THÈSE DE DOCTORAT DE
l'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Marek ZAWIRSKI

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Dependable Eventual Consistency with Replicated Data Types

soutenue le 14 janvier 2015

devant le jury composé de :

M. Marc SHAPIRO	Directeur de thèse
M. Pascal MOLLI	Rapporteur
M. Luís RODRIGUES	Rapporteur
M. Carlos BAQUERO	Examinateur
M. Jerzy BRZEZIŃSKI	Examinateur
M. Sebastian BURCKHARDT	Examinateur
M. Peter DICKMAN	Examinateur
M. Pierre SENS	Examinateur

Abstract

Web applications rely on replicated databases to place data close to their users, and to tolerate failures. Many replication systems opt for eventual consistency, which offers excellent responsiveness and availability, but may expose applications to the complexity of concurrency and failures. To alleviate this problem, recent replication systems have begun to strengthen their interface with additional guarantees, such as *causal consistency*, which protects the application from ordering anomalies, and *Replicated Data Types* (RDTs), which encapsulate concurrent updates via high level object interface and ensure convergent semantics. However, dependable algorithms for these abstractions come at a high cost in metadata size. This thesis studies three related topics: *(i)* design of RDT implementations with minimized metadata; *(ii)* design of consistency algorithms with minimized metadata; and *(iii)* the limits of the design space.

Our first contribution is a study of metadata complexity of RDTs. RDTs need metadata to provide rich semantics under concurrency and failures. Several existing implementations incur high overhead in storage space, in the number of updates, or polynomial in the number of replicas. Minimizing metadata is nontrivial without impacting their semantics or fault-tolerance. We design *optimized* set and register RDTs with metadata overhead reduced to the number of replicas. We also demonstrate metadata *lower bounds* for six RDTs, thereby proving optimality of four implementations. As a result, RDT designers and users can better navigate the design space.

Our second contribution is the design of SwiftCloud, a *replicated causally-consistent RDT object database for client-side applications*, e.g., mobile or in-browser apps. We devise algorithms to support high numbers of client-side partial replicas backed by the cloud, in a fault-tolerant manner, with small and bounded metadata. We demonstrate how to support available and consistent reads and updates, at the expense of some slight data staleness; i.e., our approach trades freshness for scalability (small and bounded metadata, parallelism), and availability (ability to switch between data centers on failure). Our experiments with thousands of client replicas confirm the design goals were achieved at a low staleness cost.

Acknowledgement

This work would not be possible without many inspiring and friendly people that I had a chance to meet and work with.

I am thankful to my advisor, Marc Shapiro, for his insights and constant encouragement that he offered me throughout our countless meetings. He taught me the role of communication and gave me the opportunity to collaborate with other people. I joined Marc's group on the grounds of my BSc/MSc experience at PUT. For that period, I owe my gratitude to Jerzy Brzeziński.

I am grateful to Nuno Preguiça, who always had time to review "unsolvable" problems and questionable solutions with me. I thank Nuno, as well as my other co-authors, Annette Bieniusa, Sérgio Duarte, Carlos Baquero, and Valter Balegas for their exceptionally amicable attitude. They all significantly contributed to Chapter 3 and Part III of this thesis.

I appreciate the guidance of Sebastian Burckhardt during my internship at Microsoft Research. I thank Sebastian for the three inspiring and enjoyable months, his openness and willingness to share his experience. His ideas considerably influenced Part II of this work.

I am thankful to all my committee members, to the reviewers in particular, for accepting to take part in this process without hesitation, offering me their precious time and feedback.

Among many others, I would like to particularly thank Masoud Saeida Ardekani, Alexey Gotsman, Pierre Sutra, and Hongseok Yang for our inspiring meetings, and all the ideas they contributed with. The feedback I got from Peter Bailis, João Leitão, Allen Clement, Vivien Quéma, Tyler Crain, and Basho's team shaped the presentation along with my own view of this work.

I am grateful to Google (Googlers) for generously supporting me with a PhD fellowship, as well as for matching me with an excellent team in Zürich for my superb internship experience.

The time I spent with my fellow grad-student friends is unforgettable. Thanks Masoud Saeida Ardekani, Pierpaolo Cincilla, Lisong Guo, Florian David, Alejandro Tomsic, Valter Balegas, and many others for sharing a round of beer with me, or a lap in Mario Kart, whenever (in)appropriate. Thank you Pierpaolo, Florian and Kasia Kapusta for your nearly 24h/7j translation services.

I am indebted to my family, especially to my parents, my sister, and my brother, for their support in all plans that I decided to carry out. They helped me take my first steps in this world, speak first English and French words, and even write my first lines of code. The presence of my dear friends all around the world has also been invaluable. Finally, I especially thank my girlfriend Marta for all her patience, efforts and smile.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Part I

Preliminaries

Chapter 1

Introduction

High availability and responsiveness are essential for many interactive web applications. Such applications often share mutable data between users in geographically distributed locations. To ensure availability and responsiveness, they rely on *geo-replicated databases* or even on *client-side data replicas*. Geo-replication provides users with access to a local replica in a nearby data center, whereas client-side replication provides access to a local replica at a user's device. This local data access is neither affected by the cost of network latency between replicas nor by failures (e.g., replica disconnection) [30, 37, 43, 44]. When, furthermore, performance and availability of update operations are crucial for an application, then these updates need to be also performed locally, without coordination with remote replicas. This entails that *concurrent updates* must be accepted at different replicas and replicated asynchronously.

Unfortunately, asynchronous replication is at odds with strong consistency models, such as linearizability or serializability [20, 53]. Strong consistency offers applications a single common view of the distributed database, but requires to execute operations in a global total order at all replicas (synchronous replication). The incompatibility of availability and fault-tolerance with strong consistency, known as the CAP theorem, forces a choice of weaker consistency models, i.e., *eventual consistency* [44, 50, 79].

Under eventual consistency, replicas are allowed to diverge transiently due to concurrent updates, but they are expected to eventually converge to a common state that incorporates all updates [79, 100]. Intermediate states expose applications to consistency anomalies. This is undesirable for users, and poses a challenge to the implementation of the database and the application [43]. Difficulties include detection of concurrent conflicting updates, and their convergent resolution, or protection from asynchronous delivery of updates and from failures.

Two complementary abstractions were proposed to alleviate the issues of eventually consistent replication. These abstractions encapsulate the complexity of asynchronous replication and failures behind an interface with a well-defined behavior. A *dependable* implementation of such an interface guarantees the behavior regardless of the underlying infrastructure (mis)behavior.

First, *Replicated Data Types* (RDTs) expose data in the database as typed objects with high-level methods. RDTs rely on type and method semantics for convergence [34, 92]. With RDTs, the database consists of objects such as counter, set, or register, that offer read and update methods, such as `increment` for a counter, or `add` and `remove` for a set. A data type encapsulates convergent replication, and resolves concurrent updates according to the type's defined semantics.

Second, *causal consistency* offers partial ordering of updates across objects [4, 66, 67]. Informally, under causal consistency, every application process observes a monotonically non-decreasing set of updates, which includes its own updates, in an order that respects the causality between operations. Applications are protected from causality violations, e.g., where read could observe update b but not update a that b is based upon. *Transactional* causal consistency further simplifies programming with atomicity. It guarantees that all the reads of a transaction come from a same database snapshot, and either all its updates are visible, or none is [14, 66, 67].

These abstractions facilitate programming against a replicated database, but their dependable implementation comes at the cost of storage and network metadata. Metadata is required, for instance, to relate asynchronously replicated updates, i.e., to determine which one caused another, or to compare replica states after a transient failure. Some metadata cost is inherent to concurrency, and to uncertainty of the status of an unresponsive replica, which could be unavailable either permanently or transiently (and could possibly perform concurrent updates).

This thesis studies the design of dependable RDT and causal consistency algorithms with minimized metadata, and the limits and the trade-offs of their design space.

1.1 Contributions

This dissertation makes two main contributions in the area of dependable algorithms for eventual consistency: a study of space optimality of RDTs, including impossibility results and optimized implementations, and the design of a causally-consistent RDT database for client-side applications. We discuss both contributions in more detail in the remainder of this section.

1.1.1 Optimality of Replicated Data Types

In the first part of the thesis, we consider the problem of minimizing metadata incurred by RDT implementations. In addition to client-observable data, RDTs use metadata to ensure correctness in the presence of concurrent updates and failures. The metadata manifests either in the implementation of RDTs or in a middleware for updates dissemination. A category of *state-based* RDT implementations, i.e., objects that communicate by exchanging their complete state [92], uses metadata directly in the RDT state. Such state-based implementations are used, for example, by server-side geo-replicated object databases [3]. Some state-based RDTs incur substantial metadata overhead that impacts storage and bandwidth cost, or even the viability of an implementation. On the other hand, the design of RDT implementations that reduce the

Data type	**Existing implementation**		**Optimized impl.**	**Any impl.**
	Source	**Bounds**	**bounds**	**lower bounds**
counter	$\underline{\text{[92]}}$	$\underline{\widehat{\Theta}(n)}$	—	$\widehat{\Omega}(n)$
add-wins set	[93]	$\widehat{\Theta}(m \lg m)$	$\underline{\text{new: } \widehat{\Theta}(n \lg m)}$	$\widehat{\Omega}(n \lg m)$
remove-wins set	[22]	$\widehat{\Omega}(m \lg m)$	—	$\widehat{\Omega}(m)$
last-writer-wins set	[22, 56]	$\widehat{\Omega}(m \lg m)$	—	$\widehat{\Omega}(n \lg m)$
multi-value register	[77, 93]	$\widehat{\Theta}(n^2 \lg m)$	$\underline{\text{new: } \widehat{\Theta}(n \lg m)}$	$\widehat{\Omega}(n \lg m)$
last-writer-wins register	$\underline{\text{[56, 92]}}$	$\underline{\widehat{\Theta}(\lg m)}$	—	$\widehat{\Omega}(\lg m)$

Table 1.1: Summary of metadata overhead results for different data types. Underlined implementations are optimal; n and m are the numbers of, respectively, replicas and updates; $\widehat{\Omega}, \widehat{O}, \widehat{\Theta}$ are lower, upper and tight bounds on metadata, respectively.

size of metadata is challenging, as it creates a tension between correctness w.r.t. type semantics, efficiency, and fault-tolerance.

We formulate and study the metadata complexity problem for state-based RDTs. We define a metadata overhead metric, and we perform a worst-case analysis in order to express asymptotic metadata complexity as a function of the number of replicas or of updates. An analysis of six existing data type implementations w.r.t. this metric indicates that most incur substantial metadata overhead, higher than required, linear in the number of updates, or square in the number of replicas. We also observe that the concurrent semantics of a data type, i.e., its behavior under concurrent updates, has a critical impact on the exact extent of the metadata complexity.

The two main contributions of this part of the thesis are *positive results*, i.e., optimized data type implementations, and *impossibility results*, i.e., lower bound proofs on the relative metadata complexity for some data type. It is natural to search for positive results first. However, it is a laborious process that requires developing nontrivial solutions, potentially incorrect ones. A lower bound sets the limits of possible optimizations and can show that an implementation is asymptotically optimal. Our lower bound proofs use a common structure that we apply to each data type using semantics-specific argument. This marks the end of the design process, saves designers from seeking the unachievable, and may lead them towards new design assumptions. Together, both kinds of results offer a comprehensive view of the metadata optimization problem.

Table 1.1 summarizes our positive and impossibility results for all studied data types. From left to right, the table lists prior implementations and their relative complexity, expressed as metadata overhead, as well as the complexity of our optimizations (if any), and our lower bounds on complexity of any implementation. We underline asymptotically optimal implementations.

We show that the existing state-based implementation of counter data type is optimal. Counter requires vector of integers to handle concurrent increments and duplicated or reordered messages.

Set data type interface offers a range of different semantics choices w.r.t. how to treat concurrent operations on the same element to converge. For instance, a priority can be given to some type of operation (`add` or `remove`) or operations can be arbitrated by timestamps (last-writer-wins [56]). These alternatives are uneven in terms of the metadata cost. Prior implementations all incur overhead in the order of the number of updates, as they keep track of the logical time of remove operations. However, this is not necessary for all set variants. Our optimized implementation of add-wins set reduces this cost to the optimum, i.e., the order of the number of replicas, by adapting a variant of version vectors to efficiently summarize observed information [77]. The lower bound on remove-wins set shows that such an optimization cannot be applied on this semantics. It is an open problem whether it is possible for last-writer-wins set.

Register type also comes in different variants. Similarly, we show they are uneven in metadata complexity. A last-writer-wins register uses timestamps to arbitrate concurrent assignments, whereas multi-value register identifies values of all conflicting writes and presents them to the application. The existing implementation of last-writer-wins register has negligible overhead and is the optimal one. On the contrary, we find that the existing implementation of multi-value register has substantial overhead, square in the number of replicas, due to inefficient treatment of concurrent writes of the same value. Our optimization alleviates the square component, and reaches the asymptotically optimal overhead, using nontrivial merge rules for version vectors.

1.1.2 Causally-Consistent Object Database for Client-Side Applications

In the second part of the thesis, we study the problem of providing object database with extended, causal consistency guarantees across RDT objects, at the client-side.

Client-side applications, such as in-browser and mobile apps, are poorly supported by the current technology for sharing mutable data over the wide-area. App developers resort to implementing their own ad-hoc application-level cache and buffers, in order to avoid slow, costly and sometimes unavailable round-trips to a data center, but they cannot solve system issues such as fault tolerance, or consistency/session guarantees [34, 96]. Recent client-side systems ensure only some of the desired properties, i.e., either make only limited consistency guarantees (at the granularity of a single object or of a small database only), do not tolerate failures, and/or do not scale to large numbers of client devices [18, 30, 37, 40, 69]. Standard algorithms for geo-replication [8, 12, 46, 66, 67] are not easily adaptable, because they were not designed to support high numbers of client replicas located outside of the server-side infrastructure.

Our thesis is that the system should be ensuring correct and scalable database access to client-side applications, addressing the (somewhat conflicting) requirements of consistency, availability, and convergence [68], at least as well as server-side systems. Under these requirements, the strongest consistency model is *transactional causal consistency with RDT objects* [66, 68].

Supporting thousands or millions of client-side replicas, under causal consistency with RDTs, challenges standard assumptions. To track causality precisely, per client, would create unacceptably fat metadata; but the more compact server-side metadata management approach has fault-tolerance issues. Additionally, full replication at high numbers of resource-poor devices would be unacceptable [18]; but partial replication of data and metadata could cause anomalous message delivery or unavailability. Furthermore, it is not possible to assume, like many previous systems [8, 46, 66, 67], that the application is located inside the data center (DC), or has a sticky session to a single DC, to solve fault tolerance or consistency problems [13, 96].

In the second part of the thesis, we address these challenges. We present the algorithms, design, and evaluation of SwiftCloud, the first distributed object database designed for a high number of replicas. It efficiently ensures consistent, available, and convergent access to client nodes, tolerating failures. To achieve this, SwiftCloud uses a flexible client-server topology, and decouples reads from writes. The client *writes fast* into the local cache, and *reads in the past* (also fast) data that is consistent, but occasionally stale. Our approach includes two major techniques:

1. **Cloud-backed support for partial replicas.** To simplify consistent partial replication at the client side and at the scale of client-side devices, we leverage the DC-side full replicas to provide a consistent view of the database to the client. The client merges this view with his own updates to achieve causal consistency. In some failure situations, a client may connect to a DC that happens to be inconsistent with its previous DC. Because the client does not have a full replica, it cannot fix the issue on its own. We leverage "reading in the past" to avoid this situation in the common case, and provide control over the inherent trade-off between staleness and unavailability: namely, a client observes a remote update only if it is stored in some number $K \geq 1$ of DCs [69]. The higher the value of K is, the more likely that an update is in both DCs, but the higher is the staleness.

2. **Protocols with decoupled, bounded metadata.** Our design funnels all communication through DCs. Thanks to this, and to "reading in the past," SwiftCloud can use metadata that decouples two aspects [61]: it *tracks causality* to enforce consistency, using small vectors assigned in the background by DCs, and *uniquely identifies* updates to protect from duplicates, using client-assigned scalar timestamps. This ensures that the metadata remains small and bounded. Furthermore, a DC can prune its log independently of clients, replacing it with a summary of delivered updates.

We implement SwiftCloud and demonstrate experimentally that our design reaches its objective, at a modest staleness cost. We evaluate SwiftCloud in Amazon EC2, against a port of WaltSocial [95] and against YCSB [42]. When data is cached, response time is two orders of magnitude lower than for server-based protocols with similar availability guarantees. With three DCs (servers), the system can accommodate thousands of client replicas. Metadata size does not depend on the number of clients, the number of failures, or the size of the database, and increases only slightly with the number of DCs: on average, 15 bytes of metadata per update, with 3 DCs, compared to kilobytes for previous algorithms with similar safety guarantees. Throughput is comparable to server-side replication for low locality workloads, and improved for high locality ones. When a DC fails, its clients switch to a new DC in under 1000 ms, and remain consistent. Under evaluated configurations, 2-stability causes fewer than 1% stale reads.

1.2 Organization

The thesis is divided into three parts. The first part contains this introduction and Chapter 2, which introduces the common background of our work: RDT model with examples.

The second part focuses on the problem of RDT metadata complexity. In Chapter 3 we formulate the problem, we evaluate existing implementations w.r.t. a common metadata metric, we explore opportunities for improvement, and we propose improved implementations of add-wins set and multi-value register RDTs. In Chapter 4, we formally demonstrate a lower bound for metadata overhead of six data types, thereby proving optimality of four implementations. We discuss the scope of our results and compare them to related work in Chapter 5.

The third part of the thesis presents the design of SwiftCloud object database. We overview and formulate the client-side replication problem in Chapter 6. In Chapter 7, we present the SwiftCloud approach, and demonstrate an implementation of this approach using small and safe metadata. Chapter 8 presents our experimental evaluation. We discuss related work in Chapter 9, where we categorize existing approaches and compare them to SwiftCloud.

Chapter 10 concludes the thesis.

1.3 Authorship and Published Results

Part of the presented material appeared in earlier publications with our co-authors.

Although we were involved in the formulation of the RDT model [34, 91–93], presented in Chapter 2, it is not our main contribution, and it is not the focus of this thesis.

The semantics and optimizations of set RDTs were co-authored with Annette Bieniusa, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Sérgio Duarte, and Valter Balegas, and published as a brief announcement at DISC'12 [23] and a technical report [21]. Some of the lower bounds and large part of the theory behind it are a result of an internship with Sebastian Burckhardt at Microsoft Research in Redmond. Both the optimized register RDT and four lower bounds were published at POPL'14 [34], co-authored with Sebastian Burckhardt, Alexey Gotsman and Hongseok Yang.

SwiftCloud was designed with Nuno Preguiça, Annette Bieniusa, Sérgio Duarte, Valter Balegas, and Marc Shapiro. The first four contributed to its implementation, which drew from an earlier prototype developed by Valter Balegas. This work is under submission, and partially covered by an earlier technical report [104]. Some of the ideas related to the staleness vs. metadata trade-off exploited in SwiftCloud were seeded in a short paper at EuroSys'12 HotCDP workshop [86], co-authored with Masoud Saeida Ardekani, Pierre Sutra and Marc Shapiro.

Chapter 2

Replicated Data Types

Any problem in computer science can be solved with another level of indirection.

David Wheeler

Contents

This chapter presents the background on the concept of *Replicated Data Type* (RDT), previously formalized in similar ways by Baquero and Moura [16], Roh et al. [85], and Shapiro et al. [92].[1] An RDT is a type of replicated object accessed via highly available methods, i.e., methods that provide immediate response, with well-defined *concurrent semantics* [34, 92]. RDT objects encapsulate the complexity of convergent replication over distributed, unreliable infrastructure behind a relatively simple interface. RDTs include basic types such as counters, sets, and registers.

In Section 2.1, we recall the motivation for RDTs. We outline what problems they address, and what is specific to the RDT approach. In particular, we highlight what makes RDTs a popular abstraction for managing highly available replicated data [3, 25, 95].

In Section 2.2, we introduce a formal model of a *data type* and its *implementation*. We illustrate the model with examples from different *categories* of implementations, in order to demonstrate the spectrum of RDT implementations, semantics, and some of the design challenges.

Not all RDT implementations provide useful behavior. We examine and specify what can characterize the intended behavior in Section 2.3: from a primary convergence requirement, to a precise *specification* of concurrent semantics of a data type. In Section 2.4, we define an execution model that describes object implementation and allows us to formally relate an implementation with a specification by a correctness condition. The correctness definition is parameterized, which leads us to formal categorization of implementations.

The material in this chapter is a prerequisite to Part II of the thesis, and is recommended reading before Part III. We highlight the relevant problems as they appear, in particular the challenges related to metadata.

The formalism and presentation used throughout this chapter follows closely a more general theory of Burckhardt et al. [33, 34], with some simplifications in both notation and model.

2.1 Motivation

High availability and responsiveness are important requirements for many interactive web applications. These applications often share mutable data between users in geographically distributed locations, i.e., data can be accessed and modified by users in different locations. To address the requirements, applications often rely on *geo-replicated databases* that replicate data across data centers spread across the world, or even on *client-side data replicas* at user devices [37, 43, 66, 95]. Replication systems provide users with fast access to local replica. This local access is unaffected by high network latency between replicas or by failures [43, 44, 66, 92, 95].[2]

[1] We use the term Replicated Data Types after a common model of Burckhardt et al. [34]. The same or specialized concept appears in the literature under a variety of names: Conflict-free/Convergent/Commutative Replicated Data Types (CRDT) [92, 93], Replicated Abstract Data Types (RADT) [85], or Cloud Types [32]. We highlight the differences between these models where relevant.

[2] Inter-continental network round-trip times reach hundreds of milliseconds, which exceeds the threshold of perception by human users and negatively impacts their experience [55, 89].

Failures are prevalent in large-scale systems, and range from individual server failures, variety of data center failures, to network failures, including partitions [11, 57]. When client-side replication is involved, extended replica disconnections are granted [79].

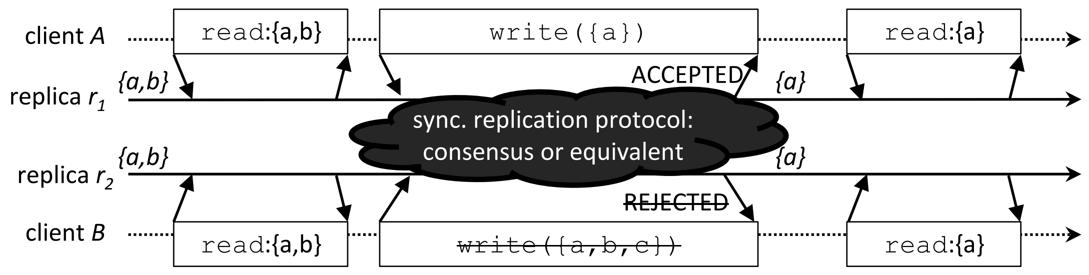
When high performance and availability of update operations are important to the application, they need to be performed at a local replica, without coordination with remote replicas [15]. This entails that updates must be accepted *concurrently* at different replicas and that they must be replicated asynchronously, which causes replicas to diverge and to execute updates in different orders. Unfortunately, this is inherently incompatible with strong consistency models, such as linearizability [53] or serializability [20], which rely on a global total order of updates (synchronous replication). This incompatibility, known as the CAP theorem [50], forces asynchronous replication to offer weaker consistency models only; namely, variants of *eventual consistency* [44, 79]. Under eventual consistency, replicas are allowed to transiently diverge under the expectation of eventual convergence towards a common value that incorporates all updates.

Let us examine more closely the differences between strongly and eventually consistent data replication. We discuss some challenges of implementing and using the latter that motivate our interest in the RDT abstraction.

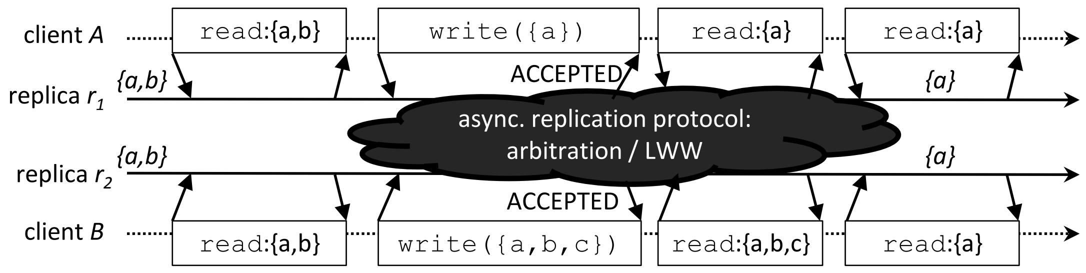
Any replication protocol needs to address **conflicts**, which we discuss next. Consider some replicated data item. For example, assume that the item represents a set of elements, such as the members of a group in a social network application, with the initial value $\{a,b\}$. In a replicated system, two clients connecting to different replicas might concurrently overwrite it with values $\{a\}$ and $\{a,b,c\}$. Concurrent writes to the same item are *conflicting*. There are four main approaches to address conflicts. One is to accept only some updates to prevent conflicts (strong consistency). The other three *optimistically* accept all concurrent updates and resolve conflicts eventually (eventual consistency). We review these approaches and illustrate them with the help of our $\{a,b\}$ example in Figure 2.1.

A. **Conflict avoidance.** Before accepting an update, the database executes a synchronous protocol to agree on a common order of updates execution on all replicas. Such a protocol consistently rejects all but one of the conflicting updates. This forces other writers to abort and retry rejected operations [43]. As this solution is strongly consistent, it is easy to program against, because replication is transparent to the application. However, it is not highly available [50], so it is out of the scope of this work.

