], Wu et al. [245], Perkins et al. [200], Sivaramakrishnan et al. [219]
t-visibility	Bailis et al. [35]	-
Vector-field	Santos et al. [211]	-
Weak	Vogels [237], Bermbach and Kuhlenkamp [46]	-
Weak fork-linearizability	Cachin et al. [73]	Shraer et al. [215]
Weak ordering	Dubois et al. [103]	-
Writes-follow-reads	Terry et al. [228]	Terry et al. [229]

Table C.1 – Definitions of consistency semantics and their implementations in research literature.

Appendix D

Hybris: Proofs and Algorithms

This appendix presents pseudocode and correctness proofs for the core parts of the Hybris protocol as described in Section 3.3. [1] In particular, we prove that Algorithm 3, satisfies linearizability, and wait-freedom (resp. finite-write termination) for PUT (resp. for GET) operations. [2] The linearizable functionality of RMDS is specified in Alg. 1, while Alg. 2 describes the simple API required from cloud stores.

[1] For simplicity, in the pseudocode we omit the *container* parameter which can be passed as argument to Hybris APIs. Furthermore, the algorithms here presented refer to the replicated version of Hybris. The version supporting erasure codes does not entail any significant modification to algorithms and related proofs.

[2] For the sake of readability, in the proofs we ignore possible DELETE operations. However, it is easy to modify the proofs to account for their effects.

D.1 Hybris protocol

Algorithm 1 RMDS functionality (linearizable).

Server state variables:
  $md \subseteq K \times TSMD$, initially $\perp$, read and written through $mdf : K \to TSMD$
  $sub \subseteq K \times (\mathbb{N}_0 \times \ldots \times \mathbb{N}_0)$, initially $\perp$, read and written through $subf : K \to (\mathbb{N}_0 \times \ldots \times \mathbb{N}_0)$

operation CONDUPDATE $(k, ts, cList, hash, size)$
  $(ts_k, -, -, -) \leftarrow mdf(k)$
  if $ts_k = \perp$ **or** $ts > ts_k$ **then**
    $mdf : k \leftarrow (ts, cList, hash, size)$
    send **notify**(k, ts) to every $cid \in subf(k)$
    $subf : k \leftarrow \emptyset$
  return OK

operation READ $(k, subscribe)$ by cid
  if $subscribe$ **then**
    $subf : k \leftarrow subf(k) \cup \{cid\}$
  return $mdf(k)$

operation LIST ()
  return $mdf(*)$

Algorithm 2 Cloud store C_i functionality.

Server state variables:
  $data \subseteq K \times V$, initially $\emptyset$, read and written through $f : K \to V$

operation *put*(key, val)
  $f : key \leftarrow value$
  return OK

operation *get*(key)
  return $f(key)$

Algorithm 3 Algorithm of Hybris client *cid*.

Types:
  $TS = (\mathbb{N}_0 \times \mathbb{N}_0) \cup \{\perp\}$, with fields *sn* and *cid* // timestamps
  $TSMD = (TS \times (C_i \times \ldots \times C_i) \times H(V) \times \mathbb{N}_0) \cup \{\perp\}$, with fields *ts*, *replicas*, *hash* and *size*

Shared objects:
  RMDS: $MWMR$ linearizable wait-free timestamped storage object, implementing Alg. 1
  $C_{0\ldots n}$: cloud stores, exposing key-value API as in Alg. 2

Client state variables:
  $ts \in TS$, initially $(0, \perp)$
  $cloudList \in \{C_i \times \ldots \times C_i\} \cup \{\perp\}$, initially $\emptyset$

operation PUT (k, v)
  $(ts, -, -, -) \leftarrow RMDS.\text{READ}(k, false)$
  if $ts = \perp$ **then** $ts \leftarrow (0, cid)$
  $ts \leftarrow (ts.sn + 1, cid)$
  $cloudList \leftarrow \emptyset$
  trigger $timer$
  forall $f + 1$ selected clouds C_i **do**
    C_i.*put*$(k|ts, v)$
  wait until $|cloudList| = f + 1$ **or** $timer$ **expires**
  if $|cloudList| < f + 1$ **then**
    forall f secondary clouds C_i **do**
      C_i.*put*$(k|ts, v)$
    wait until $|cloudList| = f + 1$
  $RMDS.\text{CONDUPDATE}(k, ts, cloudList, H(v), size(v))$
  trigger garbage collection // see Section 3.3.4
  return OK

upon *put*$(k|ts, v)$ completes at cloud C_i
  $cloudList \leftarrow cloudList \cup \{C_i\}$

operation GET (k) // worst-case, Section 3.3.5 code only
  $(ts, cloudList, hash, size) \leftarrow RMDS.\text{READ}(k, true)$
  if $ts = \perp$ **or** $cloudList = \perp$ **then return** $\perp$
  forall $C_i \in cloudList$ **do**
    C_i.*get*$(k|ts)$

upon *get*$(k|ts)$ returns $data$ from cloud C_i
  if $H(data) = hash$ **then return** $data$
  else C_i.*get*$(k|ts)$

upon received *notify*(k, ts') from RMDS such that $ts' > ts$
  cancel all pending *get*
  return GET (k)

operation LIST ()
  $mdList \leftarrow RMDS.\text{LIST}()$
  forall $md \in mdList$ **do**
    if $md.cloudList = \perp$ **then**
      $mdList \leftarrow mdList \setminus \{md\}$
  return $mdList$

operation DELETE (k)
  $(ts, cloudList, -, -) \leftarrow RMDS.\text{READ}(k, false)$
  if $ts = \perp$ **or** $cloudList = \perp$ **then return** OK
  $ts \leftarrow (ts.sn + 1, cid)$
  $RMDS.\text{CONDUPDATE}(k, ts, \perp, \perp, 0)$
  trigger garbage collection // see Section 3.3.4
  return OK

D.2 Correctness proofs

Preliminaries We define the timestamp of operation o, denoted $ts(o)$, as follows. If o is a PUT, then $ts(o)$ is the value of client's variable ts when its assignment completes at line 36, Alg. 3. Else, if o is a GET, then $ts(o)$ equals the value of ts when client executes line 57, Alg. 3 (i.e., when GET returns). We further say that an operation o *precedes* operation o', if o completes before o' is invoked. Without loss of generality, we assume that all operations access the same key k.

Lemma D.2.1 (Partial Order). *Let o and o' be two GET or PUT operations with timestamps $ts(o)$ and $ts(o')$, respectively, such that o precedes o'. Then $ts(o) \leq ts(o')$, and if o' is a PUT then $ts(o) < ts(o')$.*

Proof. In the following, prefix $o.RMDS$ denotes calls to RMDS within operation o (and similarly for o'). Let o' be a PUT (resp. GET) operation.

Case 1 (o is a PUT): then $o.RMDS.\text{CONDUPDATE}(o.md)$ at line 46, Alg. 3, precedes (all possible calls to) $o'.RMDS.\text{READ}()$ at line 52, Alg. 3 (resp., line 34, Alg. 3). By linearizability of RMDS (and RMDS functionality in Alg. 1) and definition of operation timestamps, it follows that $ts(o') \geq ts(o)$. Moreover, if o' is a PUT, then $ts(o') > ts(o)$ because $ts(o')$ is obtained from incrementing the timestamp ts returned by $o'.RMDS.\text{READ}()$ at line 34, Alg. 3, where $ts \geq ts(o)$.

Case 2 (o is a GET): then since all possible calls to $o'.RMDS.\text{READ}()$ at line 52 (resp. 34) follow the latest call of $o.RMDS.\text{READ}()$ in line 52, by Alg. 1 and by linearizability of RMDS, it follows that $ts(o') \geq ts(o)$. If o' is a PUT, then $ts(o') > ts(o)$, similarly to Case 1. □

Lemma D.2.2 (Unique PUTS). *If o and o' are two PUT operations, then $ts(o) \neq ts(o')$.*

Proof. By lines 34-36, Alg. 3, RMDS functionality (Alg. 1) and the fact that a given client does not invoke concurrent operations on the same key. □

Lemma D.2.3 (Integrity). *Let rd be a GET(k) operation returning value $v \neq \perp$. Then there exists a single PUT operation wr of the form PUT(k, v) such that $ts(rd) = ts(wr)$.*

Proof. Since rd returns v and has a timestamp $ts(rd)$, rd receives v in response to *get*$(k|ts(rd))$ from some cloud C_i. Suppose for the purpose of contradiction that v is never written by a PUT. Then, by the collision resistance of $H()$, the check at line 57 does not pass and rd does not return v. Therefore, we conclude that some operation wr issues *put* $(k|ts(rd))$ to C_i in line 40. Hence, $ts(wr) = ts(rd)$. Finally, by Lemma D.2.2 no other PUT has the same timestamp. □

Theorem D.2.4 (Atomicity). *Every execution ex of Algorithm 3 satisfies linearizability.*

Proof. Let ex be an execution of Algorithm 3. By Lemma D.2.3 the timestamp of a GET either has been written by some PUT or the GET returns $\perp$. With this in mind, we first construct ex' from ex by completing all PUT operations of the form PUT (k, v), where v has been returned by some complete GET operation. Then we construct a sequential permutation π by ordering all operations in ex', except GET operations that return $\perp$, according to their timestamps and by placing all GET operations that did not return $\perp$ immediately after the PUT operation with the same timestamp. The GET operations that did return $\perp$ are placed in the beginning of π.

Towards linearizability, we show that a GET rd in π always returns the value v written by the latest preceding PUT which appears before it in π, or the initial value of the register $\perp$ if there is no such PUT. In the latter case, by construction rd is ordered before any PUT in π. Otherwise, $v \neq \perp$ and by Lemma D.2.3 there is a PUT (k, v) operation, with the same timestamp, $ts(rd)$. In this case, PUT (k, v) appears before rd in π, by construction. By Lemma D.2.2, other PUT operations in π have a different timestamp and hence appear in π either before PUT (k, v) or after rd.