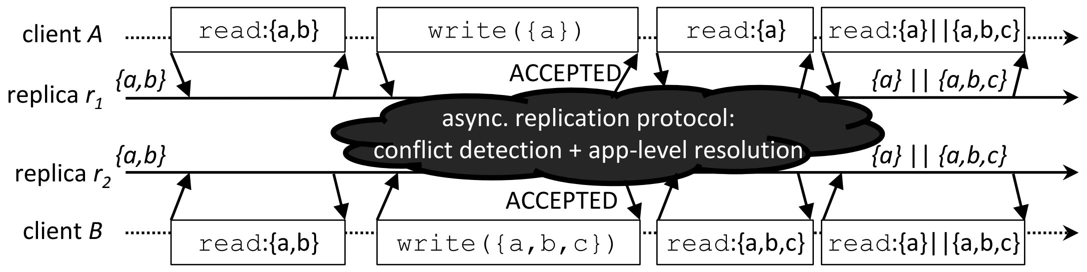
B. **Conflict arbitration using timestamps.** The database accepts all updates and replicates them, but of two concurrent updates, one dominates the other, the "last" one according to timestamps of write operations. This heuristic is known as *last-writer-wins* (LWW) [56]. Arbitrary updates may be *lost*, i.e., overwritten without being observed, even if they were reported as accepted to the client [58]. In our example in Figure 2.1B, assuming that timestamp of $\{a\}$ is larger than that of $\{a,b,c\}$, the outcome of `write`($\{a,b,c\}$) is lost. This is not a satisfying solution.



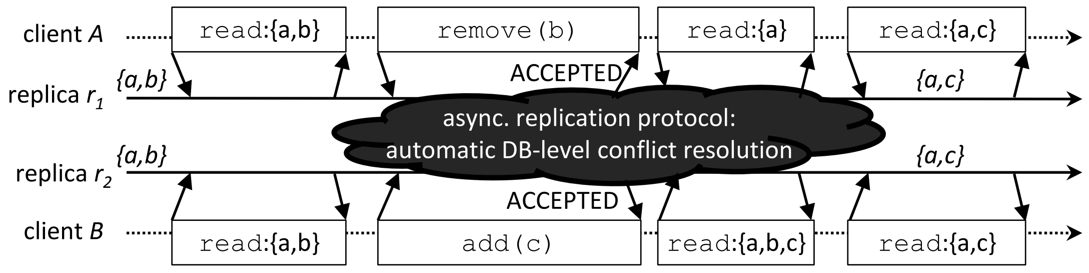
(A) Conflict avoidance approach with synchronous protocol offering strong consistency.



(B) Conflict arbitration approach with asynchronous protocol causing lost updates.



(C) Conflict detection approach with asynchronous protocol relying on application-level conflict resolution.



(D) Automatic conflict resolution approach with asynchronous protocol embedding conflict resolution.

Figure 2.1: Comparison of different approaches to conflicts on an example execution with two clients issuing conflicting updates on two different replicas, r_1 and r_2. Boxes indicate operations issued by clients; arrows indicate operation invocation and response.

C. **Conflict detection and application-level resolution.** To avoid lost updates, the database detects concurrent writes after they were accepted, and presents the set of concurrently-written values to the application. It is up to the application to resolve the conflict [44, 59, 80, 97]. Figure 2.1C illustrates this approach. Unfortunately, resolving conflicts is difficult ad-hoc at the application layer, and may not converge. For example, it is not obvious how to combine values $\{a\}$ and $\{a,b,c\}$ from our example without violating intentions of the clients that wrote them. It may require the application to add more metadata to the values it stores (e.g., to record the fact that b was *removed* and c was *added*), which may be costly and complex, especially in the presence of replica failures [44]. As a last resort, the application may present the conflict to the user and let the user decide or confirm its choice. In either case, however, the conflict resolution may need to be run concurrently with other updates and across different application processes or users, which may generate even more conflicts, and prevent or delay database convergence [38].[3] For example, one user may resolve the conflict to $\{a,c\}$, while another resolves it to $\{a,b,c\}$, creating a new conflict. This approach can quickly become *unstable* and *complex* at the application layer.

D. **Automatic database-level conflict resolution.** As an alternative, the *database* itself may automatically resolve conflicting updates. This is different from the previous approach, because it assumes the database has enough knowledge to embed convergent conflict resolution in the replication protocol. For example, as illustrated in Figure 2.1D, if the database is aware that the read of $\{a,b\}$ followed by the write of $\{a\}$ is meant to *remove* element b from a *set*, and the other pair is meant to add c, it can integrate both updates. This can be also more efficient than the application-level solution, since the metadata can be managed by the database, i.e., lower in the abstraction stack. The challenge is then to design a database protocol that will tolerate concurrency and failures, still converging to a sensible value that integrates all updates. For some updates the right integration is simple (e.g., *add* and *remove* of different elements), for others it is harder (e.g., operations on the same element).

A *Replicated Data Type* (RDT) abstraction can express all three optimistic conflict resolution approaches (B)–(D), and in particular, addresses the challenges of automated database-level conflict resolution [34, 92]. The insight is to use *typing* to indicate semantics of a data item (naming the role of object) and to mutate it via *high-level object interface* (naming the role of operations), rather than low-level read-write operations. The database implementation, and in particular a data type implementation class, can leverage the data type semantics to automate and to optimize conflict resolution. The data type semantics can also serve as a contract describing the interface to an application developer, thus separating concerns and responsibilities.

With RDTs, an application organizes its shared state as a collection of *typed objects*. For instance counter objects have `increment`, `decrement` and `read` methods, whereas set objects have `add(a)`, `remove(a)` and `read` methods. It is an object implementation duty to handle the burden of replication, and to resolve concurrent updates in a sensible manner, thereby releasing application programmers from this duty. Moreover, some categories of RDT implementations also address fault-tolerance of convergent replication, and tolerate messages reordering, message loss, disconnected replicas etc.

[3] An alternative would be to coordinate conflict resolution. However, this results in an unavailable protocol, similar to approach (A).

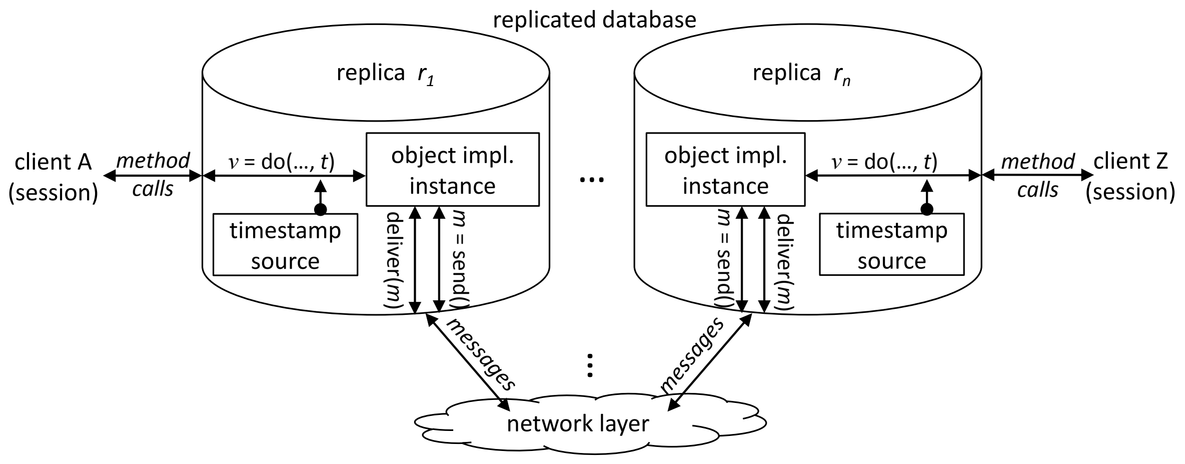


Figure 2.2: Components of the system model.

An RDT operates at the limited scope of an object. It does not support conflict-resolution involving complete database [24, 79], and does not address the other problems of eventual consistency at a larger scale, across different objects. Specifically, it does not address the somewhat orthogonal problem of update ordering, nor of atomicity guarantees. Examples of such desirable mechanisms might include cross-object causal consistency and its transactional variants [4, 18, 66]. We do not consider such ordering and atomicity guarantees for simplicity here, although the model we adopt can be extended to express them [34]. We return to intra-object ordering in Section 2.4.3, and to inter-object ordering in Part III.

2.2 System Model and Object Implementation

In this section, we formalize an implementation model of Replicated Data Type. We use the model of Burckhardt et al. [34] that can express existing implementations [16, 32, 85, 92].

Figure 2.2 gives an informal overview of the system components involved in our model. We consider a **replicated database** that hosts one or more named **objects** on a set of replicas. We assume that the database implements each object *independently*. In other words, a database consists of a single object, without loss of generality.[4] The **type signature** of an object defines its interface. An **object implementation** defines its behavior, including its replication protocol.

An application has multiple processes that run **client sessions**. Each session issues a sequence of **operations** to a replica by calling **methods**; for simplicity, we assume that every session matches exactly one logical replica. A database replica delegates method calls to the corresponding object implementation. It responds by performing a local computation step. Local processing ensures that *all* operations are highly available and responsive. Replicas communi-

[4]This assumption matches many models from the literature [16, 85, 92, 102], and recent industrial implementations of RDTs [3, 25]. It also allows us to express, in Part III, a system with independent object implementations yet offering cross-object consistency.

cate in the background via a message passing network layer to exchange updates. An object implementation includes functions that produce and apply messages.

2.2.1 Object Implementation Model

The **type** τ of an object comprises a **type signature** tuple $(\mathsf{Op}_\tau, \mathsf{Val}_\tau)$, which defines the client interface as a set of methods Op_τ (ranged over by o) and a set of return values Val_τ (ranged over by v). For simplicity, all methods share the same return value domain, with a special value $\bot \in \mathsf{Val}_\tau$ for methods that do not return any value. Without loss of generality, we assume that every type has a single side-effect free **query method** $\mathtt{read} \in \mathsf{Op}_\tau$, which returns the value of an object. The remaining methods $\mathsf{Op}_\tau \setminus \{\mathtt{read}\}$ are called **update methods** and are assumed to have side effects.

For example, a counter type, noted $\mathtt{Ctr}$, has a signature $(\mathsf{Op}_{\mathtt{Ctr}}, \mathsf{Val}_{\mathtt{Ctr}})$. Methods $\mathsf{Op}_{\mathtt{Ctr}} = \{\mathtt{inc}, \mathtt{read}\}$, respectively, increment and read the counter value. `read` returns a natural number, $\mathsf{Val}_{\mathtt{Ctr}} = \mathbb{N} \cup \{\bot\}$. Another example is an integer register type, noted $\mathtt{LWWReg}$, with methods $\mathsf{Op}_{\mathtt{LWWReg}} = \{\mathtt{write}(a) \mid a \in \mathbb{Z}\} \cup \{\mathtt{read}\}$. The latter returns integer values, $\mathsf{Val}_{\mathtt{Ctr}} = \mathbb{Z} \cup \{\bot\}$. Note that we model arguments of a method as part of a method name for simplicity.

An object is instantiated from an implementation class of some type, to be defined shortly. An object is hosted on a set of database **replicas** $\mathsf{ReplicaID}$ (ranged over by r). We assume that the database can provide the object implementation with **unique timestamps** from a domain $\mathsf{Timestamp}$ (ranged over t), totally ordered by $<$. Scalar clocks, such as Lamport's logical clock [64], or real-time clock concatenated with a unique replica ID, can serve this purpose. For simplicity, we often assume integer timestamps.

Definition 2.1 (Replicated Data Type Implementation)**.** An **implementation of replicated data type** τ is a tuple $\mathcal{D}_\tau = (\Sigma, M, \mathsf{initialize}, \mathsf{do}, \mathsf{send}, \mathsf{deliver})$, where:

- Set Σ (ranged over by σ) is the domain of replica states.
- Set M (ranged over by m) is the domain of messages exchanged between replicas.
- Function $\mathsf{initialize} : \mathsf{ReplicaID} \to \Sigma$ defines the initial state at every replica.
- Function $\mathsf{do} : \Sigma \times \mathsf{Op}_\tau \times \mathsf{Timestamp} \to \Sigma \times \mathsf{Val}_\tau$ defines methods.
- Functions $\mathsf{send} : \Sigma \to \Sigma \times M$ and $\mathsf{deliver} : \Sigma \times M \to \Sigma$ define update replication primitives.

We call $\mathcal{D}_\tau$ an implementation class of type τ. We refer to the components of a tuple $\mathcal{D}_\tau$ using dot notation, e.g., $\mathcal{D}_\tau.\Sigma$ for its states.

We now discuss the role of implementation components and the intuition behind an execution model, to be formalized later. The database instantiates the object at every replica $r \in \mathsf{ReplicaID}$ by $\mathsf{initialize}(r)$. From that point on, it manages the object at each replica using the three implementation functions explained next.

The client can perform a method $o \in \mathsf{Op}_\tau$ on a selected replica at *any* time. Method invocation causes the replica to perform $\mathsf{do}(\sigma, o, t)$ on the current object state $\sigma \in \Sigma$, with timestamp $t \in$

Timestamp. The timestamp can be used by the implementation to identify an update or to arbitrate updates. The outcome is a tuple $(\sigma', v) = \mathsf{do}(\sigma, o, t)$, where $\sigma' \in \Sigma$ becomes the new object state at the replica invoking the operation, and the operation return value $v \in \mathsf{Val}_\tau$ is presented to the client. Note that for queries $\sigma' = \sigma$. Since method invocation is a local step, the client is never blocked by an external network or replica failure.

Replication functions are applied by a database replica non-deterministically, when the network receives a message or permits a transmission, and the replica decides to perform a replication step. To send a message, a replica applies $\mathsf{send}(\sigma)$ with the local state of the object σ. This produces a tuple $(\sigma', m) = \mathsf{send}(\sigma)$, where σ' is the new state of the object (send may have a side-effect), and m is a message that the replica sends, or *broadcasts*, to all other replicas.[5] When a replica receives a message m, it can apply $\mathsf{deliver}(\sigma, m)$ on the object in state σ, which produces the new state σ' that incorporates message m.

2.2.2 Replication

Data type implementations can be categorized by how they replicate updates, and what requirements they impose on the network layer. In our prior work, Shapiro et al. [92] propose two main **implementation categories**: **operation-based (op-based)** and **state-based**. In op-based implementation, each message carries information about the *latest* updates that have been performed *at the sender* replica. Such message usually must not be lost, and sometimes requires further ordering or no-duplication guarantees. In contrast, in state-based implementation, each message carries information about *all updates* that are known to the sender; such messages can be duplicated, lost or reordered. We will formalize these differences after we familiarize the reader with their intuition. Both categories have the same expressive power, in the sense that every state-based object can be also expressed as an op-based object and vice versa [92].

One of the main challenges are the two expectations behind an object implementation: that the replicas *converge* towards a state that integrates all updates, and that both final and intermediate results provide meaningful *semantics*. This is complicated, because the convergence process driven by asynchronous replication is inherently concurrent itself, and with continuous stream of new updates. Moreover, the network layer may drop, reorder or duplicate messages. To achieve convergence and reasonable semantics, an implementation must often encode additional *metadata* in its state when it performs updates (with do function), so that the replication protocol (send and deliver functions) have enough information to interpret (reconcile) concurrent operations, or to handle network failures.

We formulated sufficient convergence conditions specific to each of the two main implementation categories in a prior work [92]. For an op-based implementation to be convergent, it suffices that all concurrently generated messages *commute*, i.e., they are insensitive to the order of

[5] A broadcast message produced by send may reach more than one replica, but the implementation can emulate a unicast primitive on top of it, by encoding information about the intended recipient inside a message (at the expense of unnecessary message complexity). None of our results is affected by this simplification.

Algorithm 2.1 Template for data type implementation.

```
$\Sigma = \langle$ *definition of state domain* $\rangle$   $M = \langle$ *definition of message domain* $\rangle$
initialize($r_i$) : $\sigma$                                  ▷ method that returns state $\sigma \in \Sigma$
    let $\sigma = \langle$ *definition of the initial state for replica r* $\rangle$
do(read, $t_o$) : $v$                      ▷ timestamp $t_o$ can be optionally used by a method
    let $a = \langle$ *definition of return value* $\rangle$
do(update(arg), $t_o$)                          ▷ update have no return value ($v = \perp$)
    $\sigma \leftarrow \langle$ *definition of state mutation by method* update(arg) $\rangle$
send() : $m$
    let $m = \langle$ *definition of generated message* $m \in M$ $\rangle$
    $\sigma \leftarrow \langle$ *definition of mutation to the local state* $\rangle$
deliver($m$)
    $\sigma \leftarrow \langle$ *definition of the state integrating message m* $\rangle$
```

delivery. For a state-based implementation to be convergent, it suffices that the domain of states Σ form a (partially ordered) *join semilattice*, such that the deliver function is a join operator for that lattice, and update methods can only cause the state to advance in the partial order of the lattice. This typically requires that the object state represents some summary of its history, which allows any two states to be integrated. We will illustrate these conditions alongside the following example implementations, and return formally to the problem when we define a category-agnostic convergence and semantics specification.

2.2.3 Examples

We illustrate the model with simple example implementations from both op-based and state-based categories. They help us to highlight some of the design alternatives and challenges, as well as primitive mechanisms used by implementations, such as clocks. More complex examples will appear later. The selected examples are a small part of our catalog of implementations [93].

Template. Algorithm 2.1 serves as a template for presentation of all data type implementations. The template presents, in pseudo-code, Definition 2.1 in a more readable manner. Text $\langle$*inside brackets*$\rangle$ indicates parts that vary between data type implementations. We organize the code in blocks by indentation. Types of arguments and return values are omitted, but can be easily inferred from the type signature, or the implementation class. We assume all implementation functions have access to the current state. Function can perform in-place mutations that define the state after a function is applied. Components of the state are named in the return value of initialize function. Comments are preceded with ▷ a triangle symbol.

Algorithm 2.2 Op-based implementation of counter (`Ctr`).

```
1: $\Sigma = \mathbb{N}_0 \times \mathbb{N}_0 \quad M = \mathbb{N}_0$
2: initialize($r_i$) : $(a, b)$
3:     let (a, b) = (0, 0)                    ▷ $a$: current value; $b$: buffer of increments
4: do(read, $t_o$) : $v$
5:     let $v = a$
6: do(inc, $t_o$)
7:     $a \leftarrow a + 1$                    ▷ record increment in the current value
8:     $b \leftarrow b + 1$                    ▷ record increment in the buffer
9: send() : $d$
10:    let $d = b$                             ▷ flush the buffer
11:    $b \leftarrow 0$
12: deliver($d$)
13:    $a \leftarrow a + d$                    ▷ add to the local value
```

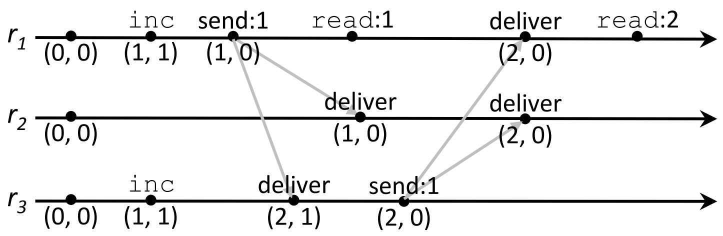


Figure 2.3: Time-space diagram of an execution of the op-based counter (Algorithm 2.2) with three replicas. Every event is labelled with the execution step, the return value of the function performing that step (above line; except for unlabelled initialize events), and replica state after the step if it changed (below line). We indicate messages exchanged between replicas with arrows, thus we omit repeating their content in deliver. Timestamps in do are specified only if used.

2.2.3.1 Counter Implementations

Op-based. Algorithm 2.2 presents an op-based implementation of a counter type (`Ctr`). The state of a replica contains the current counter value, noted a, and an integer buffer that records the number of local increments since the last message was sent, noted b. Each increment operation, `inc`, simply increments both the current value and the buffer. When a replica sends a message, the implementation flushes the buffer, i.e., the local buffer becomes the content of the message and the buffer is zeroed. A replica that delivers a message increments its local counter by the value provided in the message without modifying its local buffer.

Figure 2.3 illustrates both an example execution of the op-based counter, and our presentation format of executions. After all replicas are initialized with the initialize function, clients at replica r_1 and r_3 perform one `inc` each, which is implemented by the database replica applying the do function. The read operation that follows at replica r_1 yields value 1, which includes the outcome of the earlier local increment. The message with update from r_1 is generated using send, and reaches r_2 and r_3 before r_3 flushes his buffer. The message is incorporated using deliver, which

Algorithm 2.3 State-based implementation of counter (`Ctr`).

```
1: $\Sigma = \mathsf{ReplicaID} \times (\mathsf{ReplicaID} \mapsto \mathbb{N}_0)$   $M = \mathsf{ReplicaID} \mapsto \mathbb{N}_0$
2: initialize($r_i$) : $(r, vv)$
3:     **let** $r = r_i$                                        ▷ replica ID
4:     **let** $vv = \lambda s.0$      ▷ vector/map of number of increments per replica; initially zeros
5: do(read, $t_o$) : $v$
6:     **let** $v = \sum vv(s)$                                  ▷ sum up all increments
7: do(inc, $t_o$)
8:     $vv \leftarrow vv[r \mapsto vv(r) + 1]$                    ▷ increment own entry
9: send() : $vv_m$
10:    **let** $vv_m = vv$
11: deliver($vv_m$)
12:    $vv \leftarrow \lambda s.\max\{vv(s), vv_m(s)\}$        ▷ compute entry-wise maximum of vectors
```

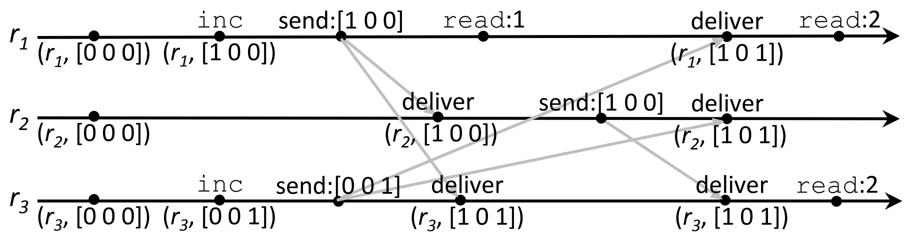


Figure 2.4: An example execution of the state-based counter (Algorithm 2.3) with three replicas. For brevity, we note vector $[r_1 \mapsto a, r_2 \mapsto b, r_3 \mapsto c]$ as $[a\ b\ c]$.

adds its content to the counter value a, but does not touch his buffer b. Eventually, r_3 sends a message, flushing his buffer b, which reaches all remaining replicas. Replicas converge towards value $a = 2$. At the end, a read at replica r_1 yields return value 2.

Intuitively, the op-based counter converge if every replica eventually applies the send function on every non-zero buffer and the network layer delivers every message to every replica *exactly once* [34]; we will formalize these conditions later. The state converges, because the addition operation employed in deliver is *commutative*, i.e., different increments can be delivered at replicas in different order and produce the same effect. Furthermore, the value that the counter replicas converge to is the total number of increments `inc` issued at all replicas [34].

State-based. Let us now consider a state-based implementation of counter, illustrated in Algorithm 2.3. Recall that in state-based implementation a message can be lost, reordered or duplicated, and should contain the complete set of operations known by the sender. A scalar integer is *insufficient* in this case. To see why, consider two replicas that concurrently increment their counter; if the current value of a counter was a scalar, the receiver's replica would have no means of distinguishing whether the received increment is already (partially) known locally or if it is a fresh one. We will demonstrate this impossibility formally, in Part II of the thesis.

Instead, every replica of a state-based counter stores information about all increments

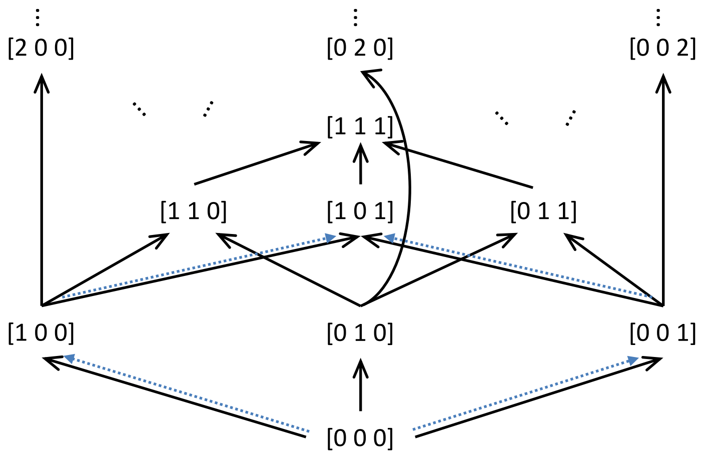


Figure 2.5: Hasse diagram of a fragment of the join-semilattice of states (vectors) of the state-based counter implementation. Solid arrows between states indicate a transitive reduction of $\sqsubset$ order; dotted blue paths indicate replicate state transitions during execution from Figure 2.4. We omit replica's own ID in a state, which is irrelevant for the order, and show only vectors.

it knows about. These can be represented efficiently as a **vector**, formally a map from replica identity to the number of increments, noted $vv : \mathsf{ReplicaID} \to \mathbb{N}_0$, such that each replica increments its own entry only. A value of a counter is simply the sum of all entries in the vector vv. The `inc` operation increments the replica's own entry in the vector (the identity of this entry is recorded as r during the initialization). To replicate, the state-based counter sends its complete vector. The receiving replica computes entry-wise maximum with its own vector.

Figure 2.4 illustrates an execution of the state-based counter. Note that the first increment at replica r_1 reaches replica r_3 both *indirectly* from replica r_2, and later, directly in a delayed message from r_1. Nevertheless, all replicas converge towards a vector $[r_1 \mapsto 1, r_2 \mapsto 1, r_3 \mapsto 0]$, which corresponds to counter value 2.