It remains to show that π preserves real-time order. Consider two complete operations o and o' in ex' such that o precedes o'. By Lemma D.2.1, $ts(o') \geq ts(o)$. If $ts(o') > ts(o)$ then o' appears after o in π by construction. Otherwise $ts(o') = ts(o)$ and by Lemma D.2.1 it follows that o' is a GET. If o is a PUT, then o' appears after o since we placed each read after the PUT with the same timestamp. Otherwise, if o is a GET, then it appears before o' as in ex'. □

Theorem D.2.5 (Availability). *Hybris PUT calls are wait-free, whereas Hybris GET calls are finite-write terminating.*

Proof. The wait freedom of Hybris PUT operations follows from: a) the assumption of using $2f + 1$ clouds out of which at most f may be faulty (and hence the *wait* statement at line 45, Alg. 3 is non-blocking), and b) wait-freedom of calls to RMDS (hence, calls to RMDS at lines 34 and 46, Alg. 3 return).

We prove finite-write termination of GET by contradiction. Assume there is a finite number of writes to key k in execution ex, yet that there is a GET(k) operation rd by a correct client that never completes. Let W be the set of all PUT operations in ex, and let wr be the PUT operation with maximum timestamp ts_{max} in W that completes the call to RMDS at line 46, Alg. 3. We distinguish two cases: (i) rd invokes an infinite number of recursive GET calls (in line 61, Alg 3), and (ii) rd never passes the check at line 57, Alg. 3.

In case (i), there is a recursive GET call in rd, invoked after wr completes conditional update to RMDS. In this GET call, the client does not execute line 61, Alg 3, by definition of wr and specification of $RMDS$.CONDUPDATE in Alg. 1 (as there is no *notify* for a $ts > ts_{max}$). A contradiction.

In case (ii), notice that key $k|ts_{max}$ is never garbage collected at $f + 1$ clouds that constitute $cloudList$ at line 46, Alg. 3 in wr. Since rd does not terminate, it receives a notification at

line 59, Alg. 3 with timestamp ts_{max} and reiterates GET. In this iteration of GET, the timestamp of rd is ts_{max}. As $cloudList$ contains $f + 1$ clouds, including at least one correct cloud C_i, and as C_i is eventually consistent, C_i eventually returns value v written by wr to a *get* call. This value v passes the hash check at line 57, Alg. 3 and rd completes. A contradiction. □

D.3 Alternative proof of Hybris linearizability

In this section, we prove the linearizability of the Hybris protocol (§ D.1) using the axiomatic framework we introduced in Chapter 2.

Preliminaries We define the timestamp of operation o, denoted $ts(o)$, as follows. If o is a PUT, then $ts(o)$ is the value of client's variable ts when its assignment completes at line 36, Alg. 3. Else, if o is a GET, then $ts(o)$ equals the value of ts when client executes line 57, Alg. 3 (i.e., when GET returns). Without loss of generality, we assume that all operations access the same key k.

Definition D.3.0.1 (Same-timestamp equivalence relation)**.** *Let st be an equivalence relation on H that groups pairs of operations having the same timestamp. Formally: $st \triangleq \{(a,b) : a,b \in H \wedge ts(a) = ts(b)\}$.*

Lemma D.3.1 (Partial order tso)**.** *Let o and o' be two GET or PUT operations with timestamps $ts(o)$ and $ts(o')$, respectively, such that $o \xrightarrow{rb} o'$. Then there exists a partial order $tso \triangleq ar \setminus st$ induced by timestamps such that: if o' is a PUT then $o \xrightarrow{tso} o'$; otherwise $(o,o') \in st \cup tso$.*

Proof. In the following, prefix $o.RMDS$ denotes calls to RMDS within operation o (and similarly for o'). Let o' be a PUT (resp. GET) operation.

Case 1 (o is a PUT): then $o.RMDS.\text{CONDUPDATE}(o.md)$ at line 46, Alg. 3, precedes (all possible calls to) $o'.RMDS.\text{READ}()$ at line 52, Alg. 3 (resp., line 34, Alg. 3). By linearizability of RMDS (and RMDS functionality in Alg. 1) and definition of operation timestamps, it follows that $ts(o') \geq ts(o)$. Moreover, if o' is a PUT, then $o \xrightarrow{tso} o'$, because $ts(o')$ is obtained from incrementing the timestamp ts returned by $o'.RMDS.\text{READ}()$ at line 34, Alg. 3, where $ts \geq ts(o)$.