Baquero and Moura [16], and our later work [92], show that the maximum operator on a vector used by deliver, together with the way the vector is incremented, guarantee convergence of the counter state towards a maximum vector. Moreover, the maximum operator is resilient to message duplication, loss, and out-of-order delivery. This is thanks to the fact that the domain of states (vectors) is a semilattice under the partial order defined as:

(2.1) $$vv \sqsubseteq vv' \iff \mathsf{dom}(vv) \subseteq \mathsf{dom}(vv') \land \forall r \in \mathsf{dom}(vv').vv(r) \leq vv'(r).$$

(We note by $\sqsubset$ a strict variant of this relation.)

Each `inc` update advances the vector in the $\sqsubseteq$ order, whereas the entry-wise maximum operator used by deliver is in fact the *least upper bound* of the semilattice:

(2.2) $$vv \sqcup vv' = \lambda r.\max\{vv(r), vv'(r)\},$$

i.e., it generates the minimum vector that dominates both input vectors. We illustrate a fragment of the lattice on Figure 2.5. Throughout an execution of the implementation, replica states advance

Algorithm 2.4 State-based implementation of last-writer-wins integer register (`LWWReg`).

```
1: $\Sigma = \mathbb{Z} \times \mathsf{Timestamp}$   $M = \mathbb{Z} \times \mathsf{Timestamp}$
2: initialize($r_i$) : $(a, t)$
3:     let $a = 0$                                   ▷ register value
4:     let $t = t_\perp$                             ▷ timestamp; initially $t_\perp = \min \mathsf{Timestamp}$
5: do(read, $t_o$) : $v$
6:     let $v = a$
7: do(write($a_o$), $t_o$)
8:     if $t_o > t$ then                             ▷ sanity check for new timestamp
9:         $(a, t) \leftarrow (a_o, t_o)$            ▷ overwrite
10: send() : $(a_m, t_m)$
11:     let $(a_m, t_m) = (a, t)$
12: deliver($(a_m, t_m)$)
13:     if $t_m > t$ then
14:         $(a, t) \leftarrow (a_m, t_m)$ ▷ overwrite the existing value if the existing timestamp is dominated
```

in this lattice (indicated with dotted paths on the figure), adding each time more information about the operations performed. Replicas traverse the lattice in parallel, while merging states with least upper bound ensures that they converge.

Compared to the op-based implementation, the state of a replica has no designated outgoing buffer, but the whole state is transferred instead. Therefore, state-based implementation *transitively delivers* updates, i.e., a replica may serve as an active relay point (as r_2 does for r_3, in our example). The implementation, however, is more complex.

It is possible to extend this solution to build a counter with an additional decrement method [93], by using a separate vector for decrements. Ensuring strong invariants over a counter, e.g., ensuring that the value remains positive, is more difficult and requires stronger consistency model in the general case.

2.2.3.2 Register Implementations

State-based LWW register. Another RDT example is an object that resolves concurrent updates by *arbitration*, using approach (B) from Section 2.1. Algorithm 2.4 presents a state-based register (`LWWReg`). Without loss of generality, we consider a register that stores integer values.

The register stores both a value, and a unique timestamp generated when the value was written; the latter is not visible to the `read` operation. Concurrent assignments to the register are resolved using the *last-writer-wins* (LWW) policy applied at all replicas: the write with a higher timestamp *wins* and remains visible [56]. LWW guarantees that all replicas eventually select the same value, since the order of timestamp is interpreted in the same way everywhere.

The implementation may use any kind of totally ordered timestamps. However, ideally, every `write` should be provided with a timestamp higher than that of the current value, so it takes effect. A popular implementation is a **logical clock**, or **Lamport clock** [64]. Lamport clock

Algorithm 2.5 Basic state-based implementation of multi-value integer register (`MVReg`).

```
1: $\Sigma = \mathsf{ReplicaID} \times \mathcal{P}(\mathbb{Z} \times (\mathsf{ReplicaID} \hookrightarrow \mathbb{N}_0))$   $M = \mathcal{P}(\mathbb{Z} \times (\mathsf{ReplicaID} \hookrightarrow \mathbb{N}_0))$
2: initialize($r_i$) : $(r, A)$
3:     let $r = r_i$                                                ▷ replica ID
4:     let $A = \emptyset$   ▷ non-overwritten entries: set of pairs $(a, vv)$ with values and version vectors
5: do(read, $t_o$) : $V$
6:     let $V = \{a \mid \exists vv : (a, vv) \in A\}$        ▷ all stored values are the latest concurrent writes
7: do(write($a$), $t_o$)
8:     let $vv = \bigsqcup\{vv' \mid (\_, vv') \in A\}$        ▷ compute entry-wise maximum of known vectors
9:     let $vv' = vv[r \mapsto (vv(r) + 1)]$                  ▷ increment own entry to dominate other vectors
10:    $A \leftarrow \{(a, vv')\}$                              ▷ replace the current value with the new entry
11: send() : $A_m$
12:    let $A_m = A$
13: deliver($A_m$)
14:    $A \leftarrow \{(a, vv) \in A \cup A_m \mid \nexists (a', vv') \in A \cup A_m : vv \sqsubset vv'\}$   ▷ keep non-overwritten entries
```

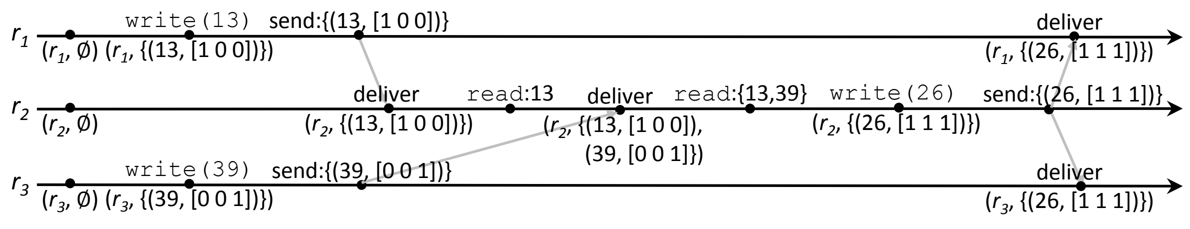


Figure 2.6: An example execution of the basic state-based multi-value register (Algorithm 2.5).

is a pair $(k, r) \in \mathbb{N} \times \mathsf{ReplicaID}$, where k is a natural number, and r is a replica ID. The order of timestamps is defined on natural numbers first, and replica IDs serve as tie-breakers, i.e., $(k, r) < (k', r') \iff k < k' \vee (k = k' \wedge r < r')$. To generate a timestamp, replica r_i computes the highest number k it has observed in all timestamps so far, noted k_{max}, and assigns the generated timestamp value $(k_{max} + 1, r_i)$. The new timestamp is fresh and dominates the observed ones.

The implementation of LWW is simple, but as we illustrated in Figure 2.1B, LWW implies that arbitrary assignments are entirely *lost*, despite being previously accepted.

State-based multi-value register. A popular alternative to LWW register is a multi-value register (`MVReg`), implementing conflict-detection approach (C), which does not lose updates. The multi-value register stores *all non-overwritten* values, and lets the application decide how to resolve them. Thus, if there are concurrent updates, it returns a set of values, $\mathsf{Val}_{\mathtt{MVReg}} = \mathcal{P}(\mathbb{Z})$, rather than a single one.

Practical state-based implementations of multi-value register encode information about overwritten values compactly, using a variation of **version vectors** by Parker et al. [77]. A version vector has the same structure as the vector we described in the counter implementation: it is a map from replica ID to the number of observed updates. The usage of a version vector is more complex, and so is its semantics. In counter, a vector is associated directly with a state and describes the number of increment events observed from each replica. In multi-value register, a vector is associated with a value written in a certain state and represents the number of operations from each replica that value *overwrites*. Equivalently, it represents the *knowledge* of the replica that wrote the value, in terms of the number of observed writes from each replica. For instance, a vector $vv = \{r_1 \mapsto 3, r_2 \mapsto 5\}$ associated to a value indicates that the value overwrites the first three writes from replica r_1, and the first five from replica r_2.

The intended behavior of multi-value register is to overwrite a value only if the writer of the new value has observed (or could have observed) the overwritten value. To this end, the implementation interprets the partial order $\sqsubseteq$ that vectors induce (Equation 2.1) as *potential causality* between operations [64, 90]. If vectors of two values are ordered by $vv_1 \sqsubseteq vv_2$, then they are causally related: the operation with vv_1 has been visible at the replica that issued the operation with vv_2, and could have caused or impacted the latter. Absence of causality between operations indicates *logical concurrency*.

Algorithm 2.5 presents a multi-value register implementation based on a standard version vector algorithm [1, 44, 59, 93]. The state of a replica contains local replica ID r, used to assign vectors, and a set of entries A, pairs of values and their vectors. The entries represent the set of non-overwritten values; formally, the entries represent an *antichain* of the causality relation induced by $\sqsubseteq$. Thus, `read` simply returns all values of the entries.

`write` operation overwrites the set of observed values. To this end, replica needs to produce a vector that dominates them in causality. The replica applies the least upper bound operator $\sqcup$ on all observed vectors (Equation 2.2), and increments the replica's own entry in the vector, to strictly dominate observed vectors, according to $\sqsubset$. The generated vector is assigned to the written value and replaces the existing set of entries.

The replication protocol sends all entries. The receiving replica integrates local entries with the received ones. It computes the new antichain, without losing any updates, as a union of local and received entries with eliminated overwritten entries, according to their version vectors.

Figure 2.6 illustrates an example execution of the implementation. Clients at replicas r_1 and r_3 perform concurrent writes of values 13 and 39, respectively. Replica r_2 receives their updates, and identifies that these are writes that do not overwrite one another, since their vectors are incomparable according to $\sqsubseteq$. Therefore, the read at r_2 returns $\{13, 39\}$. The client at r_2 subsequently overwrites the two values with value 26; the vector it assigns dominates known vectors. Therefore, when it propagates his updates, his entry for value 26 overwrites existing entries at r_1 and r_3.

Although vectors are a powerful concept that prevents data loss, they occupy space, in the order of the number of replicas. As we shall see in Part II, for some types this cost is unavoidable.

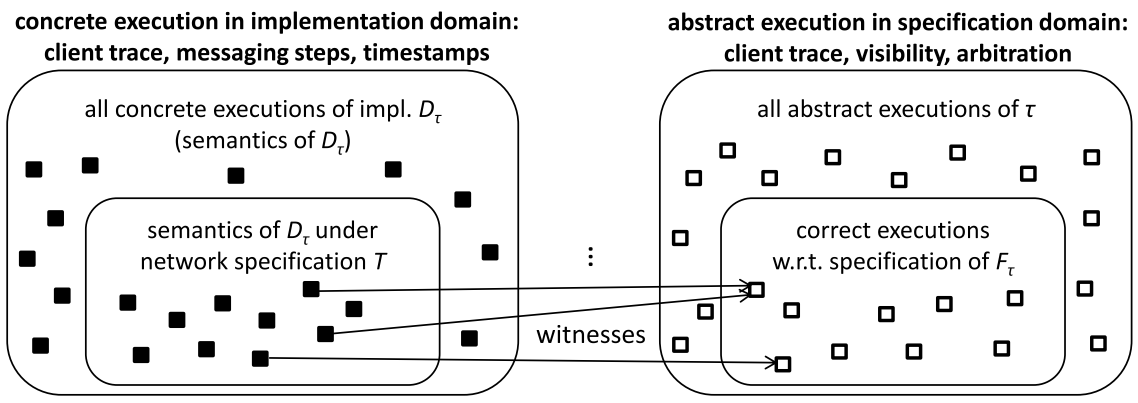

concrete execution in implementation domain: client trace, messaging steps, timestamps
all concrete executions of impl. D_τ (semantics of D_τ)
semantics of D_τ under network specification T
abstract execution in specification domain: client trace, visibility, arbitration
all abstract executions of τ
correct executions w.r.t. specification of F_τ
witnesses

Figure 2.7: Components of implementation and specification models, and their relation through correctness definition. Squares (resp. empty squares) indicate concrete (resp. abstract) executions.

2.3 Specification of Intended Behavior

The primary expectation behind any replicated object is convergence. Category-specific convergence conditions can be successfully applied to implementations [92], including the examples from the previous section. However, convergence alone is not sufficient. Towards what value the object converges, and what are the intermediate states and their values, is also important. For instance, an increment-only counter that provably converges towards 0, or occasionally returns value 0 before it converges, does not provide the intended behavior.

A precise specification of *type semantics* can act as a contract between an application and a data type implementation. There can be multiple implementations of the same type, including different optimizations or models (e.g., Algorithm 2.2 vs. Algorithm 2.3). It would be natural to express that these implementations *behave* in the same way. Implementation is often not an effective way of expressing semantics, since it requires considering low-level details of implementation and execution (consider, for example, Algorithm 2.5, or an execution from Figure 2.4).

In this section, we define and illustrate a declarative RDT specification model, after Burckhardt et al. [34]. This form of specification offers a unified and concise representation of RDT semantics, that embodies the convergence condition.

2.3.1 Specification Model

To introduce the specification model we rely on the reader's intuitive understanding of the execution model from the previous section. At a high level, our goal is to define a minimal *abstract* description of each concrete execution that would express the behavior of an implementation, and that would justify its correctness. We give an overview of the components involved in the semantic description model and the definition of correctness condition in Figure 2.7. On the left side, we illustrate sets of concrete executions in the low-level implementation domain. The specification domain, on the right, is made of abstract executions, including a trace of client operations with additional relations. A data type specification is defined on abstract execution structure, and indirectly identifies a subset of abstract executions that satisfy the specification. Our ultimate goal is to define a correctness condition for a data type implementation, that relates each concrete execution with a witness abstract execution that satisfies the specification.

The client-observable behavior of an execution can be characterized by a **trace** that carries information on operations performed by each client session, the order in which they were issued by a client session, and their return values. The intended behavior concerns return values in a trace. However, a trace alone is insufficient to specify the intended behavior, because it is missing information about relation of operations performed in *different* client sessions (at different replicas). For example, it is impossible to determine if a counter implementation behaves correctly if the operations in the trace do not specify what increments were delivered to a replica of the client that performed the operation. A trace must be extended with additional information.

Sequential specification. In a strongly-consistent setting, objects are, roughly speaking, guaranteed to execute operations in the same sequence at all replicas. Therefore, the intended behavior of type τ can be expressed as a sequential specification function $\mathcal{S}_\tau : \mathsf{Op}_\tau^+ \to \mathsf{Val}_\tau$, which specifies the return value of the last operation in a provided sequence [34]. A trace extended with a sequence of operations from the execution can be evaluated against a sequential specification.

For example, the sequential specification of a counter can be defined as [34]:

$$\mathcal{S}_{\texttt{Ctr}}(\xi\ \texttt{inc}) = \bot;$$
$$\mathcal{S}_{\texttt{Ctr}}(\xi\ \texttt{read}) = a \quad \text{where } a \text{ is the number of } \texttt{inc} \text{ operations in } \xi,$$

where $\xi \in \mathsf{Op}_{\texttt{Ctr}}^+$ is any sequence of operations. Similarly, the semantics of a sequential register is:

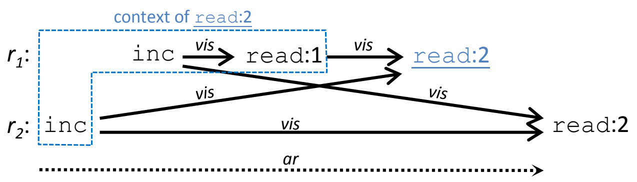
$$\mathcal{S}_{\texttt{LWWReg}}(\xi\ \texttt{write}) = \bot;$$
$$\mathcal{S}_{\texttt{LWWReg}}(\xi\ \texttt{read}) = a \quad \text{iff the last operation in } \xi|_{\texttt{write}} \text{ is } \texttt{write}(a), \tag{2.3}$$

where $\xi \in \mathsf{Op}_{\texttt{LWWReg}}^+$, and $\xi|_{\texttt{write}}$ is a restriction of ξ to `write` operations.

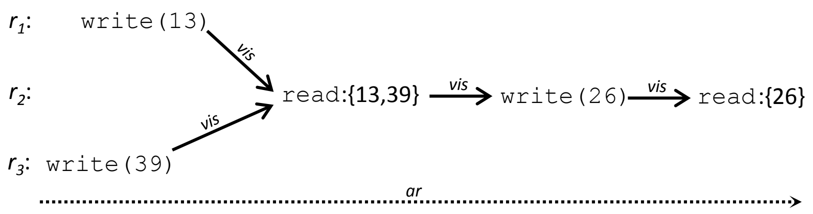
A client trace, extended with the sequence order, is correct if it satisfies the above specification. Intuitively, a sequence with `write`(1) operation, followed by `read` that returns 1, is correct w.r.t. $\mathcal{S}_{\texttt{LWWReg}}$, but a sequence with `write`(2), followed by a `read` returning 1 is not.

Concurrent specification. In the case of RDTs, the order of operations may not be the same across replicas. Thus, a specification have more elaborate structure. Burckhardt et al. [34] identify two complementary abstract orders that can impact the return value of an RDT operation:

1. A set of operations that are *visible* to the operation, and recursively, a set of operations that was visible to them. In terms of implementation, visibility describes what operations were delivered to the replica that performed the operation under consideration. The visibility



(A) Witness abstract execution for concrete execution of $\mathcal{D}_{\texttt{Ctr}}$ from Figure 2.4.



(B) Witness abstract execution for concrete execution of $\mathcal{D}_{\texttt{MVReg}}$ from Figure 2.6.

Figure 2.8: Examples of abstract executions for `Ctr` and `MVReg` data types. We indicate transitive reduction of visibility by arrows, arbitration by horizontal position of events, and replica order by their vertical level. Dashed blue box indicates the context of the underlined blue operation.

abstracts away the details of the means of delivery. (e.g., what replication protocol category and metadata were involved, what path that update followed through replicas until it was delivered, whether there were duplicated messages delivered, lost messages, etc.)

2. An order in which concurrent operations are *arbitrated*. Arbitration abstracts away the implementation of timestamps.

Formally, an *abstract execution* describes the client-observable effects of a concrete execution (trace) with additional visibility and arbitration information, as follows.

Definition 2.2 (Abstract Execution)**.** An **abstract execution** of a data type τ is a tuple $A = (E, \mathsf{repl}, \mathsf{op}, \mathsf{rval}, \mathsf{ro}, \mathsf{vis}, \mathsf{ar})$, where:

- $E \subseteq \mathsf{Event}$ is a set of **events** from a countable universe Event.
- Each event $e \in E$ represents a replica $\mathsf{repl}(e) \in \mathsf{ReplicaID}$ performing an operation $\mathsf{op}(e) \in \mathsf{Op}_\tau$ that returned value $\mathsf{rval}(e) \in \mathsf{Val}_\tau$.
- $\mathsf{ro} \subseteq E \times E$ is a **replica order**, which is a union of strict total prefix-finite orders on events at each replica.
- $\mathsf{vis} \subseteq E \times E$ is an acyclic prefix-finite **visibility relation**.
- $\mathsf{ar} \subseteq E \times E$ is a strict total prefix-finite **arbitration relation**.

We require that $\mathsf{ro} \cup \mathsf{vis}$ is acyclic, as we do not consider speculative systems [34], and assume that local updates are immediately visible, i.e., $\mathsf{ro} \subseteq \mathsf{vis}$. Pairs of events that are not ordered by visibility are said **concurrent**. Visibility is often a *partial order*; we assume this in our examples.

We give two examples of abstract executions in Figure 2.8, which characterize concrete executions of counter and multi-value register implementations from the previous section.[6]

Specification of a data type is defined on each event in an abstract execution and its *context*, i.e., the set of events visible to that event.

Definition 2.3 (Operation Context)**.** An **operation context** of operation $o \in \mathsf{Op}_\tau$ for a data type τ is a tuple $L = (o, E, \mathsf{op}, \mathsf{vis}, \mathsf{ar})$, where: $E \subseteq \mathsf{Event}$ is a finite subset of Event; $\mathsf{op} : E \to \mathsf{Op}_\tau$ defines type of operations in E; $\mathsf{vis} \subseteq E \times E$ is an acyclic visibility relation; $\mathsf{ar} \subseteq E \times E$ is a strict total arbitration relation. We note $\mathsf{Context}_\tau$ the domain of contexts for type τ that obey this structure.

For abstract execution A, the context of event $e \in A.E$ can be extracted with context function:

$$\mathsf{context}(A, e) = (A.\mathsf{op}(e), E_e, A.\mathsf{op}|_{E_e}, A.\mathsf{vis}|_{E_e}, A.\mathsf{ar}|_{E_e}),$$

where $E_e = \mathsf{vis}^{-1}(e)$ are the events in E visible to e, and $R|_{E_e}$ indicates restriction of relation R to the elements of E_e. For example, the context of the underlined blue `read` in Figure 2.8A is indicated with a blue dashed line.

Definition 2.4 (Specification)**.** A **specification** for a data type τ is a partial function $\mathcal{F}_\tau : \mathsf{Context}_\tau \rightharpoonup \mathsf{Val}_\tau$ that given an operation context for type τ specifies a return value.

By applying the specification to every event in an abstract execution, we obtain a correctness condition for abstract executions.

Definition 2.5 (Correct Abstract Execution)**.** An abstract execution A of type τ **satisfies a specification** $\mathcal{F}_\tau$, noted $A \models \mathcal{F}_\tau$, if the return value of every operation in A is the one computed by the specification applied to the context of that operation: $\forall e \in A.E : A.\mathsf{rval}(e) = \mathcal{F}_\tau(\mathsf{context}(A, e))$.

Specification defines the expected value for a given context in a *deterministic* way. The non-determinism of a distributed execution is encapsulated in client invocations, and visibility and arbitration relations of an abstract execution.

Note that Definition 2.5 embeds the *safety* aspect of the *convergence* requirement, which we call **confluence**. Confluence requires that if two operations observe the same set of operations, they must return the same value. Deterministic specification embodies this condition, and covers implementation category-specific conditions, such as semilattice of states or commutativity of operations. For eventual convergence, it remains to require the *liveness* aspect of convergence, which we call **eventual visibility** of operations. Formally, we state it as a restriction on an abstract execution A, that requires that operations are not infinitely invisible [34]:

$$\forall e \in A.E : \nexists \text{ infinitely many } f \in A.E \text{ s.t. } f \overset{\mathsf{vis}}{\not\longrightarrow} e, \tag{2.4}$$

where $e \xrightarrow{\mathsf{vis}} f$ indicates that e and f are ordered by vis relation.

[6]In the general case more than one abstract execution may characterize a single concrete execution.

Other desirable system properties that we do not discuss here, such as cross-object consistency or session guarantees, can be also formulated as a condition on abstract executions [34].

Every type specification has an implementation (provided it does not return abstract events E, invisible to the implementation). Intuitively, a naive state-based implementation of a specification stores a complete graph of visibility and arbitration, and implements `read` as $\mathcal{F}_\tau$.

2.3.2 Examples

We illustrate the specification model with a few examples from Bieniusa et al. [23], Burckhardt et al. [34], and a new one, covering the implementations from Section 2.2. These are primitive data types that could be *composed* to build more complex objects [52]; we do not study composition in this work.

2.3.2.1 Counter Specification

The specification of a **counter** (`Ctr`) is given by:

(2.5)
$$\mathcal{F}_{\texttt{Ctr}}(\texttt{read}, E, \mathsf{op}, \mathsf{vis}, \mathsf{ar}) = |\{e \in E \mid \mathsf{op}(e) = \texttt{inc}\}|;$$
$$\mathcal{F}_{\texttt{Ctr}}(\texttt{inc}, E, \mathsf{op}, \mathsf{vis}, \mathsf{ar}) = \bot,$$

i.e., read should return the number of all visible increment operations. For example, an abstract execution from Figure 2.8A satisfies specification $\mathcal{F}_{\texttt{Ctr}}$ for every `read` since the number of visible `inc` operations matches the return value.

Specification $\mathcal{F}_{\texttt{Ctr}}$ characterizes both Algorithm 2.2 and Algorithm 2.3 [34], as we formalize later. However, although it is a reasonable and popular counter semantics [3, 5, 92], this is not the only possible one. For example, one could imagine an alternative, where multiple concurrent increments count as one.

Counter specification does not make use of the arbitration relation. Indeed, none of its implementations (Algorithm 2.2 and Algorithm 2.3) uses timestamps.

2.3.2.2 Register Specifications

Last-writer-wins register. The specification of **LWW register** (`LWWReg`), characterizing Algorithm 2.4 [34], is defined by:

(2.6)
$$\mathcal{F}_{\texttt{LWWReg}}(o, E, \mathsf{op}, \mathsf{vis}, \mathsf{ar}) = \mathcal{S}_{\texttt{LWWReg}}(E^{\mathsf{ar}} o),$$

where E^{ar} is a sequence of events in E ordered by arbitration. Thus, the specification is identical to the sequential specification from Equation 2.3, applied to a context of operation linearly ordered by arbitration. It does not make use of the visibility relation between events in a context.

Relying on sequential specification is appealing in its simplicity (it could also express the counter specification $\mathcal{F}_{\texttt{Ctr}}$), but in the general case this specification pattern cannot express all semantics alternatives that do not lose arbitrary updates, as the following examples demonstrate.

Multi-value register. The specification of a **multi-value register** (MVReg), characterizing Algorithm 2.5 [34], is given by:

$$\mathcal{F}_{\texttt{MVReg}}(\texttt{read}, E, \mathsf{op}, \mathsf{vis}, \mathsf{ar}) = \{a \mid \exists e \in E : \mathsf{op}(e) = \texttt{write}(a) \wedge \nexists e' \in E : e' = \texttt{write}(b) \wedge e \xrightarrow{\mathsf{vis}} e'\}. \tag{2.7}$$

(Hereafter, we omit specification of methods that do not return any value.)