Case 2 (o is a GET): then since all possible calls to $o'.RMDS.\text{READ}()$ at line 52 (resp. 34) follow the latest call of $o.RMDS.\text{READ}()$ in line 52, by Alg. 1 and by linearizability of RMDS, it follows that $ts(o') \geq ts(o)$. If o' is a PUT, then $o \xrightarrow{tso} o'$, similarly to Case 1. □

Corollary D.3.1.1. *No two operations ordered by the returns-before partial order have strictly decreasing timestamps. Formally: $\nexists a,b \in H : a \xrightarrow{rb} b \wedge b \xrightarrow{tso} a \Leftrightarrow rb \subseteq st \cup tso$.*

Lemma D.3.2 (Unique timestamps of PUTs)**.** *If o and o' are two PUT operations, then $(o,o') \notin st$.*

Proof. By lines 34-36, Alg. 3, RMDS functionality (Alg. 1) and the fact that a given client does not invoke concurrent operations on the same key. □

Corollary D.3.2.1. *tso is a total order over PUT operations.*

Lemma D.3.3 (Integrity)**.** *Let rd be a GET(k) operation returning value $v \neq \perp$. Then, there exists a single PUT operation wr of the form PUT(k,v) such that $rd \approx_{st} wr$.*

Proof. Since rd returns v and has a timestamp $ts(rd)$, rd receives v in response to *get*$(k|ts(rd))$ from some cloud C_i. Suppose for the purpose of contradiction that v is never written by a PUT. Then, by the collision resistance of the hash function $H()$, the check at line 57 does not pass and rd does not return v. Therefore, we conclude that some operation wr issues *put* $(k|ts(rd))$ to C_i in line 40. Hence, $rd \approx_{st} wr$. Finally, by Lemma D.3.2 no other PUT has the same timestamp. □

Lemma D.3.4. *No two operations a and b non overlapping in real time, and having the same timestamp are arbitrated in a different order with respect to rb. Formally: $\nexists a, b \in H : a \xrightarrow{rb} b \wedge a \approx_{st} b \wedge b \xrightarrow{ar} a \Leftrightarrow (rb \cap st) \setminus ar = \emptyset$.*

Proof. By Lemmas D.3.1 and D.3.2 a and b can only comply with one the following cases:
Case 1: a is a PUT, b is a GET. By Lemma D.3.3, $a.ival = b.oval$. Moreover, $a.RMDS$.CONDUPDATE$(a.md)$ at line 46, Alg. 3, precedes (all possible calls to) $b.RMDS$.READ$()$ at line 52, Alg. 3. By linearizability of RMDS (and RMDS functionality in Alg. 1), it follows that $a \xrightarrow{ar} b$.
Case 2: a and b are both GETs. All possible calls to $a.RMDS$.READ$()$ at line 52, Alg. 3 precede all possible calls to the same API within operation b. By linearizability of RMDS (and RMDS functionality in Alg. 1), it follows that $a \xrightarrow{ar} b$. □

Lemma D.3.5 (REALTIME)**.** *Arbitration total order complies with returns-before partial order. Formally: $rb \subseteq ar$.*

Proof. It follows from Corollary D.3.1.1 and Lemma D.3.4. □

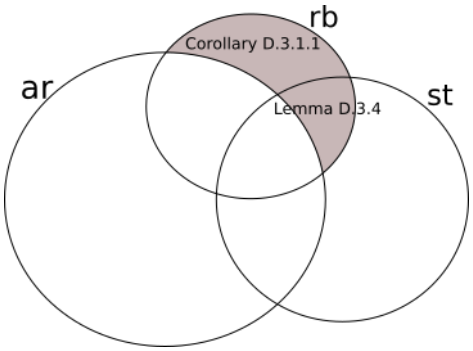


Figure D.1 – A set-based representation of Lemma D.3.5.

Lemma D.3.6 (Replicated register SINGLEORDER)**.** *Every read operation returns the last written value according to the arbitration order. Formally: $vis = ar \wedge RVAL(\mathcal{F}_{reg})$.*

Proof. Let ex be an execution of Algorithm 3. By Lemma D.3.3 the timestamp of a GET either has been written by some PUT or the GET returns $\perp$. With this in mind, we first construct ex' from ex by completing all PUT operations of the form PUT (k, v), where v has been returned by some complete GET operation. Then we construct a sequential permutation π by ordering all operations in ex', except GET operations that return $\perp$, according to some arbitration order $ar \supseteq tso$. The GET operations that return $\perp$ are placed at the beginning of π.

We show that a GET r in π always returns the value v written by the latest preceding PUT which appears before it in π (i.e., $\forall r \in H|_{rd} \wedge r.oval \neq \perp :!\exists w \in H|_{wr} \wedge w \xrightarrow{vis} r \wedge w.ival = r.oval \Rightarrow w = prec_{ar}(r)$) or the initial value of the register $\perp$ if there is no such PUT. In the latter case, by construction r is ordered before any PUT in π. Otherwise, $r.oval \neq \perp$ and by Lemma D.3.3 there is a PUT (k, v) operation, with the same timestamp, $ts(r)$. In this case, PUT (k, v) appears before r in π, by construction. By Lemma D.3.2, other PUT operations in π have different timestamps and hence appear in π either before PUT (k, v) or after r.