Here, visibility acts as a *causality* relation. The read operation returns values of all the writes that are not dominated in visibility by writes of other values.[7]

Specification of data types that use visibility structure, like multi-value register, is difficult to interpret if the visibility is not a partial (transitive) order [72]. Therefore, database implementations often enforce transitivity of visibility for these objects, as we will discuss shortly.

Value-wins. In between LWW and multi-value registers, we propose a new **value-wins register** (VWReg), which resolves concurrent assignments using a predefined order on values. Assuming a totally ordered domain of values, such as integers, the signature of object is the same as the LWW register, $(\mathsf{Op}_{\texttt{VWReg}}, \mathsf{Val}_{\texttt{VWReg}}) = (\mathsf{Op}_{\texttt{LWWReg}}, \mathsf{Val}_{\texttt{LWWReg}})$. We define the specification as follows:

$$\begin{aligned}\mathcal{F}_{\texttt{VWReg}}(\texttt{read}, E, \mathsf{op}, \mathsf{vis}, \mathsf{ar}) = \max\{a \mid \exists e \in E : \mathsf{op}(e) = \texttt{write}(a) \\ \wedge \nexists e' \in E : e' = \texttt{write}(b) \wedge e \xrightarrow{\mathsf{vis}} e'\}.\end{aligned} \tag{2.8}$$

Value-wins register can express *data-driven* conflict resolution. A specific case of a value-wins register is a *false-wins flag* and a *true-wins flag* [3]. We shall see that some set types rely on a similar approach.

If the order on the domain of values is partial, a similar extension to the multi-value register is feasible. This could reduce the number of conflicts to resolve by the application (user).

2.3.2.3 Set Specifications

The set is an example of a primitive data type for which a concurrent specification is nontrivial. Many alternatives are possible. As we shall see in Part II, the implementations are also complex.

Without loss of generality, we consider a set of integers, where element can be added or removed, $\mathsf{Op}_{\texttt{Set}} = \{\texttt{read}\} \cup \{\texttt{add}(a) \mid a \in \mathbb{Z}\} \cup \{\texttt{rem}(a) \mid a \in \mathbb{Z}\}$, and read returns $\mathsf{Val}_{\texttt{Set}} = \mathcal{P}(\mathbb{Z}) \cup \{\bot\}$.

In our prior work, Bieniusa et al. [23] recommend that any concurrent semantics should respect the sequential behavior as much as possible, in the sense that, for executions where the sequential specification applied to all linear extensions of visibility produce the same return value, that same value should be returned in the concurrent case. Some operations of a set are independent, such as operations on different elements, or idempotent, such as adding (removing) the same element twice, or removing it twice. For these independent or idempotent operations, a concurrent semantics could reduce to the sequential one.

[7] Alternatively, an extended specification could also return a multiset of values to indicate to the application when the same value was written concurrently [77, 93].

Concurrent add and remove on the same element are problematic. Indeed, they have no obvious sequential counterpart — according to sequential specification, the outcome depends on the order. The following three set specifications are popular convergent heuristics that treat this case in different ways, and otherwise rely on the common design principle of Bieniusa et al. [23].

The **LWW set** is specified as follows [23, 56]:

$$\mathcal{F}_{\texttt{LWWSet}}(\texttt{read}, E, \mathsf{op}, \mathsf{vis}, \mathsf{ar}) = \mathcal{S}_{\texttt{Set}}(E^{\mathsf{ar}}\texttt{read}), \tag{2.9}$$

where LWWSet is a type with the same signature as Set, $\mathcal{S}_{\texttt{Set}}$ is a standard sequential specification of set, and E^{ar} is defined as before. Concurrent operations on the same element are resolved according to timestamp arbitration. The weakness of this approach is that it loses some arbitrary concurrent updates on a same element, similarly to LWW register.

An alternative is to give priority to one of the operation types. A popular choice is the **add-wins set** semantics (also known as observed-remove) defined as [3, 23, 34, 92]:

$$\mathcal{F}_{\texttt{AWSet}}(\texttt{read}, E, \mathsf{op}, \mathsf{vis}, \mathsf{ar}) = \{a \mid \exists e \in E : \mathsf{op}(e) = \mathsf{add}(a) \land \nexists e' \in E : \mathsf{op}(e') = \texttt{rem}(a) \land e \xrightarrow{\mathsf{vis}} e'\}, \tag{2.10}$$

where AWSet has the same signature as Set. In add-wins, whenever there are concurrent operations on the same element a, add(a) operation wins and “cancels” the effects of every visible or concurrent rem(a) operation. Note that add-wins set specification could be also viewed as a composition of presence flags (VWReg), one for each element, where a positive value wins.

The opposite approach is known as **remove-wins set** [23]:

$$\mathcal{F}_{\texttt{RWSet}}(\texttt{read}, E, \mathsf{op}, \mathsf{vis}, \mathsf{ar}) = \{a \mid E|_{\mathsf{add}(a)} \neq \emptyset \land \forall e \in E|_{\texttt{rem}(a)} : \exists e' \in E|_{\mathsf{add}(a)} : e \xrightarrow{\mathsf{vis}} e'\}, \tag{2.11}$$

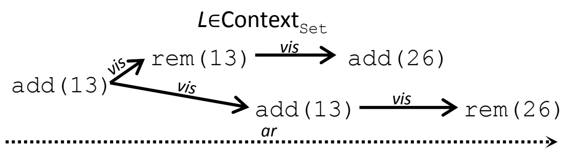
where RWSet has the same signature as Set, and $E|_o$ is the restriction of events E to operations o. Element a is in the remove-wins set if all rem(a) operations are covered in visibility by some visible add(a) operation.

Note that the set of covering add operations does not need to be the same for all removes. Otherwise, the set would have an anomaly that violates the principle of Bieniusa et al. [23]: consider two clients that concurrently perform operations rem(a);add(a) for the same element a; although initially *both* clients observe element a in the set, after they synchronize, a disappears.

Both $\mathcal{F}_{\texttt{AWSet}}$ and $\mathcal{F}_{\texttt{RWSet}}$ definitions make nontrivial use of the visibility relation, since their policy is applied only to concurrent operations. In the case of operations ordered by visibility, the sequential semantics applies.

An example operation context in Figure 2.9 illustrates the three discussed set specifications applied to read operation. The LWW set returns {13}, because add(13) $\xrightarrow{\mathsf{ar}}$ rem(13) $\xrightarrow{\mathsf{ar}}$ add(13), whereas add(26) $\xrightarrow{\mathsf{ar}}$ rem(26). The add-wins set returns {13, 26}, because both rem(a) operations are concurrent in visibility with respective add(a) operations. Conversely, the remove-wins returns an empty set.

Other alternatives include the **counting set** of Aslan et al. [10] and of Sovran et al. [95]. Here, an element is present if the difference between the number of visible add and remove



$$\mathcal{F}_{\texttt{AWSet}}(L,\texttt{read}) = \{13,26\}$$
$$\mathcal{F}_{\texttt{RWSet}}(L,\texttt{read}) = \emptyset$$
$$\mathcal{F}_{\texttt{LWWSet}}(L,\texttt{read}) = \{13\}$$

Figure 2.9: Set semantics alternatives illustrated on an operation context of `read`, with different return value specifications. We show only transitive reduction of partial visibility order.

operations is positive. This approach has anomalous semantics, as it violates the principle of Bieniusa et al. [23], similarly to the example that motivated remove-wins definition. Finally, an alternative is to set a **conflict flag** in case of concurrent $\mathtt{add}(a)$ and $\mathtt{rem}(a)$ [23] (cf. approach (C) from Section 2.1).

2.4 Execution Model and Correctness

In this section, we formalize a concrete execution model (Section 2.4.1), and relate concrete executions and abstract executions with an implementation correctness definition (Section 2.4.2), after Burckhardt et al. [34]. The correctness condition is parameterized with a pair of conditions, which leads us to a formal definition of some implementation categories (Section 2.4.3). The reader interested only in Part III of the thesis may proceed directly to Section 2.4.3.

2.4.1 Execution Model

We model a **concrete execution** of an implementation $\mathcal{D}_\tau$ as a sequence of **transitions over configurations**, according to the definitions of Figure 2.10. A configuration is a pair $(R,N) \in$ Config, which describes the state of a database: R maps each **r**eplica to an object state, and N maps all messages that were sent over **n**etwork and can be delivered, identified by their message ID, $mid \in$ MessageID. The initial empty configuration (R_0,N_0) starts with the initial state of an object and no sent messages. A transition relation $\longrightarrow_{\mathcal{D}_\tau}$: Config × Event × Config describes an atomic execution step. An execution can be visualized as a finite or infinite sequence of transitions:

$$(R_0,N_0) \xrightarrow{e_1}_{\mathcal{D}_\tau} (R_1,N_1) \xrightarrow{e_2}_{\mathcal{D}_\tau} \ldots \xrightarrow{e_k}_{\mathcal{D}_\tau} (R_k,N_k)\ldots,$$

where every transition $\xrightarrow{e_i}_{\mathcal{D}_\tau}$ is explicitly labelled by a unique event e_i and implicitly parameterized by functions that record information about event e_i, such as action(e_i), replica(e_i), etc.

There are three types of configuration transitions, corresponding to the three rules of Figure 2.10. In the first rule, replica r performs an operation o with timestamp t that transforms state $R(r)$ and yields a return value v, as computed by the implementation function do. We record this information using functions replica, action, op, timestamp, and rval, respectively. In the second rule, replica r sends a message m created by the implementation function send, which transforms the state $R(r)$, and puts a message into the map of sent messages $N(mid) = m$, using

Configurations:

$$\begin{array}{rcll} \mathsf{RState} & = & \mathsf{ReplicaID} \to \mathcal{D}_\tau.\Sigma & R_0 = \mathcal{D}_\tau.\mathsf{initialize} \\ \mathsf{NState} & = & \mathsf{MessageID} \rightharpoonup \mathcal{D}_\tau.M & N_0 = [] \\ \mathsf{Config} & = & \mathsf{RState} \times \mathsf{NState} & (R_0, N_0) \in \mathsf{Config} \end{array}$$

Transitions over configurations:

$$\frac{\mathcal{D}_\tau.\mathsf{do}(o,\sigma,t) = (\sigma',v) \quad \mathsf{action}(e) = \mathsf{do} \quad \mathsf{replica}(e) = r \quad \mathsf{op}(e) = o \quad \mathsf{timestamp}(e) = t \quad \mathsf{rval}(e) = v}{(R[r \mapsto \sigma], N) \xrightarrow{e}_{\mathcal{D}_\tau} (R[r \mapsto \sigma'], N)}$$

$$\frac{\mathcal{D}_\tau.\mathsf{send}(\sigma) = (\sigma',m) \quad \mathsf{action}(e) = \mathsf{send} \quad \mathsf{replica}(e) = r \quad mid \notin \mathsf{dom}(N) \quad \mathsf{msg}(e) = mid}{(R[r \mapsto \sigma], N) \xrightarrow{e}_{\mathcal{D}_\tau} (R[r \mapsto \sigma'], N[mid \mapsto m])}$$

$$\frac{\mathcal{D}_\tau.\mathsf{deliver}(\sigma,m) = \sigma' \quad \mathsf{action}(e) = \mathsf{deliver} \quad \mathsf{replica}(e) = r \quad \mathsf{msg}(e) = mid}{(R[r \mapsto \sigma], N[mid \mapsto m]) \xrightarrow{e}_{\mathcal{D}_\tau} (R[r \mapsto \sigma'], N[mid \mapsto m])}$$

Figure 2.10: Definitions of the set of configurations and the transition relation for a data type implementation $\mathcal{D}_\tau$. $(R,N) \in \mathsf{Config}$ is a configuration, and (R_0,N_0) is the initial configuration.

some fresh identifier mid. In the third rule, replica r delivers message $N(mid)$ from the map of sent messages N, which mutates its state $R(r)$ according to the implementation function deliver.

Note that the transitions put no restriction whatsoever on when the first two rules can be applied; the third rule requires only that every delivered message must have been previously sent.[8] Otherwise, there is no restriction on message delivery: messages can be lost, duplicated, and reordered. We introduce these optional restrictions separately in Section 2.4.2.

We now define concrete execution, similarly to the abstract one.

Definition 2.6 (Concrete Execution)**.** A **concrete execution** of a database with an implementation $\mathcal{D}_\tau$ of a type τ is a tuple $C = (E, \mathsf{eo}, \mathsf{pre}, \mathsf{post}, \mathsf{action}, \mathsf{replica}, \mathsf{op}, \mathsf{timestamp}, \mathsf{rval}, \mathsf{msg})$, where:

- $E \subseteq \mathsf{Event}$ is a countable subset of events.
- $\mathsf{eo} \subseteq E \times E$ is a strict total prefix-finite **execution order** of transition events.
- $\mathsf{pre}, \mathsf{post} : E \to \mathsf{Config}$ define, resp., **predecessor** and **successor configurations**, s.t.: $\forall e \in E : \mathsf{pre}(e) \xrightarrow{e}_{\mathcal{D}_\tau} \mathsf{post}(e)$, and $\forall e,f \in E : e \xrightarrow{\mathsf{eo}} f \land (\nexists e \xrightarrow{\mathsf{eo}} g \xrightarrow{\mathsf{eo}} f) \implies \mathsf{post}(e) = \mathsf{pre}(f)$.
- The remaining are partial **labelling functions**: action tells a type of event; timestamp, op and rval are defined for do events, where timestamp is injective to guarantee uniqueness; $\mathsf{msg} : E \rightharpoonup \mathsf{MessageID}$ is defined for send and deliver events.

We denote the **initial configuration** of C by $\mathsf{init}(C) = C.\mathsf{pre}(e_0)$; if C is finite, we note the **final configuration** by $\mathsf{final}(C) = C.\mathsf{post}(e_k)$, where e_k is the last event.

[8]This translates into *no creation* property of network links, in a distributed systems parlance [36].

2.4.2 Implementation Correctness

The semantics of an implementation can be characterized by all of its executions.

Definition 2.7 (Implementation Semantics)**.** The semantics of an implementation $\mathcal{D}_\tau$, noted $[\![\mathcal{D}_\tau]\!]$, is the set of all its concrete executions that start in the empty configuration (R_0, M_0).

Some implementations (or categories of implementations) may not provide a meaningful behavior under all executions, but only when certain network layer conditions are met. For example, the op-based counter from Algorithm 2.2 requires operations to be delivered exactly once, rather than an arbitrary number of times. Similarly, the LWW register may require a condition on the order of supplied timestamps. We express such restrictions as a **network layer specification**, noted $\mathcal{T}$, a set of allowed concrete executions defined by a *condition* on concrete executions.[9] Therefore, when considering the correctness of an implementation, we will reason about $[\![\mathcal{D}_\tau]\!] \cap \mathcal{T}$, i.e., the semantics of an implementation $\mathcal{D}_\tau$ *under* network specification $\mathcal{T}$.

In order to state a correctness condition, we would like to relate each concrete execution with a correct **witness abstract execution**, i.e., to find a correct representation of a concrete execution in the specification domain. Both types of executions, given by Definitions 2.2 and 2.6, share a similar structure, and most components of a witness execution can be directly extracted from the concrete execution. Thus, we define a witness execution for concrete execution $C \in [\![\mathcal{D}_\tau]\!] \cap \mathcal{T}$ as:

$$\mathsf{abs}(C, \mathcal{V}) = (C.E|_{\mathsf{do}}, E.\mathsf{replica}|_{\mathsf{do}}, E.\mathsf{op}|_{\mathsf{do}}, E.\mathsf{rval}|_{\mathsf{do}}, \mathsf{ro}(C)|_{\mathsf{do}}, \mathcal{V}(C), \mathsf{ar}(C)), \tag{2.12}$$

where components are restricted to client-observable do events of C, and replica order $\mathsf{ro}(C)$ and arbitration order $\mathsf{ar}(C)$ are extracted from C, using information about replicas and timestamps:

$$e \xrightarrow{\mathsf{ro}(C)} f \iff e \xrightarrow{C.\mathsf{eo}} f \land C.\mathsf{replica}(e) = C.\mathsf{replica}(f)$$
$$e \xrightarrow{\mathsf{ar}(C)} f \iff e, f \in C.E|_{\mathsf{do}} \land C.\mathsf{timestamp}(e) < C.\mathsf{timestamp}(f)$$

The remaining witness component $\mathcal{V}(C)$ offers some freedom in the witness selection: visibility order. A concrete execution specifies the delivery of messages and operation return values, but not which operations must be visible after message delivery. This parameter varies across implementation categories, and we model it as a **visibility witness** function. A visibility witness takes a concrete execution C, and produces the visibility order $\mathcal{V}(C)$ of its witness abstract execution. We will give some examples of universal witness functions shortly. For a concrete application of visibility witness, compare an execution in Figure 2.4 with the visibility in Figure 2.8A.

This leads us to the final definition of implementation correctness.

Definition 2.8 (Implementation Correctness)**.** A data type implementation $\mathcal{D}_\tau$ **satisfies a type specification** $\mathcal{F}_\tau$ w.r.t. network specification $\mathcal{T}$ and visibility witness $\mathcal{V}$, noted $\mathcal{D}_\tau \ \mathsf{sat}[\mathcal{V}, \mathcal{T}]\ \mathcal{F}_\tau$, if the witness abstract execution of every concrete execution satisfies the specification:

$$\forall C \in [\![\mathcal{D}_\tau]\!] \cap \mathcal{T} : \mathsf{abs}(C, \mathcal{V}) \models \mathcal{F}_\tau.$$

[9] More precisely, we consider only prefix-closed specifications that restrict delivery events or timestamps only.

All example implementations from Section 2.2.3 satisfy their specifications w.r.t. network specifications and visibility witnesses of their respective categories, listed next. We are not considering formal verification of correctness in this thesis, addressed by other work [34].

2.4.3 Implementation Categories

We now define and discuss a number of network specification and visibility witness choices that together define implementation categories.

To specify them, we define the **delivery order** $\mathsf{del}(C)$ of a concrete execution C as follows:

$$e \xrightarrow{\mathsf{del}(C)} f \iff e \xrightarrow{C.\mathsf{eo}} f \wedge C.\mathsf{action}(e) = \mathsf{send} \wedge C.\mathsf{action}(f) = \mathsf{deliver} \wedge C.\mathsf{msg}(e) = C.\mathsf{msg}(f).$$

2.4.3.1 Network Specifications

Network specification $\mathcal{T}$ defines what message delivery patterns are allowed. We reproduce a number of popular network specifications [34], discuss them here, and formally define them in Appendix A. They correspond to well-known definitions from distributed system models [36].

If an implementation tolerates lost, duplicated, and reordered delivery, it can operate under **any network specification** $\mathcal{T}^{\mathsf{any}}$, such that $[\![\mathcal{D}_\tau]\!] \cap \mathcal{T}^{\mathsf{any}} = [\![\mathcal{D}_\tau]\!]$. State-based implementations, such as Algorithm 2.3, do tolerate any network specification.

The **at-most-once delivery** specification $\mathcal{T}^{\leq 1}$ requires that a message is not delivered twice to the same replica, and never delivered to the sender's replica, where it is already known. **At-least-once delivery** $\mathcal{T}^{\geq 1}$ requires that every sent message must be eventually delivered at every remote replica. The conjunction of the two is **reliable delivery** $\mathcal{T}^{1}$, or **exactly-once delivery**. For example, the op-based counter from Algorithm 2.2 requires reliable delivery.

Some op-based implementations require additional ordering guarantees for message delivery. An important condition is **causal delivery** $\mathcal{T}^{\mathsf{c}}$, where delivery of a message requires that all messages known to the sender were delivered at the receiver [36]. In Appendix A.2, we give the example of an optimized op-based multi-value register that requires reliable causal delivery.

Stronger delivery assumptions can make the object implementations simpler and more space-efficient. However, implementing the underlying network delivery with stronger semantics can be more complex and costly [36].

The behavior of some arbitration-based objects relies on the properties of timestamps. **Causal timestamp** specification $\mathcal{T}^{\mathsf{ct}}$ requires that timestamp provided by a replica are greater than any timestamp it has previously observed. This is the case, for instance, with Lamport clocks.

Note that correctness w.r.t. Definition 2.8 does not in itself ensure the liveness of convergence, i.e., eventual visibility (Equation 2.4). This may require additional liveness restrictions, such as that every update is eventually followed by send event, i.e., **eventual flush** specification $\mathcal{T}^{\mathsf{f}}$. Moreover, even implementations that tolerate any network specification may require additional *fairness* of network channels to ensure convergence, so that not all messages are lost [36].

Our system model does not treat replica failures explicitly. Transient and permanent replica failures are indistinguishable from lost messages. We discuss the consequences where relevant.

2.4.3.2 Visibility Witnesses

A visibility witness $\mathcal{V}$ specifies the visibility of events as affected by message delivery.

Implementations with a **transitively delivering** witness, e.g., state-based implementations, transmit all updates known to the sender replica to the receiver replica. Formally:

$$\mathcal{V}^{\mathsf{state}}(C) = (\mathsf{ro}(C) \cup \mathsf{del}(C))^{+}|_{\mathsf{do}}.$$

Conversely, implementations with a **selectively delivering** witness [34], e.g., op-based implementations, transmit only the latest updates of the sender:

$$\begin{aligned}\mathcal{V}^{\mathsf{op}}(C) = \mathsf{ro}(C) \cup \{(e,f) \mid e,f, \in C.E|_{\mathsf{do}} \land \exists e',f' : e \xrightarrow{\mathsf{ro}(C)} e' \xrightarrow{\mathsf{del}(C)} f' \xrightarrow{\mathsf{ro}(C)} f \\ \land \neg\exists e'' : e \xrightarrow{\mathsf{ro}(C)} e'' \xrightarrow{\mathsf{ro}(C)} e' \land C.\mathsf{action}(e'') = \mathsf{send}\}\end{aligned}$$

Other possible witnesses include: protocols with intermediate approaches (i.e., neither op-based nor state-based) [7], protocols that require additional communication round-trips to transmit updates [5], or protocols with topology-restrictions [35].

An implementation with a witnesses that transmits updates earlier and transitively may speed up convergence, especially during failures. However, it may also saturate the network with larger messages.

Visibility witness and network specifications can together enforce desirable updates ordering guarantees at an object level, such as *causal consistency* [4, 34]. We make use of it in Part III.

2.4.3.3 Main Categories

We conclude with a comparison of the two main implementation categories, op-based and state-based, presented in Table 2.1. Although it is not an exhaustive list, other categories tend to be some variation or combination of the two. We will discuss some of them in Chapter 5.

The main characteristics of an implementation category are its network specification and visibility witness. Op-based implementations impose strong requirements on the network layer specification; details depend on type and implementation, but in the common case, they rely on reliable causal delivery. In contrast, state-based objects have almost no network requirements. An op-based implementation transmits only the recent local updates, whereas a state-based transmits all updates transitively.

The presence of different implementation categories raises a natural question: which one is the best choice? There is no universal answer. We review this question from different angles, in the rows of Table 2.1.

State and message metadata size is typically low for op-based implementations, and much higher for state-based ones (compare, for instance, Algorithm 2.2 with Algorithm 2.3). Both of

	Op-based	**State-based**
Network specification requirement	strong: up to $\mathcal{T}^{\mathsf{1c}}$ (reliable causal delivery)	weak: $\mathcal{T}^{\mathsf{any}}$ (any, with fairness)
Transfer / visibility of updates	selective: $\mathcal{V}^{\mathsf{op}}$	transitive: $\mathcal{V}^{\mathsf{state}}$
State metadata size	lower	higher
Multi-versioning integration	external / easy (Part III)	internal / more difficult
Cross-object guarantees integration	external / easy (Part III)	internal / more difficult
Bandwidth / buffer optimizations	log reduction [35, 78]	deltas [7, 45]
Topology-specific optimizations	possible [35]	possible [5]
Example implementations	SwiftCloud (Part III), Gemini [65], Walter [95]	Riak DT [1]

Table 2.1: Comparison of op-based and state-based implementation categories.

them permit type-specific metadata optimizations. This motivates our metadata optimality study in Part II for state-based implementations.

Low metadata size of the op-based implementations does not indicate their cost is always lower in absolute terms, since the network layer specification that they require also has a cost. The stronger the network specification, the smaller the op-based object metadata, but the higher is the cost and the complexity of the update delivery protocol implementation. In contrast, the delivery protocol for state-based implementations does not impose any significant cost, and in particular, does not require to store a log of messages to send. On the other hand, the delivery protocol is easily *shared* between op-based objects.

Another angle of comparison is support for cross-object consistency protocols, and their multi-versioning mechanisms [14, 46, 66, 67]. These can be easily integrated *externally* with op-based implementations, using a standard log-based implementation of the database. In contrast, for state-based implementations this is more complex and less modular (it requires custom implementation for every type).

These factors motivate our choice of op-based category for the SwiftCloud system with cross-object consistency and versioning, described in Part III. Conversely, state-based implementations are used by object stores implementing independent objects without cross-object consistency.

Bandwidth and buffer utilization optimizations are possible for variants of both categories. Update delivery protocols for op-based implementations can use semantics of messages (updates) to reduce the log of updates to propagate [35, 59, 78, 82]. Similarly, recent variants of state-based implementations send only fragments of their states [7, 45], as we discuss in Chapter 5. Topology-specific optimizations that rely on type semantics also exist for both categories [5, 35].

Part II

Optimality of Replicated Data Types

Chapter 3

Metadata Space Complexity Problem

Every town has a story. Tombstone has a legend.

Tombstone by George P. Cosmatos

Contents

This chapter concerns the problem of minimizing the metadata cost incurred by RDT implementations. Metadata is information required by implementations to handle concurrent updates and failures. For some implementations the space complexity of metadata is substantial, especially for state-based ones. Designing a space-optimized implementation is nontrivial — an incorrect design may violate correctness under complex concurrency or failure scenarios. This chapter studies this problem.

In Section 3.1, we define a metric for metadata overhead, which supports an asymptotic worst-case analysis. In Section 3.2, we review some existing state-based implementations, and study the opportunities for improvements. In particular, we look at a few difficult instances of the problem for variants of set, register and counter types. For many of them existing implementations incur an overhead in the order of the number updates, or in (the square of) the number the replicas in the system. We present the design of two implementations that improve over the existing ones, and minimize the metadata overhead down to the number of replicas. We also report on unsuccessful optimization attempts.

3.1 Problem Statement

Object states in replicated data types include not only the client-observable content, but also *metadata* needed for conflict resolution and for masking network failure. Similarly, messages include metadata. The space taken by this metadata is a major factor of storage and bandwidth cost, or even of feasibility. As illustrated earlier by both counter (Algorithm 2.3) and by register (Algorithm 2.5) implementations, metadata is particularly large for state-based implementations. Such implementations are used, for example, by object stores without cross-object consistency guarantees, such as Riak 2.0 [3]. This motivates our study of *state metadata complexity for state-based implementations* in this part of the thesis. We do not directly consider message size; the size of a state sets a bound on the size of a message sending that state.[1]

To measure and compare space requirements of different implementations of a data type, we need a common metric. To this end, we consider how data are represented. An **encoding** of a set S is an injective function $\mathsf{enc} : S \to \Lambda^+$ (i.e., encoding is decodable), where Λ is some suitably chosen finite set of characters that is fixed and common for all data representation; for example, binary encoding ($\Lambda = \{0,1\}$) or byte encoding ($\Lambda = \{0,1,\dots,255\}$). Sometimes, we clarify the domain being encoded using a subscript: e.g., $\mathsf{enc}_{\mathbb{N}}(1)$ indicates the encoding of value 1 from the domain of integers. For $s \in S$, we define $\mathsf{len}_S(s)$ as the length of $\mathsf{enc}_S(s)$, i.e., the number of characters. The length can vary within the encoded domain: e.g., the length of encoding of an integer k is proportional to its value, i.e., $\mathsf{len}_{\mathbb{N}_0}(k) \in \Theta(\lg k)$. We use standard encodings schemes for primitive values (such as integers, sets, tuples, etc.) used in object return values $\mathsf{enc}_{\mathsf{Val}_\tau}$ of

[1]We discuss some orthogonal work on message size optimizations in Chapter 5.

each data type τ we consider; the standard encodings are explicitly specified in Appendix B.1. For sake of generality, we allow *arbitrary* but fixed encoding of implementation states, $\mathsf{enc}_{\mathcal{D}_\tau.\Sigma}$.

The state of an object contains both client-observable value and metadata, and both can *vary* within and across different executions. As the size of the value is a lower bound on the size of the state, our analysis focuses on metadata, or how much overhead does the metadata causes. We call metadata the part of the state that is not returned in a `read` query. Formally, for a concrete execution $C \in [\![\mathcal{D}_{\mathcal{D}_\tau}]\!]$ and a read event $e \in C.E|_{\texttt{read}}$, we define $\mathsf{state}(e)$ to be the state of the object accessed at e: $\mathsf{state}(e) = R(C.\mathsf{replica}(e))$ for $(R, \_) = C.\mathsf{pre}(e)$. The **metadata overhead** is the ratio of the size of the object state over the size of the read value.

In this work, we focus on the *worst-case analysis*, which is relevant, for example, under high concurrency or during network partitions. It does not necessarily represent average or long-term behavior. We observe that for all the data types that we analyze the overhead is related to the number of update operations and/or the number of replicas involved in an execution. We quantify the worst-case overhead by taking the maximum of the overhead ratio over all read operations in all executions with a given number of replicas n and update operations m.

Definition 3.1 (Maximum and Worst-Case Metadata Overhead)**.** The **maximum metadata overhead** of an execution $C \in [\![\mathcal{D}_{\mathcal{D}_\tau}]\!]$ of an implementation $\mathcal{D}_\tau$, noted $\mathsf{mmo}(\mathcal{D}_\tau, C)$, is:

$$\mathsf{mmo}(\mathcal{D}_\tau, C) = \max\left\{ \frac{\mathsf{len}_{\mathcal{D}_\tau.\Sigma}(\mathsf{state}(e))}{\mathsf{len}_{\mathsf{Val}_\tau}(C.\mathsf{rval}(e))} \mid e \in C.E|_{\texttt{read}} \right\}.$$

The **worst-case metadata overhead** of an implementation $\mathcal{D}_\tau$ over all executions with n replicas and m updates ($2 \le n \le m$), noted $\mathsf{wcmo}(\mathcal{D}_\tau, n, m)$, is:

$$\begin{aligned}\mathsf{wcmo}(\mathcal{D}_\tau, n, m) = \max\{\mathsf{mmo}(\mathcal{D}_\tau, C) \mid C \in [\![\mathcal{D}_\tau]\!] \wedge n = |\{C.\mathsf{replica}(e) \mid e \in C.E\}| \\ \wedge\, m = |\{e \in C.E \mid C.\mathsf{op}(e) \in \mathsf{Op}_\tau \setminus \{\texttt{read}\}\}|\}.\end{aligned}$$

The definition of mmo takes into account only states prior to `read` queries, which may appear limited. However, wcmo definition quantifies over all executions, which includes any number of `read` queries. Similarly, we consider only executions with $m \ge n$, since we are interested in the asymptotic overhead of executions where all replicas can be mutated (i.e., perform at least one update operation); we ignore read-only replicas for brevity.

The definition of the worst-case metadata overhead allows us to express the *asymptotic* complexity of implementations, a classical tool in worst-case analysis.

Definition 3.2 (Asymptotic Metadata Overhead)**.** Consider an implementation $\mathcal{D}_\tau$ and a positive function $f(n,m)$.

- f is an **asymptotic upper bound**, noted $\mathcal{D}_\tau \in \widehat{O}(f(n,m))$, if $\sup\limits_{n,m\to\infty} \dfrac{\mathsf{wcmo}(\mathcal{D}_\tau, n, m)}{f(n,m)} < \infty$,

$$\text{i.e., } \exists K > 0 : \forall m \ge n \ge 2 : \mathsf{wcmo}(\mathcal{D}_\tau, n, m) < K f(n,m).$$

- f is an **asymptotic lower bound**, noted $\mathcal{D}_\tau \in \widehat{\Omega}(f(n,m))$, if $\lim_{n,m\to\infty} \frac{\mathsf{wcmo}(\mathcal{D}_\tau,n,m)}{f(n,m)} \neq 0$,

$$\text{i.e., } \exists K > 0 : \forall m_0 \geq n_0 \geq 2 : \exists n \geq n_0, m \geq n_0 : \mathsf{wcmo}(\mathcal{D}_\tau,n,m) > Kf(n,m).$$

- f is an **asymptotically tight bound**, noted $\mathcal{D}_\tau \in \widehat{\Theta}(f(n,m))$, if it is both an upper and a lower asymptotic bound.

Note that we use the $\widehat{\text{hat}}$ symbol to distinguish metadata overhead bounds from function bounds.

An asymptotic bound characterizes metadata overhead of an implementation concisely, and serves to derive a notion of asymptotic implementation *optimality*.

Definition 3.3 (Optimality). Implementation $\mathcal{D}_\tau$ is an **asymptotically optimal state-based implementation** of type τ with an upper bound complexity $\mathcal{D}_\tau \in \widehat{O}(f(n,m))$ if it is correct, i.e., $\mathcal{D}_\tau\ \mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]\, \mathcal{F}_\tau$, and its upper bound is a lower bound for all correct implementations:

$$\forall \mathcal{D}'_\tau : \mathcal{D}'_\tau\ \mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]\, \mathcal{F}_\tau \implies \mathcal{D}'_\tau \in \widehat{\Omega}(f(n,m))$$

The goal of this part of the thesis is the design and/or identification of asymptotically optimal state-based implementations.

3.2 Optimizing Implementations

In this section, we analyze the metadata overhead of some existing data type implementations, and explore opportunities for improved, optimized designs.

3.2.1 Successful Optimizations

We begin with two successful nontrivial metadata optimizations using a variant of version vectors. They reduce the overhead to the order of the number of replicas in the system.

3.2.1.1 Add-Wins Set

The set is one of the most fundamental data types. It can be used directly and as a basis for types such as maps or graphs [27, 52, 93]. Among the alternatives discussed in Section 2.3.2.3, the add-wins set semantics $\mathcal{F}_{\texttt{AWSet}}$ (Equation 2.10) is a popular one [3, 45, 72].

Existing implementation. Algorithm 3.1 presents the basic state-based implementation of add-wins set by Shapiro et al. [92], translated into our notation.

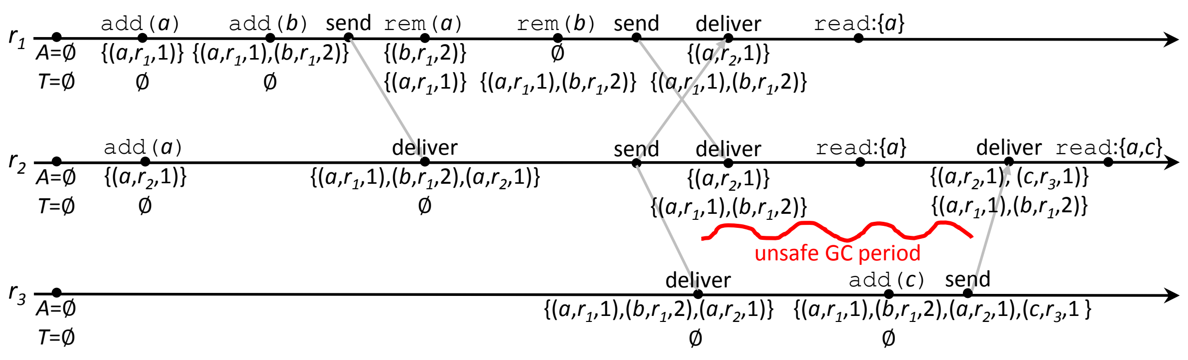
To implement add-wins, the idea is to distinguish different invocations of $\mathsf{add}(a)$ by attaching, to each added element, a hidden unique token, or timestamp, t. We use a custom timestamp $t = (r,k)$, where r is the replica ID and k is a natural number assigned by that replica. The concrete implementation stores a set A of active **element instance** triples (a,r,k), one for each

Algorithm 3.1 Naive state-based implementation of add-wins set (AWSet).

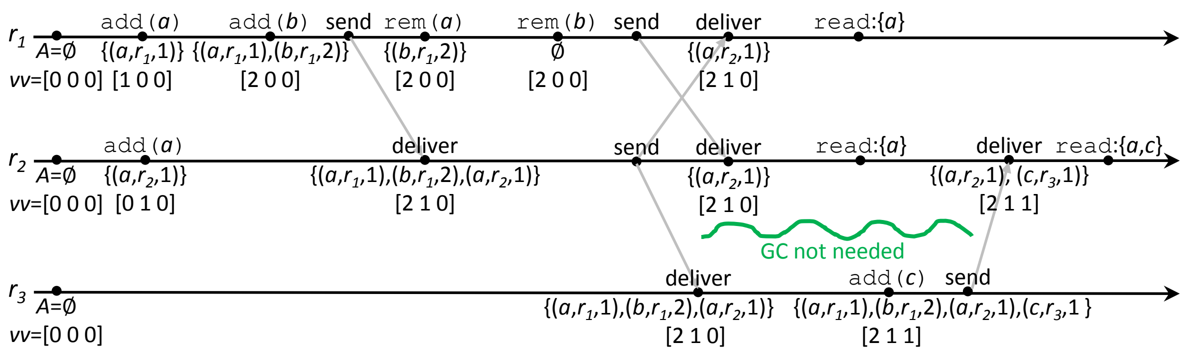
```
1: $\Sigma = \mathsf{ReplicaID} \times \mathcal{P}(\mathbb{Z} \times \mathsf{ReplicaID} \times \mathbb{N}_0) \times \mathcal{P}(\mathsf{ReplicaID} \times \mathbb{N}_0)$
2: $M = \mathcal{P}(\mathbb{Z} \times \mathsf{ReplicaID} \times \mathbb{N}_0) \times \mathcal{P}(\mathsf{ReplicaID} \times \mathbb{N}_0)$
3: initialize($r_i$) : $(r, A, T)$
4:     let $r = r_i$                                   ▷ replica ID
5:     let $A = \emptyset$        ▷ active element instances: triples (element $a$, replica $r'$, counter $k$)
6:     let $T = \emptyset$   ▷ tombstones: timestamps of removed instances: pairs (replica $r'$, counter $k$)
7: do(read, $t_o$) : $V$
8:     let $V = \{a \mid \exists r', k : (a, r', k) \in A\}$          ▷ return values of active instances
9: do(add($a$), $t_o$)
10:    let $k = \max\{k' \mid (\_, r, k') \in A \vee (r, k') \in T \vee k' = 0\} + 1$   ▷ next timestamp for new instance
11:    $A \leftarrow A \cup \{(a, r, k)\}$                ▷ create and add a new instance of $a$
12: do(rem($a$), $t_o$)
13:    let $D = \{(a, r', k) \mid \exists r', k : (a, r', k) \in A\}$      ▷ identify all instances of element $a$
14:    $A \leftarrow A \setminus D$                        ▷ remove them from active instances
15:    $T \leftarrow T \cup \{(r', k) \mid (a, r', k) \in D\}$        ▷ add their timestamps to tombstones
16: send() : $(A_m, T_m)$
17:    let $A_m = A$
18:    let $T_m = T$
19: deliver($(A_m, T_m)$)
20:    $A \leftarrow (A \cup A_m) \setminus \{(a, r', k) \mid (r', k) \in T \cup T_m\}$   ▷ merge all instances, except tombstones
21:    $T \leftarrow T \cup T_m$                            ▷ merge sets of tombstones
```

add(a), rather than just elements. An element a is removed by removing every instance (a, r, k) of that element from the active set and recording its timestamp (r, k) in a **tombstone set** T. Timestamps in A and tombstones in T are therefore maintained as disjoint sets. An element is in the set, i.e., is included in return value of read, if there is an instance of it in the active set; tombstones are not visible to read, and they are used only during replication as a trace of removal. A tombstoned element, can be always added again by creating a new instance (a, r', k') with a fresh timestamp (r', k'), different from the old one, $(r', k') \neq (r, k)$. If the same element a is both added and removed concurrently, the remove concerns only observed instances and not the concurrently-added unique instance. Therefore the add wins, by adding a new instance. The replication protocol transmits information about active and tombstone instances. The receiver removes active instances that are present in the received tombstones set. Burckhardt et al. [34] prove that this implementation indeed satisfies the specification $\mathcal{F}_{\mathtt{AWSet}}$.

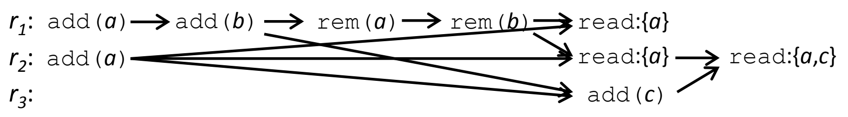
Figure 3.1A illustrates an example execution. Initially the state is empty. Replica r_1 adds some integer elements a and b, and later removes them by placing their timestamps in the tombstone set. Concurrent add of element a at replica r_2 generates a new instance. Therefore, when replicas r_1 and r_2 synchronize, the instance of a from replica r_2 "wins" over the remove and makes a visible. Replica r_3 delivers a message from r_2 including all add updates, but no rem updates, before it becomes disconnected (as indicated by a squiggly line). During disconnection, it


r1
add(a) add(b) send rem(a) rem(b) send deliver read:{a}
A=∅ {(a,r1,1)} {(a,r1,1),(b,r1,2)} {(b,r1,2)} ∅ {(a,r2,1)}
T=∅ ∅ ∅ {(a,r1,1)} {(a,r1,1),(b,r1,2)} {(a,r1,1),(b,r1,2)}
r2
add(a) deliver send deliver read:{a} deliver read:{a,c}
A=∅ {(a,r2,1)} {(a,r1,1),(b,r1,2),(a,r2,1)} {(a,r2,1)} {(a,r2,1), (c,r3,1)}
T=∅ ∅ ∅ {(a,r1,1),(b,r1,2)} {(a,r1,1),(b,r1,2)}
unsafe GC period
r3
deliver add(c) send
A=∅ {(a,r1,1),(b,r1,2),(a,r2,1)} {(a,r1,1),(b,r1,2),(a,r2,1),(c,r3,1 }
T=∅ ∅ ∅

(A) Execution of the naive implementation (Algorithm 3.1).


r1
add(a) add(b) send rem(a) rem(b) send deliver read:{a}
A=∅ {(a,r1,1)} {(a,r1,1),(b,r1,2)} {(b,r1,2)} ∅ {(a,r2,1)}
vv=[0 0 0] [1 0 0] [2 0 0] [2 0 0] [2 0 0] [2 1 0]
r2
add(a) deliver send deliver read:{a} deliver read:{a,c}
A=∅ {(a,r2,1)} {(a,r1,1),(b,r1,2),(a,r2,1)} {(a,r2,1)} {(a,r2,1), (c,r3,1)}
vv=[0 0 0] [0 1 0] [2 1 0] [2 1 0] [2 1 1]
GC not needed
r3
deliver add(c) send
A=∅ {(a,r1,1),(b,r1,2),(a,r2,1)} {(a,r1,1),(b,r1,2),(a,r2,1),(c,r3,1 }
vv=[0 0 0] [2 1 0] [2 1 1]

(B) Execution of the optimized implementation (Algorithm 3.2).


r1: add(a) add(b) rem(a) rem(b) read:{a}
r2: add(a) read:{a} read:{a,c}
r3: add(c)

(C) Witness abstract execution for the two executions. Arrows show visibility, arbitration is irrelevant.

Figure 3.1: Concrete executions of two implementations of add-wins set (`AWSet`), and their witness abstract execution. Replica IDs in an object state, and message content for send (the copy of state) are ommited. The period of interrupted connectivity is indicated with a squiggly line.

performs an add(c). After r_3 reconnects to r_2, replicas are still able to determine which elements present in replica r_3 were removed, and which are new, thanks to unique instances, and the tombstone set.

This approach is correct, but incurs high metadata overhead, in the order of number of updates m in an execution.

Theorem 3.1. *Let $\mathcal{D}_{\texttt{AWSet}}$ be the naive add-wins set implementation defined in Algorithm 3.1, such that $\mathcal{D}_{\texttt{AWSet}}$ sat[$\mathcal{V}^{\text{state}}, \mathcal{T}^{\text{any}}$] $\mathcal{F}_{\texttt{AWSet}}$. The complexity of $\mathcal{D}_{\texttt{AWSet}}$ is $\widehat{\Theta}(m \lg m)$.*

A formal proof is in Appendix B.2. Intuitively, every add requires a new instance (even sequential or concurrent adds of a same element), which remains as a timestamp in the tombstone set after being removed. The factor $\lg m$ is the cost of storing the identity of each instance.

Algorithm 3.2 Optimized state-based implementation of add-wins set (`AWSet`).

```
$\Sigma = \text{ReplicaID} \times (\text{ReplicaID} \to \mathbb{N}_0) \times \mathcal{P}(\mathbb{Z} \times \text{ReplicaID} \times \mathbb{N}_0)$
$M = (\text{ReplicaID} \to \mathbb{N}_0) \times \mathcal{P}(\mathbb{Z} \times \text{ReplicaID} \times \mathbb{N}_0)$
initialize($r_i$) : $(r, vv, A)$
    let $r = r_i$    ▷ replica ID
    let $vv = \lambda s.0$    ▷ vector summarizing timestamps known to this replica
    let $A = \emptyset$    ▷ active element instances: triples (element $a$, replica $r'$, counter $k'$)
do(read, $t_o$) : $V$
    let $V = \{a \mid \exists r', k : (a, r', k) \in A\}$    ▷ return values of active instances
do(add($a$), $t_o$)
    let $D = \{(a, r, k) \mid \exists k : (a, r, k) \in A\}$    ▷ collect all instances of $a$ created by local replica
    $A \leftarrow A \cup \{(a, r, vv(r) + 1)\} \setminus D$    ▷ add new instance of $a$ and replace local instances if any
    $vv \leftarrow vv[r \mapsto vv(r) + 1]$    ▷ increment own entry in the vector
do(rem($a$), $t_o$)
    let $D = \{(a, r', k) \mid \exists r', k : (a, r', k) \in A\}$    ▷ identify all instances of element $a$
    $A \leftarrow A \setminus D$    ▷ remove them from active instances
send() : $(vv_m, A_m)$
    let $vv_m = vv$
    let $A_m = A$
deliver($(vv_m, A_m)$)
    let $A' = A \setminus \{(a, r', k) \notin A_m \mid k \in \mathbb{N} \wedge k \le vv_m(r')\}$    ▷ keep local instances not rem. remotely
    let $A'' = A_m \setminus \{(a, r', k) \notin A \mid k \in \mathbb{N} \wedge k \le vv(r')\}$    ▷ keep remote instances not removed locally
    $A \leftarrow A' \cup A''$
    $vv \leftarrow vv \sqcup vv_m$    ▷ update vector to entry-wise maximum (cf. Equation 2.2)
```

Discarding instances is not simple. Unfortunately, getting rid of instances is not easy in a system with failures. They protect from duplicated and reordered messages, and disconnected replicas. A background **garbage collection** (GC) protocol could collect unnecessary instances [51, 102] (we will discuss it in Chapter 5). However, this incurs overhead too, and requires *update stability* for progress, i.e., every replica must acknowledge receiving an update. Replica can be exempted from acknowledgment only if it permanently crashed, but such an exemption requires perfect failure detector [36], thus network or replica failure can prevent GC for extended duration. On the other hand, it is *unsafe* to GC an instance before it is known to be replicated everywhere.

An execution from Figure 3.1A, described earlier, illustrates this problem. While replica r_3 is disconnected (or failed), replicas r_2 and r_3 do not know if r_3 received tombstones for elements a and b. If they were to remove tombstone for a and b, when r_3 connects again to r_2, r_2 would be unable to distinguish a fresh instance of element c and previously removed instances of a and b.

Optimized implementation. Algorithm 3.2 presents a new optimized design of add-wins set implementation. It addresses the metadata complexity problem by summarizing: *(1)* removed and *(2)* re-added element instances. We explain our design by comparison to the naive one. We describe optimization *(1)* first, assuming $D = \emptyset$ in Line 10, which disables optimization *(2)*.

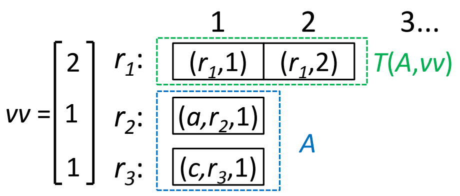


Figure 3.2: Semantics of a version vector in the optimized add-wins set. The illustration refers to the last state of replica r_2, (r_2, vv, A) in Figure 3.2. Horizontally-aligned boxes illustrate the instances (timestamps) represented by vector entry for that replica, whereas dashed rectangles represent the sets A and $T(vv, A)$.

Optimization 1: Efficient summary of removed elements without tombstones. Our solution is similar to the metadata of a replicated file system of Bjørner [24] (see Chapter 5). It is based on the observation that an implementation does not need to store an explicit set of tombstones to represent information about removals. If each replica assigns timestamps contiguously, i.e., $(r_i, 1), (r_i, 2), \ldots$, and the protocol ensures their in-order delivery (as in the case of state-based transitive delivery), it suffices to recall the number of adds from each replica. Thus, a version vector can *compactly summarize knowledge about the timestamps that the replica has observed*. Formally, let function $\mathcal{TS}(vv)$ define the timestamps represented by a vector vv:

$$\mathcal{TS}(vv) = \{(r,k) \mid r \in \mathsf{dom}(vv) \land k \in \mathbb{N} \land k \le vv(r)\}. \tag{3.1}$$

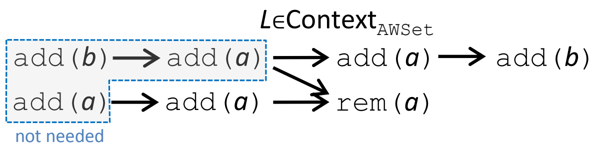
The concrete state in the optimized implementation is a triple (r, vv, A), where r is the replica ID, vv is a vector, and A is a set of active instances. There is no tombstone set. The timestamps represented by vv include both active instances A and timestamps of the missing tombstones. Although the tombstones set is not part of state, it is possible to *logically* reconstruct it from a concrete state (r, vv, A) of the optimized implementation. We can write: $\mathcal{TS}(vv) = \{(r,k) \mid \exists a : (a,r,k) \in A\} \uplus T(vv, A)$, where $T(vv, A)$ is the logical tombstone set. Therefore:

$$T(vv, A) = \mathcal{TS}(vv) \setminus \{(r,k) \mid \exists a : (a,r,k) \in A\}. \tag{3.2}$$

Figure 3.2 illustrates this relation between version vector vv, set A, and the computed tombstone set $T(vv, A)$. It is based on the example state of replica r_2 from Figure 3.1B.

The optimized algorithm needs to use this new data structure. The add method now associates an instance to the replica's own entry in the vector, which is incremented. The `rem` method is unchanged compared to the naive implementation, except for the absence of the tombstone set. The most complex optimization concerns the deliver replication logic. Version vector is used to check for timestamp inclusion.[2] deliver computes the new set of active instances as a union of two sets: local instances that were not removed remotely, and remote instances that were not removed locally. In Line 20, the algorithm keeps a local instance unless the remote replica observed it

[2]Note that this is different from how the multi-value register (Section 2.2.3) uses vectors, comparing vectors assigned to events to check if events are concurrent.