It remains to show that the converse proposition holds, i.e., formally: $\forall w, r \in H : r.oval \neq \perp \wedge w = prec_{ar}(r) \Rightarrow w \xrightarrow{vis} r \wedge w.ival = r.oval$. Suppose, for the purpose of contradiction, that $w.ival \neq r.oval$. Then, by Lemma D.3.3 there exists another PUT w_1 such that $w_1.ival = r.oval$. By construction of π $w_1 = prec_{ar}(r)$, and by hypothesis $w = prec_{ar}(r)$, thus $w_1 = w$: a contradiction. □

Theorem D.3.7 (Linearizability)**.** *Every execution ex of Algorithm 3 resulting in a history H satisfies linearizability. Formally: $H \models$ SINGLEORDER $\wedge$ REALTIME $\wedge$ RVAL$(\mathcal{F}_{reg})$.*

Proof. It follows from Lemmas D.3.5 and D.3.6. □

Annexe E

French Summary

La cohérence dans les systèmes de stockage distribués : des principes à l'application au stockage dans le nuage

E.1 La cohérence dans les systèmes de stockage répartis non transactionnels

Au cours des années, le mot "cohérence" a connu différentes définitions dans les domaines des systèmes distribués et des bases de données. Alors que dans les années 80, la cohérence signifiait généralement forte cohérence, plus tard défini aussi comme linéarisation, ces dernières années, avec l'avènement de systèmes hautement disponibles et évolutifs, la notion de cohérence a été à la fois affaiblie et floue. De plus, en dépit de sa pertinence dans le contexte des systèmes concurrents et distribués, le concept de cohérence a manqué historiquement d'un cadre de référence pour décrire ses aspects dans les communautés de chercheurs et de professionnels.

Dans le passé, certains efforts conjoints entre la recherche et l'industrie ont permis de formaliser, de comparer et même de standardiser les sémantiques *transactionnelles* [21, 126, 8]. Cependant, ces travaux ne tiennent pas compte des progrès de la dernière décennie de la recherche sur les bases de données, et ils ne considèrent pas la sémantique *non-transactionnelle*.

Récemment, la cohérence non transactionnelle a connu une reprise en raison de la popularité croissante des systèmes NoSQL. Par conséquent, de nouveaux modèles ont été conçus pour tenir compte de diverses combinaisons de problèmes de tolérance de panne et d'invariants d'application. Les chercheurs se sont efforcés de formuler les exigences minimales en termes

d'exactitude et, par conséquent, de coordination, pour permettre la conception de systèmes distribués rapides et fonctionnels [34, 30]. En outre, une tendance de recherche continue et passionnante a abordé cette question en s'appuyant sur différents outils et couches, en fonction des langages de programmation [16] dans les structures de données [213] et les correcteurs statiques au niveau de l'application [219, 125].

En tant que première contribution de cette thèse, nous proposons une étude de principe sur la sémantique de cohérence non-transactionnelle. Nous basons notre étude sur le modèle mathématique pour définir la sémantique de cohérence fournie dans [65], que nous avons étendue et raffinée. Ce modèle permet la définition de la sémantique de cohérence déclarative et composable, qui peut être exprimée en termes de prédicats logiques de premier ordre sur des entités graphiques qui, à leur tour, décrivent la *visibilité* et l'*ordre* d'événements. La table E.1 présente les entités les plus importantes de ce modèle, qui sont expliquées aussi dans le Chapitre 2.

Entité	Description
Operation (op)	Single operation. Includes : process id, type, input and output values, start and end time.
History (H)	History of an execution. Includes : set of operations, returns-before partial order, same-session and same-object equivalence relations.
Visibility (vis)	Acyclic partial order on operations. Accounts for propagation of write operations.
Arbitration (ar)	Total order on operations. Specifies how the system resolves conflicts.

Table E.1 – Résumé des entités les plus pertinentes du modèle décrit dans le Chapitre 2.

À titre d'exemple, une sémantique de cohérence qui exige le respect de l'ordre en temps réel comprendrait le prédicat suivant :

$$\textsc{RealTime} \triangleq rb \subseteq ar \tag{E.1}$$

Nous avons utilisé ce modèle pour formuler des définitions formelles pour la sémantique de plus de 50 modèles de cohérence que nous avons étudié — les définitions formelles sont rapportées dans l'Annexe A. Pour le reste, nous avons présenté des descriptions informelles qui donnent un aperçu de leur caractéristique et de leurs forces relatives.

De plus, grâce à l'approche axiomatique que nous avons adoptée, nous avons mis en place un cluster de sémantique selon des critères qui tiennent compte de leur nature et de leurs caractères communs. Grâce à ces nouvelles définitions formelles, nous sommes en mesure de les comparer et de les placer dans une hiérarchie partiellement ordonnée selon leur «force» sémantique, comme le montre la Figure 2.1.

En outre, nous établissons la correspondance entre ces sémantiques et les implémentations de prototypes et de systèmes décrit dans la littérature de recherche (Annexe C). Enfin, dans l'Annexe B, nous fournissons des preuves de relations de force entre les modèles sémantiques mis en évidence dans la Figure 2.1.