$L \in \text{Context}_{\texttt{AWSet}}$
add (b) → add (a) → add (a) → add (b)
add (a) → add (a) → rem (a)
not needed

$$\mathcal{F}_{\texttt{AWSet}}(L, \texttt{read}) = \{a, b\}$$

Figure 3.3: Operation context of an add-wins set illustrating opportunity for coalescing adds. The set of operations that are irrelevant for semantics of `read` is indicated with a blue dashed shape.

previously (according to vv_m) and does not include it as an active instance in A_m anymore, which indicates a remote removal. Similarly, in Line 21, the algorithm integrates a remote instance unless the local replica observed it previously and removed it. The new version vector represents the combined knowledge, i.e., the least upper bound (Equation 2.2) of local and remote vectors.

One way to reason about correctness of the optimization is to relate it with the naive implementation. The deliver algorithm is equivalent to the naive implementation. Consider any state (r, A, T) of the naive implementation (resp., an equivalent optimized state (r, vv, A)) and a message (A_m, T_m) (resp., an equivalent optimized message (vv_m, A_m)). The delivery of the message in Line 20 of the naive implementation is equivalent to Line 22 of the optimized one:

$$\begin{aligned}(3.3)\qquad A &\overset{(i)}{\leftarrow} (A \cup A_m) \setminus \{(a, r', k) \mid \exists (r', k) \in T \cup T_m\}\\ &= (A \setminus \{(a, r', k) \mid \exists (r', k) \in T \cup T_m\}) \cup (A_m \setminus \{(a, r', k) \mid \exists (r', k) \in T \cup T_m\})\\ &\overset{(ii)}{=} (A \setminus \{(a, r', k) \mid \exists (r', k) \in T_m\}) \cup (A_m \setminus \{(a, r', k) \mid \exists (r', k) \in T\})\\ &\overset{(iii)}{=} (A \setminus \{(a, r', k) \notin A_m \mid (r', k) \in \mathcal{TS}(vv_m)\}) \cup (A_m \setminus \{(a, r', k) \notin A \mid (r', k) \in \mathcal{TS}(vv)\})\\ &\overset{(iv)}{=} (A \setminus \{(a, r', k) \notin A_m \mid k \in \mathbb{N} \wedge k \leq vv_m(r')\}) \cup (A_m \setminus \{(a, r', k) \notin A \mid k \in \mathbb{N} \wedge k \leq vv(r')\}),\end{aligned}$$

where (i) is the assignment from Line 20 of the naive implementation; step (ii) is due to disjointness of A and T (also, A_m and T_m), (iii) is an application of Equation 3.2, and (iv) comes from application of Equation 3.1. A similar argument holds for vectors.

Optimization 2: Coalescing adds. Another optimization *coalesces* multiple active instances of a same element. We observe that, among multiple add operations on the same element, only the *latest ones are relevant* in the add-wins semantics. Consider for example an operation context from Figure 3.3. The latest add(a) at replica r_1 and r_2, and the latest add(b) at r_1 are sufficient to determine the correct return value of `read`. Old adds become irrelevant when new adds arrive. Therefore, we can “coalesce” them: rather than storing all instances of add(a), it suffices to record a vector of the latest timestamps for element a generated by each replica, or equivalently, a set of the latest timestamps for a from each replica.

The optimized add(a) method does exactly that in Line 10 of Algorithm 3.2. When a new instance of element a is generated, the set D of prior instances of a generated by this replica is discarded; D is an empty set or a singleton. This guarantees that in any state A the number of different instances of a same element a is at most in the number of replicas n.

Figure 3.1B illustrates an execution of the optimized implementation, with the same client and network behavior as Figure 3.1A. Note that replicas r_1 and r_2 are now unaffected by unavailability of replica r_3, and discard removed element instances immediately. When r_3 communicates back, the version vector of r_2 prevents removed instances of a and b from reappearing.

Improvement. The two optimizations significantly reduce metadata overhead, down to the order of the number of replicas n.

Theorem 3.2. *Let $\mathcal{D}^*_{\texttt{AWSet}}$ be the optimized add-wins set implementation defined in Algorithm 3.2, such that $\mathcal{D}^*_{\texttt{AWSet}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\texttt{AWSet}}$. The complexity of $\mathcal{D}^*_{\texttt{AWSet}}$ is $\widehat{\Theta}(n \lg m)$.*

A proof is in Appendix B.2. Intuitively, the vector vv has n entries (or less, with some additional minor optimizations that we omit), and every element a has at most n different instances in the set A. The $\lg m$ factor is the size of a vector entry. In Chapter 4 we will show that this implementation is *optimal*, i.e., further state reduction for the worst-case is impossible without affecting fault-tolerance or semantics.

Burckhardt et al. [34] formally prove the optimization correct w.r.t. $\mathcal{F}_{\texttt{AWSet}}$ specification. Mukund et al. [72] offers an alternative proof via bisimulation between the naive and the optimized implementation, along the lines of Equation 3.3 that shows part of the obligations.

Applicability. The optimization for add-wins set is relatively general and could be applied to other similar types, for instance, an add-wins map object [3]. The optimized set can be also viewed as an optimized implementation of a collection of value-wins registers $\mathcal{F}_{\texttt{VWReg}}$ (Equation 2.8) restricted to boolean values. Every register corresponds to a unique element; if an element is present in the set, then the corresponding register has the winning value.

3.2.1.2 Multi-Value Register

The multi-value register is a basic primitive for conflict detection and application-level conflict resolution. Its semantics $\mathcal{F}_{\texttt{MVReg}}$ are specified in Equation 2.7.

Existing implementation. We first analyze the standard implementation from Algorithm 2.5 (Section 2.2.3), proposed in similar forms by different authors [44, 59, 93].

Theorem 3.3. *Let $\mathcal{D}_{\texttt{MVReg}}$ be the multi-value register implementation defined in Algorithm 2.5, such that $\mathcal{D}_{\texttt{MVReg}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\texttt{MVReg}}$. The complexity of $\mathcal{D}_{\texttt{MVReg}}$ is $\widehat{\Theta}(n^2 \lg m)$.*

A proof is in Appendix B.2. The overhead may appear surprisingly high, since each value stored by the implementation carries a single version vector, which could suggest an overhead of $n \lg m$. However, in the most costly case, the same value could be written concurrently by up

Algorithm 3.3 Incorrect state-based implementation of multi-value integer register (`MVReg`).

```
1: $\Sigma = \mathsf{ReplicaID} \times \mathcal{P}(\mathbb{Z} \times (\mathsf{ReplicaID} \hookrightarrow \mathbb{N}_0))$   $M = \mathcal{P}(\mathbb{Z} \times (\mathsf{ReplicaID} \hookrightarrow \mathbb{N}_0))$
2: ...                                            ▷ all other methods are identical to Algorithm 2.5
3: $\mathsf{deliver}(A_m)$
4:     let $A' = \{(a, \bigsqcup\{vv \mid (a, vv') \in A \cup A_M\}) \mid (a, \_) \in A \cup A_M\}$   ▷ compute max. vector per element
5:     $A \leftarrow \{(a, vv) \in A' \mid \forall (a', vv') \in A' : vv \not\sqsubset vv'\}$   ▷ keep elem. not dominated by any other elem.
```

to n replicas. In this case, the actual observable value is a singleton, whereas the state stores n version vectors.[3]

We illustrate this problem in Figure 3.4A. Three replicas concurrently write value 0, assigning it three different version vectors. A fourth replica r_4 delivers their writes, and needs to store all three vectors. This causes $n^2 \lg m$ overhead, as demonstrated by a read operation on replica r_4. The overhead reduces only with subsequent writes.

This overhead may be problematic in practice, as we expect states made of concurrent writes of the same value, which do not require conflict resolution, to last longer than transient states with unresolved conflicting writes. Moreover, this happens in the desirable case of concurrent convergent conflict resolution.

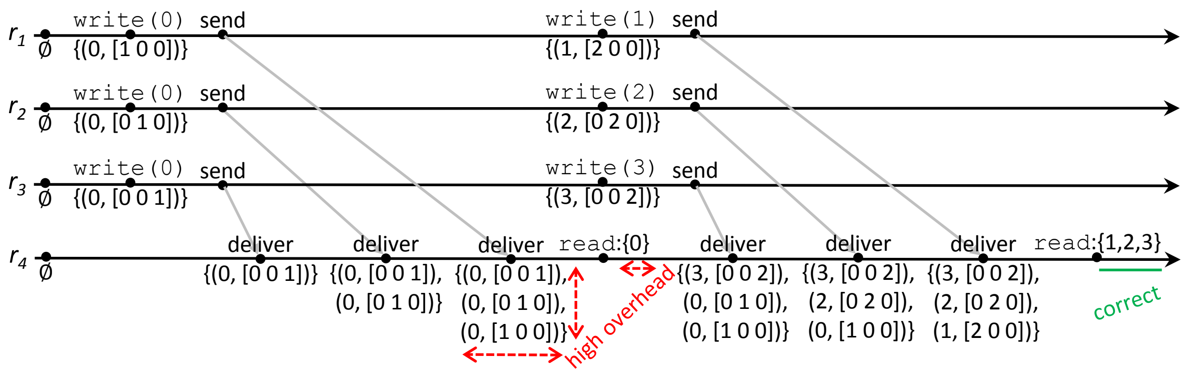
Incorrect optimization. In an optimization attempt, we explored the idea of *merging the vectors of identical elements* in order to store at most one vector per unique value. Algorithm 3.3 applies this idea in the deliver function.

It merges two states by first merging entries for identical elements. The vector of a merged entry is the least upper bound of all vectors for the element (Equation 2.2). This step eliminates multiple entries per element, and brings the overhead down to $\widehat{\Theta}(n \lg m)$. In the second step of deliver, the implementation attempts to eliminate dominated entries, in the same way as the original implementation, by removing an entry if it is dominated by another one. As it turns out, these optimizations are *incorrect*. This demonstrates the difficulty of correct optimization.

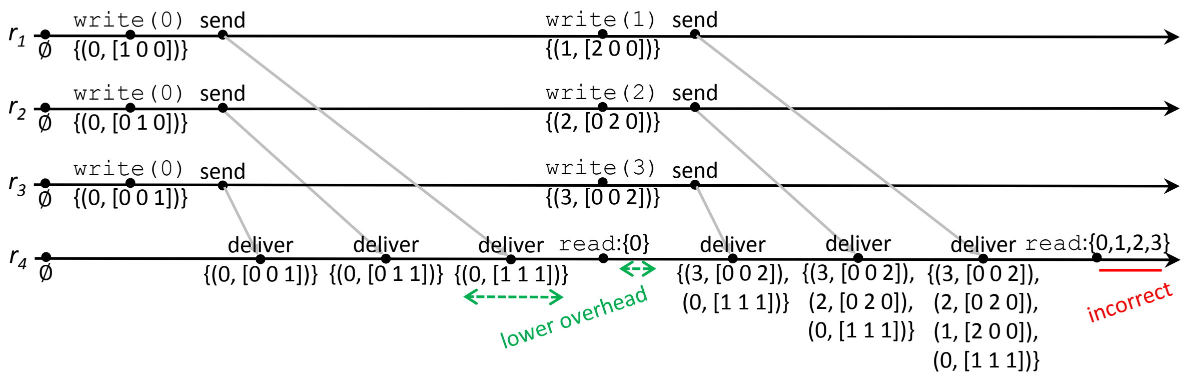
Figure 3.4B illustrates the counterexample. When three concurrent writes of value 0 reach replica r_4, they are merged into a single vector, as explained above. However, each of replicas r_1, r_2, and r_3, subsequently overwrite value 1 with new values, respectively, 1, 2, and 3, being unaware of each other's updates. When these three writes reach r_4, both earlier value 0, and the new values remain in the state, since no vector dominates another. The read operation returns all four values $\{0,1,2,3\}$, which is *incorrect*. As illustrated in the correct abstract execution in Figure 3.4D, all three events of `write`(0) operation were overwritten in visibility by `write`(i) where $i \in \{1,2,3\}$, therefore 0 should not appear in the return value at replica r_4,

Correct optimization. Algorithm 3.4 presents a correct optimized design of multi-value register implementation. This design uses a more complex logic to eliminate merged entries.

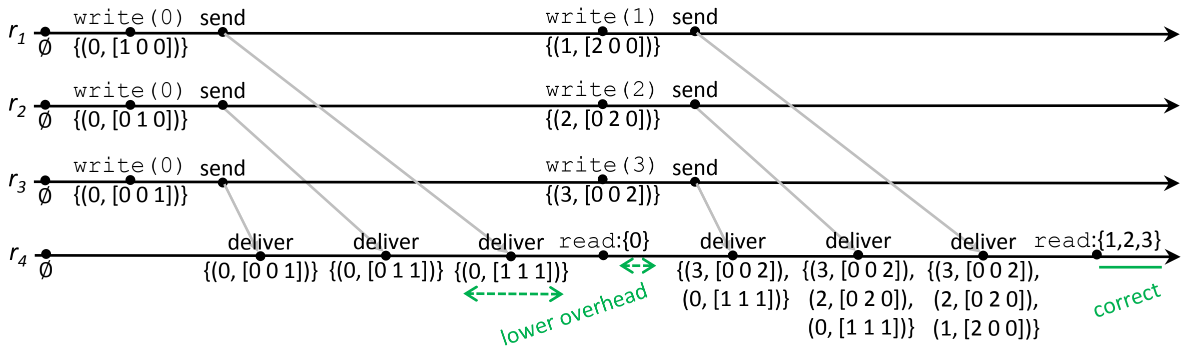
[3]Even for a variant of multi-value register that returns a multi-set in this case [93], the overhead would be n^2.


r1 ∅ write(0) {(0, [1 0 0])} send write(1) {(1, [2 0 0])} send
r2 ∅ write(0) {(0, [0 1 0])} send write(2) {(2, [0 2 0])} send
r3 ∅ write(0) {(0, [0 0 1])} send write(3) {(3, [0 0 2])} send
r4 ∅ deliver {(0, [0 0 1])} deliver {(0, [0 0 1]), (0, [0 1 0])} deliver {(0, [0 0 1]), (0, [0 1 0]), (0, [1 0 0])} read:{0} deliver {(3, [0 0 2]), (0, [0 1 0]), (0, [1 0 0])} deliver {(3, [0 0 2]), (2, [0 2 0]), (0, [1 0 0])} deliver {(3, [0 0 2]), (2, [0 2 0]), (1, [2 0 0])} read:{1,2,3}
high overhead
correct

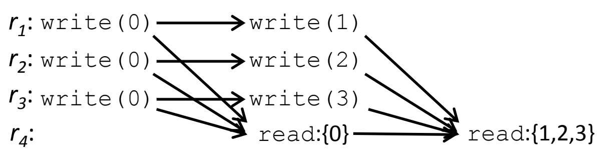
(A) Execution of the basic implementation (Algorithm 2.5).


r1 ∅ write(0) {(0, [1 0 0])} send write(1) {(1, [2 0 0])} send
r2 ∅ write(0) {(0, [0 1 0])} send write(2) {(2, [0 2 0])} send
r3 ∅ write(0) {(0, [0 0 1])} send write(3) {(3, [0 0 2])} send
r4 ∅ deliver {(0, [0 0 1])} deliver {(0, [0 1 1])} deliver {(0, [1 1 1])} read:{0} deliver {(3, [0 0 2]), (0, [1 1 1])} deliver {(3, [0 0 2]), (2, [0 2 0]), (0, [1 1 1])} deliver {(3, [0 0 2]), (2, [0 2 0]), (1, [2 0 0]), (0, [1 1 1])} read:{0,1,2,3}
lower overhead
incorrect

(B) Execution of the incorrectly optimized implementation (Algorithm 3.3).


r1 ∅ write(0) {(0, [1 0 0])} send write(1) {(1, [2 0 0])} send
r2 ∅ write(0) {(0, [0 1 0])} send write(2) {(2, [0 2 0])} send
r3 ∅ write(0) {(0, [0 0 1])} send write(3) {(3, [0 0 2])} send
r4 ∅ deliver {(0, [0 0 1])} deliver {(0, [0 1 1])} deliver {(0, [1 1 1])} read:{0} deliver {(3, [0 0 2]), (0, [1 1 1])} deliver {(3, [0 0 2]), (2, [0 2 0]), (0, [1 1 1])} deliver {(3, [0 0 2]), (2, [0 2 0]), (1, [2 0 0])} read:{1,2,3}
lower overhead
correct

(C) Execution of the correctly optimized implementation (Algorithm 3.4).



(D) Witness abstract execution for correct executions. Arrows show visibility, arbitration is irrelevant.

Figure 3.4: Example execution of three different implementations of multi-value register (MVReg), and a corresponding abstract execution. Note the problematic concurrent writes of the same value. The figure omit the replica ID. The message content for send is always a copy of the state.

Algorithm 3.4 Optimized state-based implementation of multi-value integer register (`MVReg`).

```
1: $\Sigma = \mathsf{ReplicaID} \times \mathcal{P}(\mathbb{Z} \times (\mathsf{ReplicaID} \hookrightarrow \mathbb{N}_0))$   $M = \mathcal{P}(\mathbb{Z} \times (\mathsf{ReplicaID} \hookrightarrow \mathbb{N}_0))$
2: ...                                   ▷ all other methods are identical to Algorithm 2.5
3: $\mathsf{deliver}(A_m)$
4:     let $A' = \{(a, \bigsqcup\{vv \mid (a, vv') \in A \cup A_M\}) \mid (a, \_) \in A \cup A_M\}$   ▷ compute max. vector per element
5:     $A \leftarrow \{(a, vv) \in A' \mid vv \not\sqsubseteq \bigsqcup\{vv' \mid \exists a' \neq a : (a', vv') \in A'\}\}$   ▷ keep elem. not domin. by all others
```

Similarly to the previous implementation, deliver first merges entries with identical elements, using the least upper bound of all vectors for the element as a merged vector. The second phase, of eliminating dominated entries, is different. It removes an entry if the least upper bound of vectors of entries for *all other elements* dominates the vector of that entry, i.e., it expresses the case where an entry was merged, but different writes overwrite the original parts of the merged entry. This rule subsumes the incorrect rule which considered only a direct domination by a single entry. This logic allows to discard merged entries correctly, as proved by Burckhardt et al. [34].

An execution from Figure 3.4C illustrates how the optimized implementation corrects the problematic counterexample. This time, when replica r_4 delivers concurrent `write`(1), `write`(2), and `write`(3) operations that overwrite `write`(0), it discards the entry for value 0 correctly. Indeed, the merged vector it stores for value 0, namely $[r_1 \mapsto 1, r_2 \mapsto 1, r_3 \mapsto 1]$, is dominated by the least upper bound of vectors for values 1, 2, and 3. Therefore, `read` on replica r_4 yields $\{1,2,3\}$ correctly, as in the witness abstract execution (Figure 3.4D).

Theorem 3.4. *Let $\mathcal{D}^*_{\mathtt{MVReg}}$ be the multi-value register implementation defined in Algorithm 3.4, such that $\mathcal{D}^*_{\mathtt{MVReg}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]\, \mathcal{F}_{\mathtt{MVReg}}$. The complexity of $\mathcal{D}^*_{\mathtt{MVReg}}$ is $\widehat{\Theta}(n \lg m)$.*

A proof is in Appendix B.2. This implementation is guaranteed to store one entry per element, yielding the overhead of a single vector. In Chapter 4, we will show that this implementation is asymptotically optimal; in Chapter 5 we will also discuss a recent improvement of Almeida et al. [6], with the same asymptotic complexity, but improved average case.

3.2.2 Prior Implementations and Unsuccessful Optimizations

A number of other important data types exist, for which we did not manage to find a correct optimization. Some of them provide a similar behavior to add-wins set and multi-value register, and could act as their replacement. We report on their complexity here, which motivates some of the questions regarding optimality that we address in Chapter 4.

3.2.2.1 Last-Writer-Wins Register

In the presence of nonnegligible overhead of multi-value register implementation, it is natural to investigate the complexity of arbitration-based last-writer-wins register. LWW register is specified as $\mathcal{F}_{\mathtt{LWWReg}}$ in Equation 2.6. We presented the implementation in Section 2.2.3 (Algorithm 2.4).

Algorithm 3.5 State-based implementation of remove-wins set (RWSet).

```
1: $\Sigma = \mathcal{P}(\mathbb{Z} \times \mathsf{Timestamp}) \times \mathcal{P}(\mathbb{Z} \times \mathsf{Timestamp})$   $M = \mathcal{P}(\mathbb{Z} \times \mathsf{Timestamp}) \times \mathcal{P}(\mathbb{Z} \times \mathsf{Timestamp})$
2: initialize($r_i$) : $(A, T)$
3:     let $T = \emptyset$                    ▷ remove instances: pairs (element $a$, timestamp $t$)
4:     let $A = \emptyset$    ▷ add instances covering remove instances: pairs (element $a$, timestamp $t$)
5: do(read, $t_o$) : $V$
6:     let $V = \{a \mid (\exists t : (a,t) \in A) \wedge (\nexists t' : (a,t') \in T)\}$   ▷ elements with add instance but no rem ins.
7: do(add($a$), $t_o$)
8:     let $D = \{(a,t) \in T\}$                    ▷ identify existing rem instances for element $a$
9:     if $D = \emptyset$ then
10:        $A \leftarrow A \cup \{(a, t_o)\}$ ▷ add a new add instance for $a$ if there is no remove instance to cover
11:    else
12:        $A \leftarrow A \cup D$               ▷ otherwise, turn remove instances into add instances...
13:        $T \leftarrow T \setminus D$                              ▷ ... and discard remove instances
14: do(rem($a$), $t_o$)
15:    $T \leftarrow T \cup \{(a, t_o)\}$                     ▷ add a new remove instance for element $a$
16: send() : $(A_m, T_m)$
17:    let $A_m = A$
18:    let $T_m = T$
19: deliver($(A_m, T_m)$)
20:    $T \leftarrow (T \cup T_m) \setminus (A \cup A_m)$       ▷ union remove instances excluding those covered by adds
21:    $A \leftarrow A \cup A_m$                                    ▷ union sets of add instances
```

Theorem 3.5. *Let $\mathcal{D}^*_{\texttt{LWWReg}}$ be the last-writer-wins register implementation defined in Algorithm 2.4, such that $\mathcal{D}^*_{\texttt{LWWReg}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\texttt{LWWReg}}$. The complexity of $\mathcal{D}^*_{\texttt{LWWReg}}$ is $\widehat{\Theta}(\lg m)$.*

A proof is in Appendix B.2. LWW register incurs a negligible logarithmic overhead (the cost of a timestamp). In Chapter 4, we will show that this marginal cost is unavoidable. Compared to the multi-value register, the overhead is much lower for the LWW register, particularly when the number of replicas is high or there is a replica churn, i.e., replicas disappear and join frequently.

3.2.2.2 Remove-Wins Set

Given that implementations of the add-wins set have noticeable overhead, it is natural to consider other semantics of sets. Algorithm 3.5 is an implementation of remove-wins set semantics $\mathcal{F}_{\texttt{RWSet}}$ (Equation 2.10) by Bieniusa et al. [22].

The implementation is somewhat similar to the naive add-wins set, as it keeps track of every add and rem operations in two sets: a set of add instances A with pairs (a,t), where a is an element and t is a timestamp, and a set of remove instances T with the same type of pairs. The two sets are disjoint. The read method returns an element a if there is an add(a) instance and no rem(a) instance. The rem method simply creates a new remove instance. The add operation turns any remove instances of the element into add instances, or, if there is none, creates a new add instance. The replication protocol is similar to the naive add-wins set.

To our surprise, the overhead of the remove-wins implementation is even *higher* than that of the naive add-wins implementation. Recall that the naive add-wins implementation stored only timestamps of removed elements, not their value. The implementation of remove-wins semantics must store also the value of removed elements, because removes must dominate concurrent adds unaware of their timestamps. Since removed elements are of variable, unbounded size, there is *no upper bound* w.r.t. m or n.[4] Formally, we can give only an underestimated lower bound.

Theorem 3.6. *Let $\mathcal{D}_{\texttt{RWSet}}$ be the remove-wins set implementation defined in Algorithm 3.5, assuming that $\mathcal{D}_{\texttt{RWSet}}\ \mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]\ \mathcal{F}_{\texttt{RWSet}}$. The complexity of $\mathcal{D}_{\texttt{RWSet}}$ is $\widehat{\Omega}(m \lg m)$.*

A proof is in Appendix B.2. Intuitively, the implementation stores at least a timestamp for every remove operation, which incurs the worst-case overhead in the order of the number of operations.

In Chapter 4, we prove that the remove-wins semantics requires to maintain precise information about every `rem` operation, which makes it impossible to efficiently summarize information about removed instances, similar to the add-wins set.[5] A practical implementation can only resort to a background garbage collection protocol [51, 56], which raises other issues.

3.2.2.3 Last-Writer-Wins Set

Another alternative set semantics is the last-writer-wins (LWW) $\mathcal{F}_{\texttt{LWWSet}}$, specified in Equation 2.9. Algorithm 3.6 presents the implementation of LWW set of Bieniusa et al. [22], improved over Shapiro et al. [93]. The implementation maintains the set of latest operations, one for each element updated by either `add` or `rem`. The latest operation is selected according to the timestamp. The type of operation is represented as a presence flag: 1 for `add` and 0 for remove. An element a is included in `read` value if the operation with the latest timestamp has a positive flag value.

Although the implementation is simple, it also incurs an unbounded metadata overhead, as it stores removed elements. Therefore, we claim only an underestimated lower bound.

Theorem 3.7. *Let $\mathcal{D}_{\texttt{LWWSet}}$ be the LWW set implementation defined in Algorithm 3.6, assuming that $\mathcal{D}_{\texttt{LWWSet}}\ \mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]\ \mathcal{F}_{\texttt{LWWSet}}$. The complexity of $\mathcal{D}_{\texttt{LWWSet}}$ is $\widehat{\Omega}(m \lg m)$.*

A proof is in Appendix B.2. Intuitively, the implementation stores timestamps for removed elements, which incurs the overhead in the order of the number of removes.

It is an open problem whether storing timestamps per updated element is necessary for LWW set semantics. In Chapter 3 we answer it only partially. It appears difficult to apply optimizations inspired by the add-wins, since different removed elements may have different timestamps. If this overhead is indeed unavoidable, one could only resort to background garbage collection [56].

[4]If all values stored in a set have a constant size, bound exists; however, metadata overhead metric would become an absolute metadata metric in this case, and the m factor would be bounded by the (constant) cardinality of the domain of values.

[5]Minor optimizations are possible. As they do not change the worst-case behavior, we omit them in Algorithm 3.5.

Algorithm 3.6 State-based implementation of last-writer-wins set (LWWSet).

```
1: $\Sigma = \mathcal{P}(\mathbb{Z} \times \mathsf{Timestamp} \times \{0,1\})$
2: $M = \mathcal{P}(\mathbb{Z} \times \mathsf{Timestamp} \times \{0,1\})$
3: initialize($r_i$) : $S$
4:     let $S = \emptyset$    ▷ set of latest operations: triples (element $a$, timestamp $t$, presence-flag $v$)
5: do(read, $t_o$) : $V$
6:     let $V = \{a \mid \exists t : (a,t,1) \in S\}$ ▷ return values of elements that are present (presence flag = 1)
7: do(add($a$), $t_o$)
8:     if $t_o > t$ then    ▷ sanity check for new timestamp
9:         let $D = \{(a,t,v) \in A\}$    ▷ replace the latest operation on $a$ (if any)...
10:        $A \leftarrow (A \setminus D) \cup \{(a,t_o,1\}$    ▷ ...with a positive presence flag
11: do(rem($a$), $t_o$)
12:     if $t_o > t$ then    ▷ sanity check for new timestamp
13:         let $D = \{(a,t,v) \in A\}$    ▷ replace the latest operation on $a$ (if any)...
14:         $A \leftarrow (A \setminus D) \cup \{(a,t_o,0\}$    ▷ ...with a negative presence flag
15: send() : $S_m$
16:     let $S_m = S$
17: deliver($S_m$)
18:     $S \leftarrow \{(a,t,v) \in S \cup S_m \mid \forall (a,t',v') \in S \cup S_m : t' \neq t \implies t' < t\}$ ▷ keep the latest flag per elem.
```

Note that LWW set can be viewed as a collection of LWW boolean registers of presence flags, one for each element, similarly to the add-wins set considered earlier as a collection of presence-wins registers. Unlike the add-wins set, where some metadata can be shared in a collection, an implementation of the LWW set is no cheaper than implementation of independent registers.

3.2.2.4 Counter

The counter implementation in Algorithm 2.3 (Section 2.2.3) uses vectors to keep track of the number of increments per replica. We can now formally state its complexity.

Theorem 3.8. *Let $\mathcal{D}^*_{\texttt{Ctr}}$ be the counter implementation defined in Algorithm 2.3, such that $\mathcal{D}^*_{\texttt{Ctr}}\ \mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]\, \mathcal{F}_{\texttt{Ctr}}$. The complexity of $\mathcal{D}^*_{\texttt{Ctr}}$ is $\widehat{\Theta}(n)$.*

A proof is in Appendix B.2. Intuitively, the size of a vector is n times bigger than the size of a scalar counter value. In Chapter 4 we will prove that this implementation is optimal.

3.3 Summary

In this chapter, we introduced the metadata space complexity problem and investigated optimization opportunities for six data types. We found that many implementations incur high metadata overhead, linear in the number of updates, or polynomial in the number of replicas. We presented two optimizations, for the add-wins set and the multi-value register. We also found that the complexity of the best known implementations highly depends on the data type semantics.

Chapter 4

Lower Bounds on Complexity and Implementation Optimality

Think it over, think it under.

Winnie-the-Pooh by A. A. Milne

Contents

In the previous chapter, we focused on positive results, i.e., implementations with reduced metadata complexity. The design of optimized implementation is a laborious process that requires developing non-trivial solutions. There is a limit to this process, beyond which optimizations within an implementation category are impossible. In this chapter, we introduce general *lower bound* proofs on metadata overhead, which identify the limits, i.e., we present *impossibility* results. Lower bounds show that some of the presented implementations, and in particular our optimized designs, are *optimal*.

In Section 4.1, we present a technique for proving a lower bound on metadata overhead that applies to *any* state-based implementation of a certain type. In Section 4.2, we apply it to prove optimality of four state-based implementations, and simply lower bounds for two other data types. In Section 4.3, we summarize both positive and impossibility results from both chapters. Together, they offer a comprehensive view of the metadata overhead problem.

Although we present optimization and impossibility results separately, as an intellectual process, they influenced one another. Specifically, designing optimized implementations helped us with lower bound proofs. Conversely, proofs directed us towards some optimizations. Generally, they help to understand the limiting factors in the semantics and in the implementation category.

4.1 Proof Technique

Our previous proofs of lower and upper bounds for a single concrete implementation rely on relatively standard techniques. (Which is why we deferred all prior proofs to the appendix). However, proving a lower bound that applies to *any* implementation is challenging.

4.1.1 Experiment Family

The goal is to show that for any correct state-based implementation $\mathcal{D}_\tau$ (that is, such that $\mathcal{D}_\tau$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_\tau$), the object state must store some minimum amount of information. We achieve this by constructing an **experiment family**, which is a collection of executions $\mathbf{C}_\alpha$ of the same size, where $\alpha \in Q$ is an **experiment index** from some **index set** Q, that identifies an individual experiment. Each experiment execution in the collection ends with a distinguished **culprit read** event $\mathbf{e}_\alpha \in \mathbf{C}_\alpha.E$, where the overhead argument applies. Culprit reads execute in the **culprit object states**, noted $\mathsf{state}(\mathbf{e}_\alpha)$. The experiments are designed in such a way that the object states must be distinct across experiments, which then implies a lower bound $\lg_{|\Lambda|} |Q|$ on the size of their encoding (i.e., the size required to encode an element of a set of cardinality $|Q|$).

To prove that the culprit states are distinct, we construct black-box tests that execute the methods of $\mathcal{D}_\tau$ on the states and show that the tests must produce different results for each $\mathsf{state}(\mathbf{e}_\alpha)$ provided $\mathcal{D}_\tau$ is correct. In other words, the tests demonstrate that these states are *distinguishable* by future steps of an execution, after the read. Formally, the tests induce a **read-back function** that satisfies $\mathsf{readback}(\mathsf{state}(\mathbf{e}_\alpha)) = \alpha$, relying on the data type *specification*.

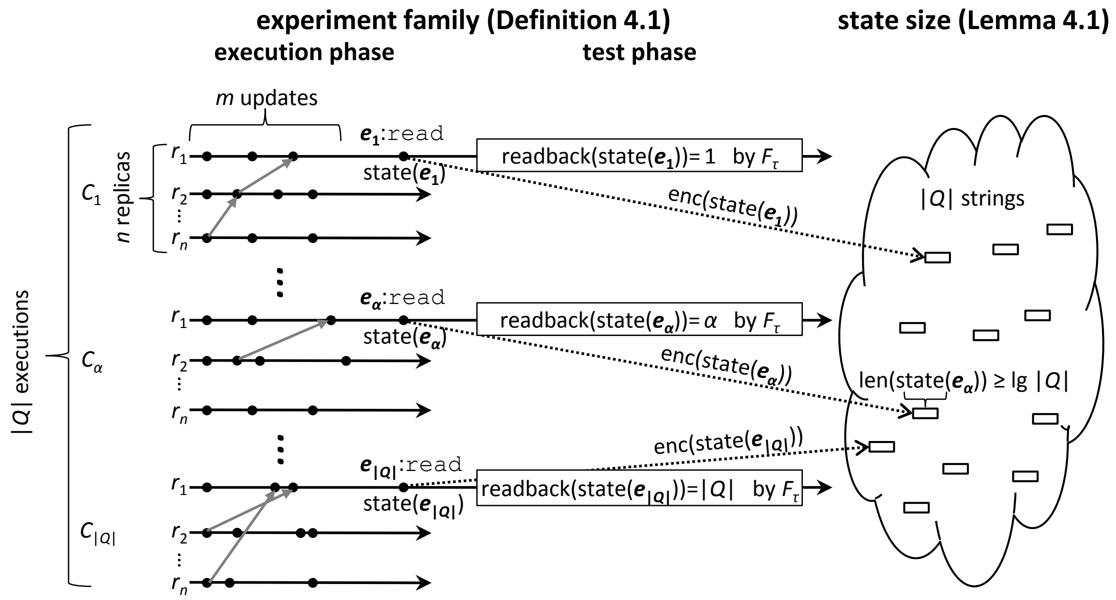


Figure 4.1: Structure of experiment family and implications of Lemma 4.1 on the size of object state encoding. For simplicity, the index set Q is a set of positive integers.

We encapsulate this core argument in the following definition and lemma.

Definition 4.1 (Experiment Family). An **experiment family** for an implementation $\mathcal{D}_\tau$ is a tuple $(Q, n, m, \mathbf{C}, \mathbf{e}, \mathsf{readback})$ where Q is a finite **index set**, $2 \le n \le m$, and for each **experiment index** $\alpha \in Q$, $\mathbf{C}_\alpha \in [\![\mathcal{D}_\tau]\!]$ is an **experiment** execution with n replicas and m updates, $\mathbf{e}_\alpha \in \mathbf{C}_\alpha.E|_{\mathtt{read}}$ is a **culprit read** in that execution and $\mathsf{readback} : \mathcal{D}_\tau.\Sigma \to Q$ is a **readback** function satisfying $\mathsf{readback}(\mathsf{state}(\mathbf{e}_\alpha)) = \alpha$.

Lemma 4.1. *If* $(Q, n, m, \mathbf{C}, \mathbf{e}, \mathsf{readback})$ *is an experiment family for an implementation* $\mathcal{D}_\tau$*, then the worst-case metadata overhead of* $\mathcal{D}_\tau$ *is at least:*

$$\mathsf{wcmo}(\mathcal{D}_\tau, n, m) \ge \lfloor \lg_{|\Lambda|} |Q| \rfloor / (\max_{\alpha \in Q} \mathsf{len}(\mathbf{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha))).$$

Proof. Since $\mathsf{readback}(\mathsf{state}(\mathbf{e}_\alpha)) = \alpha$, the states $\mathsf{state}(\mathbf{e}_\alpha)$ are all distinct and so are their encodings $\mathsf{enc}(\mathsf{state}(\mathbf{e}_\alpha))$, by injectivity. Since there are fewer than $|Q|$ strings of length strictly less than $\lfloor \lg_{|\Lambda|} |Q| \rfloor$, for some experiment $\alpha \in Q$ we have $\mathsf{len}(\mathsf{enc}(\mathsf{state}(\mathbf{e}_\alpha))) \ge \lfloor \lg_{|\Lambda|} |Q| \rfloor$. Then

$$\mathsf{wcmo}(\mathcal{D}_\tau, n, m) \ge \mathsf{mmo}(\mathcal{D}_\tau, \mathbf{C}_\alpha) \ge \frac{\mathsf{len}(\mathsf{state}(\mathbf{e}_\alpha))}{\mathsf{len}(\mathbf{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha))} \ge \frac{\lfloor \lg_{|\Lambda|} |Q| \rfloor}{\max_{\alpha' \in Q} \mathsf{len}(\mathbf{C}_{\alpha'}.\mathsf{rval}(\mathbf{e}_{\alpha'}))}.$$

□

The lemma indicates that some culprit read in the experiment family is responsible for the worst-case metadata overhead related to the size of the family and of the returned value.

Figure 4.1 illustrates the structure of an experiment family, and shows the consequences of Lemma 4.1 on the state space complexity.

To apply Lemma 4.1 to the best effect, we need to find experiment families with $|Q|$ as large as possible and $\mathsf{len}(\mathbf{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha))$ as small as possible. Finding such families for a specific data type is challenging, as we have no systematic way to derive them. When searching for experiment families for a given specification, we rely on intuitions about which situations force replicas to store a lot of information.

4.1.2 Driver Programs

To simplify the presentation, we avoid specifying an experiment family directly as a sequence of concrete execution steps, by defining it indirectly using a **driver program** that generate these execution steps (e.g., see Table 4.1). A driver program is written in imperative pseudocode with three types of instructions, each of which triggers a uniquely-determined configuration transition:

- $v \leftarrow \mathsf{do}_r\ o^{\ t}$ Do operation o at replica r with timestamp t, assign the return value to v.
- $\mathsf{send}_r(mid)$ Send a message with identifier mid at replica r.
- $\mathsf{deliver}_r(mid)$ Deliver the message with identifier mid at replica r.

We structure code into procedures, and define a program as a sequence of procedure calls.

When a driver program terminates, it may produce return value from the last procedure. For a program P, an implementation $\mathcal{D}_\tau$, and a configuration (R,N), we let $\mathsf{exec}(\mathcal{D}_\tau,(R,N),P)$ be the concrete execution of the data type implementation $\mathcal{D}_\tau$ starting in (R,N) that results from running P; we define $\mathsf{result}(\mathcal{D}_\tau,(R,N),P)$ as the return value of P in this run.

Programs explicitly supply timestamps for the do instruction and message identifiers for send and deliver instructions. We require that they do this correctly, i.e., that they respect the uniqueness of timestamps, message IDs, and deliver only previously-sent messages. This is the case for all our driver programs.[1] For most of our programs, timestamps (arbitration) are irrelevant, as they concern visibility-based data types, but we include them for completeness and make use of them for arbitration-based data types.

4.2 Lower Bounds

In this section we apply the above proof technique to obtain lower bounds for six data types.

4.2.1 Counter

We begin with one of the simplest examples: the counter data type (`Ctr`).

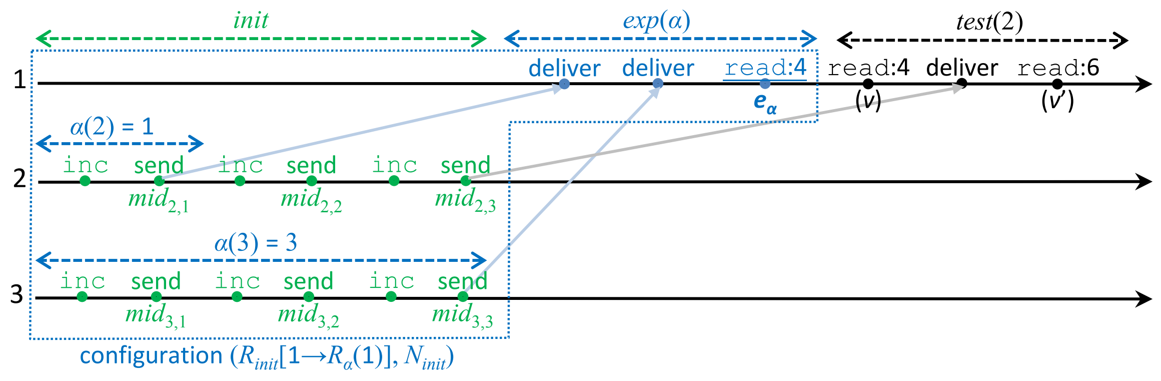
Theorem 4.1. *If* $\mathcal{D}_{\mathtt{Ctr}}\ \mathsf{sat}[\mathcal{V}^{\mathsf{state}},\mathcal{T}^{\mathsf{any}}]\ \mathcal{F}_{\mathtt{Ctr}}$*, then the complexity of* $\mathcal{D}_{\mathtt{Ctr}}$ *is* $\widehat{\Omega}(n)$*.*

We start by formulating a suitable experiment family.

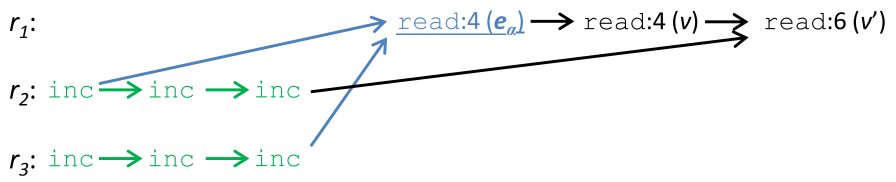
[1]Moreover, all executions generated by our driver programs respect causal timestamps $\mathcal{T}^{\mathsf{ct}}$. Therefore, the results hold also under a stronger network specification.

Conditions on #replicas & #updates	$m \geq n \geq 2 \quad \wedge \quad m \bmod (n-1) = 0$
Index set	$Q = ([2..n] \to [1..\frac{m}{n-1}])$
Family size	$\lvert Q \rvert = (\frac{m}{n-1})^{n-1}$
Driver programs	1: **procedure** *init* 2: **for all** $r \in [2..n]$ **do** 3: **for all** $i \in [1..\frac{m}{n-1}]$ **do** 4: do_r `inc` $^{rm+i}$ 5: $\mathsf{send}_r(mid_{r,i})$ 6: **procedure** $exp(\alpha)$ 7: **for all** $r \in [2..n]$ **do** 8: $\mathsf{deliver}_1(mid_{r,\alpha(r)})$ 9: do_1 `read` $^{(n+2)m} \triangleright \mathbf{e}_\alpha$ 10: $\triangleright$ (culprit read) 11: **procedure** $test(r)$ 12: $v \leftarrow \mathsf{do}_1$ `read` $^{(n+3)m}$ 13: $\mathsf{deliver}_1(mid_{r,\frac{m}{n-1}})$ 14: $v' \leftarrow \mathsf{do}_1$ `read` $^{(n+4)m}$ 15: **return** $\frac{m}{n-1} - (v' - v)$
Definition of exp. execution $\alpha \in Q$	$\mathbf{C}_\alpha = \mathsf{exec}(\mathcal{D}_\tau, (R_0, N_0), init; exp(\alpha))$ where $(R_0, N_0) = (\mathcal{D}_\tau.\mathsf{initialize}, \emptyset)$
Definition of read-back function $\mathsf{readback} : \mathcal{D}_\tau.\Sigma \to Q$	$\mathsf{readback}(\sigma) = \lambda r : [2..n].\mathsf{result}(\mathcal{D}_\tau, (R_{init}[1 \mapsto \sigma], N_{init}), test(r))$ where $(R_{init}, N_{init}) = \mathsf{post}(\mathsf{exec}(\mathcal{D}_\tau, (R_0, N_0), init))$

Table 4.1: Experiment family $(Q, n, m, \mathbf{C}, \mathbf{e}, \mathsf{readback})$ used in the lower bound proof for counter (`Ctr`).



(A) Concrete execution C'_α.



(B) Witness abstract execution $\mathsf{abs}(C'_\alpha, \mathcal{V}^{\mathsf{state}})$. Arrows indicate visibility. Arbitration is not shown.

Figure 4.2: Example experiment ($n = 3$ and $m = 6$) and test for counter (`Ctr`). Dashed lines and colors indicate phases of the experiment; blue dotted shape represents the configuration where the *test* driver program is applied to read-back $\alpha(2)$; underlined read indicates the culprit read.

Lemma 4.2. *If* $\mathcal{D}_{\mathtt{Ctr}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\mathtt{Ctr}}$*,* $n \geq 2$ *and* $m \geq n$ *is a multiple of* $(n-1)$*, then tuple* $(Q, n, m, \mathbf{C}, \mathbf{e}, \mathsf{readback})$ *as defined in Table 4.1 is an experiment family.*

Informally, the idea of the experiments is to force replica 1 to remember one number for each of the other replicas in the system, which then introduces an overhead proportional to n; cf.

the optimal implementation in Algorithm 2.3. We show one experiment execution in Figure 4.2, together with the visibility witness for that execution. All experiments start with a common *initialization phase*, defined by *init*, where each of the replicas $2..n$ performs $m/(n-1)$ increments and sends a message after each increment. All messages remain undelivered until the *unique experiment phase*, defined by $exp(\alpha)$. There, replica 1 delivers exactly one message from each replica $r = 2..n$, selected using $\alpha(r)$. An experiment concludes with the culprit read $\mathbf{e}_\alpha$ on replica 1.

The role of *read-back* is to determine the experiment index provided the culprit state of replica 1 in some experiment. Read-back works by performing separate tests for each of the replicas $r = 2..n$, defined by $test(r)$, i.e., it reads-back the index piece by piece. For example, to determine which message was sent by replica 2 during the experiment in Figure 4.2, the program $test(2)$: reads the counter value at replica 1, getting 4; delivers the final message by replica 2 to it; and reads the counter value at replica 1 again, getting 6. By observing the difference, the program can determine which message from replica 2 was sent during the experiment: $\alpha(2) = 3-(6-4) = 1$. Note that all messages delivered during the read-back were sent during the initialization phase, which is common to all executions.

Proof of Lemma 4.2. The only nontrivial obligation is to prove $\mathsf{readback}(\mathsf{state}(\mathbf{e}_\alpha)) = \alpha$. In the following steps, we first demonstrate that applying program *test* in the final configuration of an experiment produces the experiment index, and later that the *test* program does not rely on any other information than the final state of replica 1.

Let the final configuration after the experiment phase be $(R_\alpha, N_\alpha) = \mathsf{final}(\mathbf{C}_\alpha)$. Then

$$\begin{aligned}\alpha(r) &\overset{(i)}{=} \mathsf{result}(\mathcal{D}_{\mathtt{Ctr}}, (R_0, N_0), (init; exp(\alpha); test(r))) \quad = \mathsf{result}(\mathcal{D}_{\mathtt{Ctr}}, (R_\alpha, N_\alpha), test(r)) \\ &\overset{(ii)}{=} \mathsf{result}(\mathcal{D}_{\mathtt{Ctr}}, (R_{init}[1 \mapsto R_\alpha(1)], N_{init}), test(r)) = \mathsf{readback}(R_\alpha(1))(r) = \mathsf{readback}(\mathsf{state}(\mathbf{e}_\alpha))(r),\end{aligned}$$

where:

(i) This is due to $\mathcal{D}_{\mathtt{Ctr}}\ \mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]\ \mathcal{F}_{\mathtt{Ctr}}$, as we explained informally above. Let

$$C'_\alpha = \mathsf{exec}(\mathcal{D}_{\mathtt{Ctr}}, (R_0, N_0), (init; exp(\alpha); test(r)))$$

be an extension of execution $\mathbf{C}_\alpha$ with the test program steps (e.g., Figure 4.2A). Consider the witness abstract execution $\mathsf{abs}(C'_\alpha, \mathcal{V}^{\mathsf{state}})$ (e.g., Figure 4.2B). Then the operation context of the first read in $test(r)$ in the abstract execution contains $\sum_{r=2}^{n} \alpha(r)$ increments, while that of the second read contains $(m/(n-1)) - \alpha(r)$ more increments. A correct implementation $\mathcal{D}_{\mathtt{Ctr}}$ must return the number of visible increments.

(ii) We have $N_\alpha = N_{init}$ because $exp(\alpha)$ does not send any messages. Also, R_α and $R_{init}[1 \mapsto R_\alpha(1)]$ can differ only in the states of the replicas $2..n$. These cannot influence the run of $test(r)$, since it performs execution steps on replica 1 only.

□

Proof of Theorem 4.1. To fulfill the definition of lower bound $\widehat{\Omega}$ (Definition 3.2), for any given n_0, m_0, we pick $n = n_0$ and some $m \geq n_0$ such that m is a multiple of $(n-1)$ and $m \geq n^2$. Take the experiment family $(Q, n, m, \mathbf{C}, \mathbf{e}, \mathsf{readback})$ given by Lemma 4.2. Then for any α, $\mathbf{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha)$ is at most the total number of increments m in $\mathbf{C}_\alpha$, by $\mathcal{F}_{\mathtt{Ctr}}$. Using Lemma 4.1 and $m \geq n^2$, for some constants K_1, K_2, K_3, K independent from n_0, m_0 we get:

$$\begin{aligned}\mathsf{wcmo}(\mathcal{D}_{\mathtt{Ctr}}, n, m) \geq & \lfloor \lg_{|\Lambda|} |Q| \rfloor / (\max_{\alpha \in Q} \mathsf{len}(\mathbf{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha))) \\ \geq & K_1 \frac{\lg_{|\Lambda|}(\frac{m}{n-1})^{n-1}}{\mathsf{len}_{\mathbb{N}}(m)} \geq K_2 \frac{n \lg(m/n)}{\lg m} \geq K_3 \frac{n \lg \sqrt{m}}{\lg m} \geq Kn.\end{aligned}$$

□

Our experiment family illustrates a specific case where the overhead may reach level $\widehat{\Theta}(n)$. It is not exhaustive, in the sense that many other executions with n replicas and m updates may reach the same level of overhead. Other scenarios may be more complex, e.g., involve replicas that were disconnected for a long time and communicate back, or transitive delivery of updates.

4.2.2 Add-Wins Set

Theorem 4.2. *If* $\mathcal{D}_{\mathtt{AWSet}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\mathtt{AWSet}}$*, then the complexity of* $\mathcal{D}_{\mathtt{AWSet}}$ *is* $\widehat{\Omega}(n \lg m)$*.*

Lemma 4.3. *If* $\mathcal{D}_{\mathtt{AWSet}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\mathtt{AWSet}}$*,* $n \geq 2$ *and* $m \geq n$ *is such that* $(m-1)$ *is a multiple of* $(n-1)$*, then the tuple* $(Q, n, m, \mathbf{C}, \mathbf{e}, \mathsf{readback})$ *in Table 4.2 is an experiment family.*

The main idea of the experiments defined in this lemma is to force replica 1 to remember the number of observed element instances even after they have been removed at that replica; cf. the optimized add-wins set implementation from Section 3.2.1.1. The experiments follow a similar pattern to those for `Ctr`, but use different operations. In the *init* phase, each replica $2..n$ performs $\frac{m-1}{n-1}$ operations, adding a designated element 0, which are interleaved with sending messages. In the experiment phase *exp*(α), one message from each replica $r = 2..n$, selected by $\alpha(r)$, is delivered to replica 1. At the end of execution, replica 1 removes 0 from the set and performs the culprit read $\mathbf{e}_\alpha$. The return value of this read is always the empty set, by $\mathcal{F}_{\mathtt{AWSet}}$.