E.2 Stockage robuste et fortement cohérent dans le Cloud hybride

Le stockage dans le Cloud hybride consiste à stocker des données sur des locaux privés ainsi que sur un (ou plusieurs) fournisseur de stockage public dans un Cloud distant. Pour les entreprises, cette conception hybride apporte le meilleur des deux mondes : les avantages du stockage public dans le Cloud (par exemple, l'élasticité, les systèmes de paiement flexibles et la durabilité sans danger pour les catastrophes) ainsi que le contrôle des données d'entreprise. En un sens, le Cloud hybride élimine dans une large mesure les préoccupations que les entreprises ont de faire confiance à leurs données aux Clouds commerciaux. En conséquence, les solutions de stockage de Clouds hybrides de classe entreprise sont en plein essor avec tous les principaux fournisseurs de stockage offrant leurs solutions exclusives.

Comme une approche alternative pour résoudre les problèmes de confiance et de fiabilité associés aux fournisseurs publics de stockage dans le Cloud, plusieurs travaux de recherche (par exemple, [53, 42, 245]) ont permis de stocker les données de manière robuste dans les Clouds publics en exploitant plusieurs fournisseurs de Cloud. En bref, l'idée derrière ces systèmes publics de stockage multi-nuages tels que DepSky [53], ICStore [42] et SPANStore [245] est de tirer parti de plusieurs fournisseurs de Cloud dans le but de distribuer la confiance à travers les Clouds, d'accroître la fiabilité, la disponibilité et la performance et/ou l'adressage des problèmes de verrouillage des fournisseurs (par example, le coût).

Cependant, les systèmes de stockage multi-Clouds robustes existants souffrent de graves limites. En particulier, la robustesse de ces systèmes ne concerne pas la cohérence : ces systèmes fournissent une cohérence au mieux proportionnelle [53] à celle des Clouds sous-jacents qui fournit très souvent seulement une cohérence éventuelle [237]. En outre, ces systèmes de stockage dispersent les métadonnées de stockage dans les Clouds publics, ce qui augmente la difficulté de la gestion du stockage et affecte les performances. Enfin, les systèmes de stockage multi-Clouds existants ignorent les ressources sur des locaux privés.

Nous proposons Hybris, le premier système de stockage de Cloud hybride robuste, qui unifie l'approche du Cloud hybride avec celle du stockage multi-Clouds robuste.

E.2.1 Principales caractéristiques d'Hybris

Hybris est un système de stockage à valeurs multiples et multi-lecteurs qui garantit une forte cohérence (c.-à-d., linearisation [139]) de lectures et écritures. L'idée clé derrière Hybris est qu'il conserve tout stockage de *métadonnées* sur des locaux privés, même lorsque ces métadonnées concernent des données externalisées aux Clouds publics (voir la Figure 3.1 pour l'architecture de haut niveau d'Hybris) . La métadonnée Hybris est légère (cc 40 octets par objet) et se compose de : i) numéro de version, ii) hash, iii) pointeurs vers des Clouds qui stockent la copie de la valeur et iv) la taille de la valeur. Hybris réplique les données pour la fiabilité en utilisant des API de stockage en nuage (par exemple, Amazon S3, Rackspace CloudFiles, etc.). Les métadonnées sont également reproduites dans des locaux privés — donc, la conception d'Hybris ne présente aucun point d'échec.

Plus précisément, notre modèle de système est *hybride*. À savoir, les clients et (une minorité de) serveurs de métadonnées peuvent échouer en s'écrasant et expérimentent des périodes arbitraires, longues mais finies, d'asynchronisme des communications dans un Cloud privé. En revanche, les Clouds publics ne sont pas fiables et peuvent même présenter des comportements malveillants. Nous modélisons la communication entre les Clouds publics et les clients comme purement asynchrones, sans aucune limite aux retards des communications.

La conception d'Hybris permet les caractéristiques suivantes.

Cohérence renforcée Hybris garantit la linéarisation des lectures et des écritures même en présence de Clouds publics finalement cohérents. À cette fin, Hybris utilise un nouveau schéma que nous appelons *renforcement de la cohérence* : il tire parti d'une forte cohérence des métadonnées stockées localement pour masquer les incohérences possibles des données stockées sur des Clouds publics éventuellement cohérents. Dans l'Annexe D, nous présentons le pseudo-code du protocole mis en œuvre par Hybris, ainsi que la preuve de sa linéarisation. De plus, dans l'Annexe D.3, nous prouvons la linéarisation de Hybris en utilisant le cadre que nous avons introduit dans le Chapitre 2.

BFT avec 2f + 1 Nuages non fiables Hybris peut masquer les défauts malveillants (aussi appelés byzantins) de Clouds publics allant jusqu'à f. Cependant, contrairement aux systèmes de stockage BPT (de l'anglais, Byzantine-Fault Tolerance) qui impliquent des nœuds de stockage de données 3f + 1 pour masquer les f malveillants, Hybris est le premier système de stockage

BFT qui ne nécessite que 2 nœuds +1 (nuages publics) dans le pire des cas. La mise en œuvre de référence d'Hybris prend également en charge le cryptage de clé symétrique côté client pour la confidentialité des données.

Efficacité Hybris est efficace et encourt un faible coût. Dans le cas commun, une écriture de Hybris implique un peu moins de $f + 1$ de nuages publics, alors qu'une lecture implique seulement un seul nuage, même si tous les nuages ne sont pas fiables. Hybris réalise ceci en utilisant des fonctions de hachage cryptographiques, et sans compter sur des primitives cryptographiques coûteuses. En stockant des métadonnées localement, Hybris évite les communications coûteux pour les opérations légères qui ont eu des problèmes avec les systèmes multi-Clouds précédents. Enfin, Hybris réduit en option les exigences de stockage en prenant en charge le code d'effacement [208], au détriment de l'augmentation du nombre de nuages impliqués.

Évolutivité L'écueil potentiel de l'adoption d'une telle architecture composée est que les ressources privées peuvent représenter goulot d'étranglement à l'échelle. Hybris évite ce problème en gardant l'empreinte des métadonnées très faible. À titre d'illustration, la variante répliquée d'Hybris maintient environ 50 octets de métadonnées par clé, ce qui est un ordre de grandeur plus petit que les systèmes comparables [53]. En conséquence, le service de métadonnées Hybris, résidant dans des locaux de confiance, peut facilement supporter jusqu'à 30k d'écriture / s et près de 200k lecture / s, tout en étant entièrement répliqué pour la tolérance de panne. En outre, Hybris offre des fonctionnalités multi-écrivains multi-lecteurs par clé grâce au contrôle de concurrence [136] sans attendre, ce qui augmente encore le passage à l'échelle d'Hybris par rapport aux systèmes basés sur le verrouillage [245, 53, 52].

Afin de mieux répondre à la diversité des besoins en matière de cohérence par rapport aux compromis de performance, Hybris implémente et met en évidence la sémantique **cohérence accordable**. À savoir, pour chaque exécution, il est possible de faire en sorte que Hybris respecte deux modèles de cohérence en alternative à la linéarisation, c'est-à-dire la cohérence *read-your-write* et *bounded staleness*. Enfin, Hybris implémente **writes transactionnelles**. Ces opérations permettent des écritures atomiques qui couvrent différentes clés.

E.2.2 Implémentation et résultats

Pour maintenir une petite empreinte d'Hybris, nous avons choisi de reproduire de manière robuste ses métadonnées en utilisant le service de coordination Apache ZooKeeper [142] (voir la Figure 3.1). Les clients d'Hybris agissent simplement comme clients de ZooKeeper — notre système n'implique aucune modification à ZooKeeper, facilitant ainsi le déploiement d'Hybris et son adoption future. En outre, nous avons conçu le service de métadonnées Hybris pour être facilement portable de ZooKeeper à n'importe quel magasins de données RDBMS ou NoSQL répliqués et qui exportent une opération de mise à jour conditionnelle (par exemple, HBase ou MongoDB).

Nous avons implémenté Hybris en Java [1] et nous l'avons évalué à travers une série de repères. Nos résultats expérimentaux montrent que Hybris surpasse de manière constante les systèmes de stockage multi-Clouds robustes à la fine pointe de la technologie (par exemple, [53]) avec une latence inférieure jusqu'à 2-3x dans le cas commun, se compare de la même manière que les Clouds individuels tout en engendrant un faible coût. De plus, en utilisant le service de métadonnées basé sur ZooKeeper déployé sur trois serveurs de produits, Hybris lit l'échelle au-delà de 150 kops/s, tandis que les écritures augmentent jusqu'à 25 kops/s (resp., 35 kops/s) avec SSD (resp., NVRAM) comme solution de durabilité de ZooKeeper. Les Figures 3.4, 3.5 et 3.7 illustrent certains résultats sur la performance globale et l'évolutivité d'Hybris.

[1] Le code d'Hybris est disponible sur : https://github.com/pviotti/hybris

E.3 Vérification déclarative et automatisée de la cohérence

La cohérence est le principal critère d'exactitude des systèmes de stockage distribués. Malgré les récents efforts *consistency-by-construction* proposés par des méthodes formelles [243, 168], dans le monde réel, la plupart des systèmes de stockage sont encore développés de manière ponctuelle. Plus précisément, la plupart du temps, les praticiens commencent par la mise en oeuvre, et plus tard procèdent à la vérification par tests limités (ex. utilisation de tests unitaires et/ou tests d'intégration). En conséquent, diverses approches ont été conçues pour donner une manière générale de vérification de la mise en oeuvre de modèles de cohérence. Cependant, les approches conçues jusqu'à présent ne s'appliquent qu'à un sous-ensemble restreint de modèles de cohérence et sont prévues pour fonctionner dans des contextes spécifiques (ex. stockage dans le Cloud, bases de données transactionnelles, etc.).

Nous estimons que l'étape initiale pour construire un cadre de test de cohérence efficace et complet devrait être l'utilisation d'un modèle de cohérence théoriquement rationnel. Pour ce faire, nous préconisons le choix d'utiliser une approche déclarative pour définir un ensemble de sémantiques de base applicables à l'ensemble des modèles de cohérence. En particulier, nous voudrions avoir de la visibilité côté client, des opérations de lecture/écriture et des configurations de l'état global. En utilisant tirant parti des prédicats logiques qui incluent ces deux perspectives, nous définissons une sémantique de cohérence qui capture, sous forme de composants graphiques, les aspects les plus importants des différentes exécutions du système, en l'occurrence la commande et la visibilité des événements. De cette manière, la vérification d'une implémentation d'une sémantique de cohérence donnée revient à trouver, pour une exécution, les différentes configurations de l'état global qui valident un prédicat logique, tout en prenant en compte les événements côté client.

En bref, nous proposons une approche déclarative et basée sur la propriété de la vérification de la cohérence, dans le volet des travaux précédents réalisés dans le cadre d'un test de logiciel générique [83]. Nous expérimentons cette approche en mettant en œuvre Conver, un prototype pratique de vérification de la cohérence, développé en Scala. [2]

[2] Le code de Conver est disponible à l'adresse : https://github.com/pviotti/conver

E.3.1 Principales caractéristiques

Nous avons trouvé un modèle supportant le paradigme deńclaratif pour les sémantiques de cohérence dans le travail Burckhardt [65], et que nous avons étendu et raffiné comme décrit dans le Chapitre 2.

Ce modèle supporte l'expression de la sémantique déclarative et composable de la cohérence en tant que prédicats logiques de premier ordre sur les entités graphiques qui décrivent la visibilité et l'ordre des opérations.

Afin de vérifier les modèles de cohérence listés dans le Chapitre 2, nous devrions construire les entités qui composent des exécutions abstraites. En particulier, nous devrions établir l'ordre d'arbitrage (ar) établi par le système de stockage en cours de vérification.

Cependant, l'approche de test en boite noire que nous avons adoptée rend difficile l'élaboration de l'ordre d'arbitrage, car il ne peut utiliser que les processus clients. Compte tenu de cela, nous avons rejeté l'ordre d'arbitrage global pour vérifier la sémantique de la cohérence et nous nous sommes concentrés uniquement sur les informations qui pourraient être collectées du côté du client. En conséquence, le spectre de la sémantique vérifiable par notre mise en œuvre actuelle est réduit à ceux répertoriés dans la Figure 4.1.

Dans ce qui suit, nous discutons brièvement les principales caractéristiques de Conver.

Génération de cas de test ciblés Nous avons mis en place une série d'heuristiques pour générer des exécutions de cas de test adaptée pour une vérification sémantique spécifique. Par exemple, en fonction du modèle de cohérence sous vérification, Conver ajuste le rapport entre les opérations de lecture et d'écriture, ou établit une coordination sans limites entre les clients pour mieux exercer leur concurrence.

Injection de fautes L'injection de fautes est une technique d'essai qui vise à remettre en cause la mise en œuvre des banques de stockage de données en rajoutant le non-déterminisme intrinsèque des systèmes distribués. Grâce à son approche de test en boîte noire (de l'anglais "black box testing"), Conver a été instrumenté pour injecter des défauts externes, tels que les partitions réseau et les pannes de processus.

Dans ce qui suit, nous identifions plusieurs extensions possibles de Conver, que nous prévoyons d'implémenter dans un temps futur.

Modèles de cohérence transactionnels Conver peut être étendu pour supporter la vérification des modèles de cohérence transactionnels [77]. Le modèle d'état pris en charge par Conver a déjà été adapté pour exprimer la sémantique transactionnelle. De ce fait, cette extension impliquerait le soutien d'entités sémantiques supplémentaires pour exprimer les fonctionnalités transactionnelles dans le modèle et la mise en œuvre d'un algorithme pour détecter les anomalies transactionnelles [8].

Correspondance à des invariants au niveau applicatif Une tendance actuelle de la recherche préconise l'utilisation d'invariants au niveau applicatif pour permettre Une approche fine (mais moins portable) de la verification et la réalisation de la cohérence. [34, 125]. À cet égard, nous pensons qu'il serait intéressant d'étudier comment ces invariants correspondent à la sémantique de bas niveau dans Conver. Pour ce faire, une couche supplémentaire dans l'architecture Conver peut imiter les cas d'usages applicatif communs et les modèles d'interaction, et les associer à la sémantique des lecture-écriture de base de données.

Mesure de latence de cohérence Conver peut être étendu pour mesurer la latence des opérations et le débit. De cette façon, il servirait d'outil pour explorer les compromis entre la cohérence et la performance dans les bases de données, à la fois hors ligne et de manière incrémentale, comme le montrent les travaux récents de Fan et al. [107].

Simulation et vérification Les traces d'exécution générées par Conver peuvent être entrées dans des framework de simulation. En outre, les données des exécutions de Conver peuvent aider à construire des modèles mathématiques sur le stockage de données sous test qui pourrait permettre le développement de spécifications formelles [141], qui, à son tour, permettrait le contrôle du modèle ou des preuves formelles d'exactitude.