To perform the read-back of $\alpha(r)$ for replica $r = 2..n$, *test*(r) delivers all messages by replica r to replica 1 in the order they were sent and, after each such delivery, checks if replica 1 now reports the element 0 as part of the set. From $\mathcal{D}_{\mathtt{AWSet}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\mathtt{AWSet}}$ and the definition of $\mathcal{F}_{\mathtt{AWSet}}$ from Equation 2.10, we get that exactly the first $\alpha(r)$ such deliveries will have no effect on the contents of the set: the respective add operations have already been observed by the remove operation that replica 1 performed in the experiment phase. Thus, if 0 appears in the set right after delivering the i-th message of replica r, then $\alpha(r) = i - 1$. If 0 does not appear by the time the loop is finished, then $\alpha(r) = (m-1)/(n-1)$.

For example, consider an experiment and *test*(2) in Figure 4.3. The read-back delivers the first message from replica 2 at replica 1, and reads value $\emptyset$, since all visible adds are covered

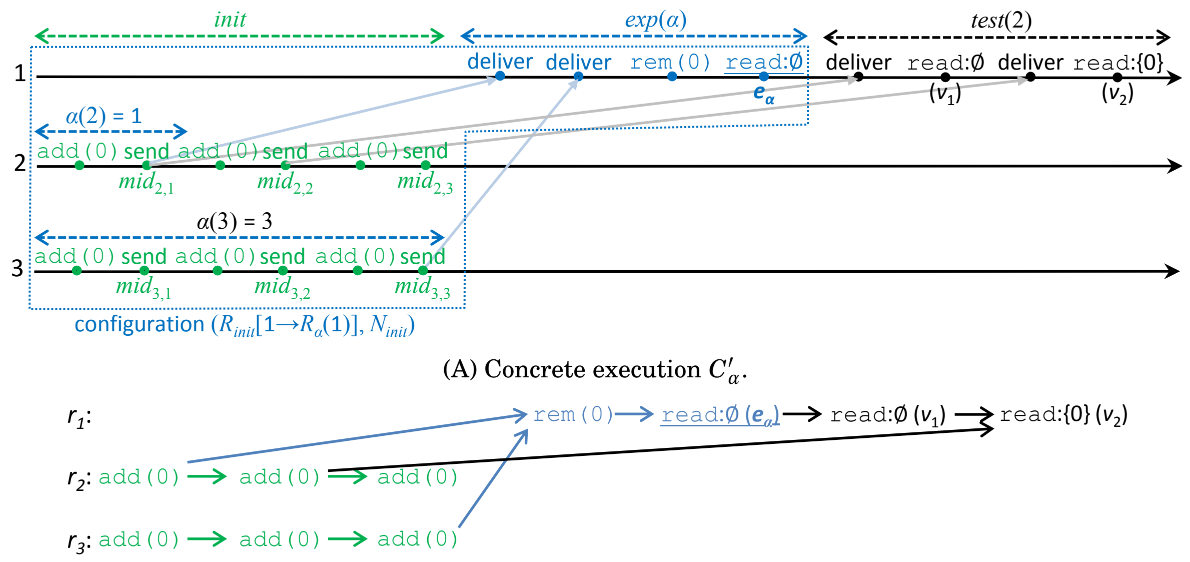
Conditions on #replicas & #updates	$m \geq n \geq 2 \quad \wedge \quad (m-1) \bmod (n-1) = 0$
Index set	$Q = ([2..n] \to [1..\frac{m-1}{n-1}])$
Family size	$\lvert Q \rvert = (\frac{m-1}{n-1})^{n-1}$

Driver programs

```
1:  procedure init
2:      for all $r \in [2..n]$ do
3:          for all $i \in [1..\frac{m-1}{n-1}]$ do
4:              $\mathsf{do}_r\ \mathsf{add}(0)^{\,rm+i}$
5:              $\mathsf{send}_r(mid_{r,i})$

6:  procedure exp($\alpha$)
7:      for all $r \in [2..n]$ do
8:          $\mathsf{deliver}_1(mid_{r,\alpha(r)})$
9:      $\mathsf{do}_1\ \mathtt{rem}(0)^{\,(n+2)m}$
10:     $\mathsf{do}_1\ \mathtt{read}^{\,(n+3)m}$ ▷ $\mathbf{e}_\alpha$
11:     ▷ (culprit read)

12: procedure test($r$)
13:     for all $i \in [1..(\frac{m-1}{n-1})]$ do
14:         $\mathsf{deliver}_1(mid_{r,i})$
15:         $v \leftarrow \mathsf{do}_1\ \mathtt{read}^{\,(n+4)m+i}$
16:         if $0 \in v$ then
17:             return $i-1$
18:     return $\frac{m-1}{n-1}$
```

Definition of exp. execution $\alpha \in Q$	$\mathbf{C}_\alpha = \mathsf{exec}(\mathcal{D}_\tau, (R_0, N_0), init; exp(\alpha))$ where $(R_0, N_0) = (\mathcal{D}_\tau.\mathsf{initialize}, \emptyset)$
Definition of read-back function $\mathsf{readback} : \mathcal{D}_\tau.\Sigma \to Q$	$\mathsf{readback}(\sigma) = \lambda r : [2..n].\mathsf{result}(\mathcal{D}_\tau, (R_{init}[1 \mapsto \sigma], N_{init}), test(r))$ where $(R_{init}, N_{init}) = \mathsf{post}(\mathsf{exec}(\mathcal{D}_\tau, (R_0, N_0), init))$

Table 4.2: Experiment family $(Q, n, m, \mathbf{C}, \mathbf{e}, \mathsf{readback})$ used in the lower bound proof for add-wins set (AWSet).



init
exp(α)
test(2)
1
2
3
deliver deliver rem(0) read:∅
e_α
deliver read:∅ deliver read:{0}
(v1)
(v2)
α(2) = 1
add(0) send add(0) send add(0) send
mid2,1 mid2,2 mid2,3
α(3) = 3
add(0) send add(0) send add(0) send
mid3,1 mid3,2 mid3,3
configuration (R_init[1→R_α(1)], N_init)
r1: rem(0) read:∅ (e_α) read:∅ (v1) read:{0} (v2)
r2: add(0) add(0) add(0)
r3: add(0) add(0) add(0)

(A) Concrete execution C'_α.

(B) Witness abstract execution $\mathsf{abs}(C'_\alpha, \mathcal{V}^{\mathsf{state}})$. Arrows indicate visibility. Arbitration is not shown.

Figure 4.3: Example experiment ($n = 3$ and $m = 7$) and test for add-wins set (AWSet). Dashed lines and colors indicate phases of the experiment; blue dotted shape represents the configuration where the *test* driver program is applied to read-back $\alpha(2)$; underlined read indicates the culprit read.

by remove. Next, it delivers the second message from replica 2 and reads again, which returns value {0} this time, because the second add(0) from replica 2, concurrent to rem(0), became visible. Therefore, the second send was the first that made element 0 appear, which indicates how many add(0) operations from replica 2 were sent to replica 1 during the experiment: $\alpha(2) = 2 - 1 = 1$.

Proof of Lemma 4.3. The only nontrivial obligation is to prove that $\mathsf{readback}(\mathsf{state}(\mathbf{e}_\alpha)) = \alpha$. Let $(R_\alpha, N_\alpha) = \mathsf{final}(\mathbf{C}_\alpha)$. Then

$$\alpha(r) \overset{(i)}{=} \mathsf{result}(\mathcal{D}_{\texttt{AWSet}}, (R_0, N_0), (\mathit{init}; \mathit{exp}(\alpha); \mathit{test}(r))) \quad = \mathsf{result}(\mathcal{D}_{\texttt{AWSet}}, (R_\alpha, N_\alpha), \mathit{test}(r))$$
$$\overset{(ii)}{=} \mathsf{result}(\mathcal{D}_{\texttt{AWSet}}, (R_{init}[1 \mapsto R_\alpha(1)], N_{init}), \mathit{test}(r)) = \mathsf{readback}(R_\alpha(1))(r) = \mathsf{readback}(\mathsf{state}(\mathbf{e}_\alpha))(r),$$

where:

(i) This is due to $\mathcal{D}_{\texttt{AWSet}}\ \mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]\ \mathcal{F}_{\texttt{AWSet}}$, as we explained informally above. Let

$$C'_\alpha = \mathsf{exec}(\mathcal{D}_{\texttt{AWSet}}, (R_0, N_0), (\mathit{init}; \mathit{exp}(\alpha); \mathit{test}(r))).$$

be an extension of experiment $\mathbf{C}_\alpha$ with the steps of *test* program (e.g., Figure 4.3A), and $\mathsf{abs}(C'_\alpha, \mathcal{V}^{\mathsf{state}})$ be the witness abstract execution of this extension (e.g., Figure 4.3B). Let v_i denote value of a read into v in the i-th iteration of the loop in the program $\mathit{test}(r)$ (Line 15). By $\mathcal{F}_{\texttt{AWSet}}$ the value of v_i is determined by the set of visible add and rem operations in the operation context, and their relation. The operation context contains:

(a) first $\max(\alpha(r), i)$ operations add(0) from replica r, and

(b) first $\alpha(r')$ operations add(0) from every replica $r' \neq r$, and

(c) and a single rem(0) made on replica 1.

We will now analyze the visibility relation between all add(0) and rem(0) events. None of the add(0) observed the rem(0) in its operation context in $\mathsf{abs}(C'_\alpha, \mathcal{V}^{\mathsf{state}})$. The operation context of rem(0) includes the first $\alpha(r')$ operations add(0) from every replica r', including replica r, i.e., it covers events (b) and part of events (a). Therefore, read into v_i observes $\max(0, i - \alpha(r))$ operations add(0) from r that were not visible to rem(0). By specification $\mathcal{F}_{\texttt{AWSet}}$ the return value of the read in this case is:

$$v_i = \begin{cases} \{0\} & \text{if } i - \alpha(r) > 0 \\ \emptyset & \text{otherwise} \end{cases}$$

i.e., {0} appears in the return value when read observes add(0) concurrent to rem(0). Therefore:

$$\alpha(r) = \min\{i \mid v_{i+1} = \{0\} \vee i = \frac{m-1}{n-1}\}.$$

(ii) We have $N_{init} = N_\alpha$ because $\mathit{exp}(\alpha)$ does not send any messages. Besides, $R_{init}[1 \mapsto R_\alpha(1)]$ and R_α can differ only in the states of the replicas $2..n$. These cannot influence the run of $\mathit{test}(r)$, since it performs execution steps on replica 1 only.

□

A proof of Theorem 4.2 is similar to the proof for the counter. Thus, we defer it to Appendix B.3. Since the remaining proofs also follow a similar structure, hereafter we focus on the intuition and their distinctive aspects, and refer the reader interested in the details to the appendix.

4.2.3 Remove-Wins Set

Theorem 4.3. *If $\mathcal{D}_{\texttt{RWSet}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\texttt{RWSet}}$, then the complexity of $\mathcal{D}_{\texttt{RWSet}}$ is $\widehat{\Omega}(m)$.*

Lemma 4.4. *If $\mathcal{D}_{\texttt{RWSet}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\texttt{RWSet}}$, $m \geq n \geq 3$ and m is a multiple of 4, then the tuple $(Q, n, m, \mathbf{C}, \mathbf{e}, \mathsf{readback})$ in Table 4.3 is an experiment family.*

The main idea of the experiments is to force replica 1 to remember a bit of information about every element *removed* from the set; cf. the implementation of remove-wins set in Algorithm 3.5, which remembers even more information in tombstone instances. The structure of the experiment is different from previous examples. Since the overhead bound is independent of the number of replicas, three replicas are sufficient for the experiment; the remaining replicas perform only "dummy" read operations to fulfill the wcmo definition w.r.t. the number of replicas n. In the *init* phase, replica 2 performs $m/4$ remove operations for elements $1..m/4$, interleaved with sending messages; concurrently, replica 3 also removes all these elements, sends a message, adds all of them again, and sends another message, performing $m/2$ operations in total. Removes from replica 3 will hide and uncover elements when desired.

In the experiment, replica 1 delivers a subset of the messages with removes sent by replica 2 in the order that they were sent, and adds elements $i = 1..m/4$. The subset of messages to deliver is controlled by index α which contains a set of elements. If an element a is in α, then replica 1 delivers a message containing `rem`(a) operation from replica 2 before replica 1 adds a to the set; otherwise it adds a to the set without being aware of the concurrent remove. Eventually, replica 1 delivers a message from replica 3 that removes all elements, and performs the culprit read. The read returns an empty set, by $\mathcal{F}_{\texttt{RWSet}}$, since removes from replica 3 dominate all adds.

The read-back function retrieves the whole index α at once. First, it delivers the last messages from replicas 2 and 3 to replica 1, which includes all removes at replica 2 and all adds at replica 3, dominating previous removes from replica 3. Then it performs a read that returns the set of elements corresponding directly to the index α. To understand how it works, consider the witness abstract execution for this execution (e.g., Figure 4.3B). The read observes the complete set of add and remove operations from all replicas. The visibility relation is partially determined by the experiment index α. All removes from replica 3 are covered by adds at replica 3, whereas a subset of removes from replica 2 is covered by adds from replica 1, according to the experiment index α; remaining removes are concurrent to adds. From $\mathcal{F}_{\texttt{RWSet}}$, those elements for which there is a removes without covering add must not appear in the read return value.

For example, consider an experiment and *test* in Figure 4.4. In the experiment phase, we observe that `add`(1) and `add`(3) at replica 1 observe, respectively, `rem`(1) and `rem`(3) from replica 2, whereas `add`(2) is concurrent to `rem`(2) at replica 2. Therefore, when the read-back delivers the last message from replica 3 to replica 1, the read returns value $\{1,3\}$.

Formal proofs of Lemma 4.4 and Theorem 4.3 are in Appendix B.3.

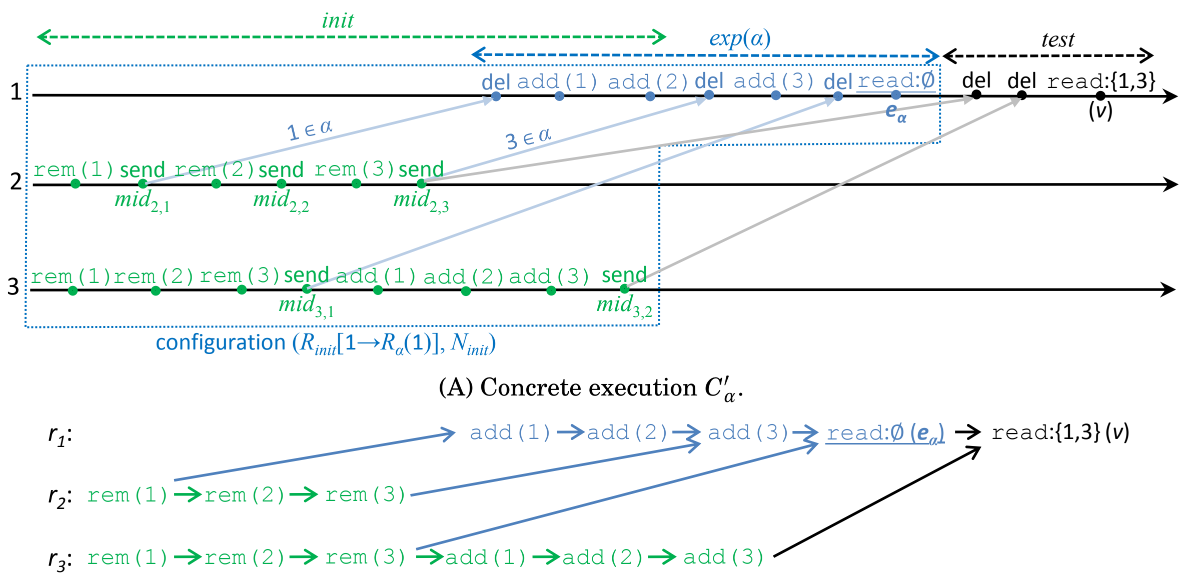
Conditions on #replicas & #updates	$m \geq n \geq 3 \quad \wedge \quad m \bmod 4 = 0$
Index set	$Q = \mathcal{P}(\{1,\ldots,m/4\})$
Family size	$\lvert Q \rvert = 2^{m/4}$

Driver programs

```
procedure $init$
    for all $i \in [1..\frac{m}{4}]$ do
        do$_2$ rem($i$) $^{i}$
        send$_2$($mid_{2,i}$)
        do$_3$ rem($i$) $^{m+i}$
    send$_3$($mid_{3,1}$)
    for all $i \in [1..\frac{m}{4}]$ do
        do$_3$ add($i$) $^{2m+i}$
    send$_3$($mid_{3,2}$)
    for all $r \in [4..n]$ do ▷ dummy
        do$_r$ read $^{3m+j}$        ▷ reads

procedure $exp(\alpha)$
    for all $i \in [1..\frac{m}{4}]$ do
        if $i \in \alpha$ then
            deliver$_1$($mid_{2,i}$)
        do$_1$ rem(0) $^{4m+i}$
    deliver$_1$($mid_{3,1}$)
    do$_1$ read $^{5m+1}$ ▷ $\mathbf{e}_\alpha$
    ▷ (culprit read)

procedure $test$
    deliver$_1$($mid_{2,m/4}$)
    deliver$_1$($mid_{3,2}$)
    $v \leftarrow$ do$_1$ read $^{4m+2}$
    return $v$
```

Definition of exp. execution $\alpha \in Q$	$\mathbf{C}_\alpha = \text{exec}(\mathcal{D}_\tau,(R_0,N_0),init;exp(\alpha))$ where $(R_0,N_0) = (\mathcal{D}_\tau.\text{initialize},\emptyset)$
Definition of read-back function readback : $\mathcal{D}_\tau.\Sigma \to Q$	$\text{readback}(\sigma) = \text{result}(\mathcal{D}_\tau,(R_{init}[1 \mapsto \sigma],N_{init}),test)$ where $(R_{init},N_{init}) = \text{post}(\text{exec}(\mathcal{D}_\tau,(R_0,N_0),init))$

Table 4.3: Experiment family $(Q,n,m,\mathbf{C},\mathbf{e},\text{readback})$ used in the lower bound proof for remove-wins set (RWSet).


init
exp(α)
test
1
2
3
del add(1) add(2) del add(3) del read:∅
del del read:{1,3}
(v)
e_α
1 ∈ α
3 ∈ α
rem(1) send rem(2) send rem(3) send
$mid_{2,1}$ $mid_{2,2}$ $mid_{2,3}$
rem(1) rem(2) rem(3) send add(1) add(2) add(3) send
$mid_{3,1}$ $mid_{3,2}$
configuration (R_{init}[1→R_α(1)], N_{init})
r_1: add(1) → add(2) → add(3) → read:∅ (e_α) → read:{1,3} (v)
r_2: rem(1) → rem(2) → rem(3)
r_3: rem(1) → rem(2) → rem(3) → add(1) → add(2) → add(3)

(A) Concrete execution C'_α.

(B) Witness abstract execution abs($C'_\alpha, \mathcal{V}^{\text{state}}$). Arrows indicate visibility. Arbitration is not shown.

Figure 4.4: Example experiment ($n = 3$ and $m = 12$) and test for remove-wins set (RWSet). Dashed lines and colors indicate phases of the experiment; blue dotted shape represents the configuration where the *test* driver program is applied to read-back α; underlined read indicates the culprit read.

4.2.4 Last-Writer-Wins Set

Theorem 4.4. *If* $\mathcal{D}_{\texttt{LWWSet}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]\, \mathcal{F}_{\texttt{LWWSet}}$*, then the complexity of* $\mathcal{D}_{\texttt{LWWSet}}$ *is* $\widehat{\Omega}(n \lg m)$*.*

Lemma 4.5. *If* $\mathcal{D}_{\texttt{LWWSet}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]\, \mathcal{F}_{\texttt{LWWSet}}$*,* $n \geq 2$ *and m is a multiple of* $2n-2$*, then the tuple* $(Q, n, m, \mathbf{C}, \mathbf{e}, \mathsf{readback})$ *in Table 4.4 is an experiment family.*

The idea of the experiments is to force replica 1 to remember the latest timestamp it observed from each replica, for some elements, which then introduces an overhead proportional to the $n \lg m$. The existing implementation in Algorithm 3.6 introduces higher overhead.

In the *init* phase of experiments, each replica $2..n$ performs $\frac{m}{2(n-1)}$ operations, interchangeably adding and removing the same element equal to replica ID, and sending a message after each operation. We use monotonically growing timestamps for operations at each replica, which induces their arbitration order consistent with the order of operations at each replica. All messages send in the *init* phase of the program are undelivered until the second phase, defined by *exp*(α). In the experiment phase *exp*(α), one message from each replica $r = 2..n$ is delivered to replica 1, selected by $2\alpha(r)$. This message includes the first $\alpha(r)$ add operations and the first $\alpha(r)$ remove operations at replica r. The experiment concludes with the culprit read $\mathbf{e}_\alpha$. The read returns an empty set, by $\mathcal{F}_{\texttt{LWWSet}}$, since among the visible operations, removes are the latest in arbitration for each element.

To perform the read-back of $\alpha(r)$ for replica $r = 2..n$, *test*(r) delivers all odd messages by replica r to replica 1 in the order they were sent. After delivery of each message, the test checks if replica 1 now reports the element r as part of the set. Each odd message includes one add operation that is not covered by subsequent remove at replica 1. From $\mathcal{D}_{\texttt{LWWSet}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]\, \mathcal{F}_{\texttt{LWWSet}}$ and the definition of $\mathcal{F}_{\texttt{LWWSet}}$, we get that exactly the first $\alpha(r)$ such deliveries will have no effect on the contents of the set: they deliver $\texttt{add}(r)$ operations that have been already observed by the $\texttt{rem}(r)$ operation (delivered at replica 1 in the experiment phase) that dominates them in the arbitration order. Thus, if element r appears in the set right after delivering the i-th message of replica r, then $\alpha(r) = i - 1$. If r does not appear by the time the loop is finished, then $\alpha(r) = m/(2n-2)$.

For example, consider an experiment and *test*(2) in Figure 4.5. The read-back delivers the first message from replica 2 at replica 1, and reads value $\emptyset$, since all visible adds are covered in arbitration by visible remove; next, it delivers the second odd (third overall) message from replica 2 and reads again, which returns value $\{2\}$ this time, because the second $\texttt{add}(2)$ from replica 2 is the latest operation in arbitration among all the visible operations on element 2. Therefore, the second send was the first that made element 2 appear, which indicates how many $\texttt{rem}(2)$ operations from replica 2 was delivered to replica 1 during the experiment: $\alpha(2) = 2 - 1 = 1$.

Formal proofs of Lemma 4.5 and Theorem 3.7 are in Appendix B.3.

4.2.5 Multi-Value Register

Theorem 4.5. *If* $\mathcal{D}_{\texttt{MVReg}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]\, \mathcal{F}_{\texttt{MVReg}}$*, then the complexity of* $\mathcal{D}_{\texttt{MVReg}}$ *is* $\widehat{\Omega}(n \lg m)$*.*

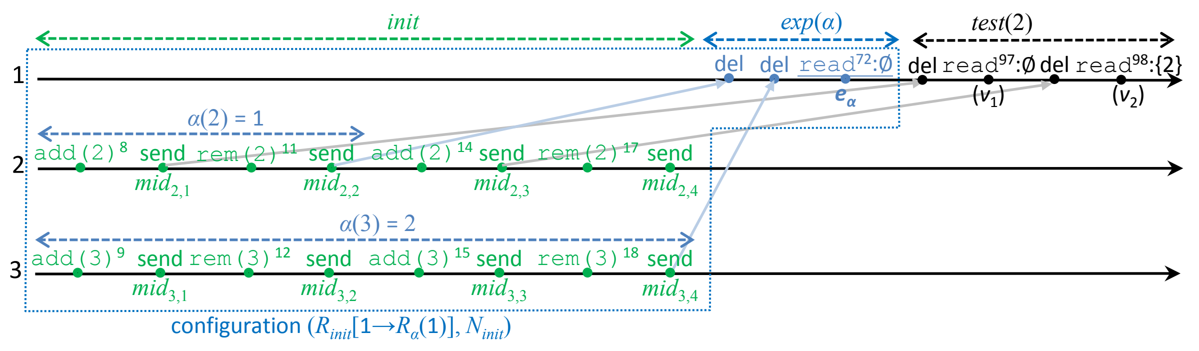
Conditions on #replicas & #updates	$m \geq n \geq 2 \quad \wedge \quad m \bmod (2n-2) = 0$
Index set	$Q = ([2..n] \to [1..\frac{m}{2n-2}])$
Family size	$\lvert Q \rvert = (\frac{m}{2n-2})^{n-1}$

Driver programs

```
1:  procedure $init$
2:      for all $r \in [2..n]$ do
3:          for all $i \in [1..\frac{m}{2n-2}]$ do
4:              $\mathsf{do}_r$ add($r$) $^{2ni+r}$
5:              $\mathsf{send}_r(mid_{r,2i-1})$
6:              $\mathsf{do}_r$ rem($r$) $^{2ni+n+r}$
7:              $\mathsf{send}_r(mid_{r,2i})$

8:  procedure $exp(\alpha)$
9:      for all $r \in [2..n]$ do
10:         $\mathsf{deliver}_1(mid_{r,2\alpha(r)})$
11:     $\mathsf{do}_1$ read $^{3nm}$ $\triangleright$ $\mathbf{e}_\alpha$
12:     $\triangleright$ (culprit read)

12: procedure $test(r)$
13:     for all $i \in [1..(\frac{m}{2n-2})]$ do
14:         $\mathsf{deliver}_1(mid_{r,2i-1})$
15:         $v \leftarrow \mathsf{do}_1$ read $^{4nm+i}$
16:         if $r \in v$ then
17:             return $i-1$
18:     return $\frac{m}{2n-2}$
```

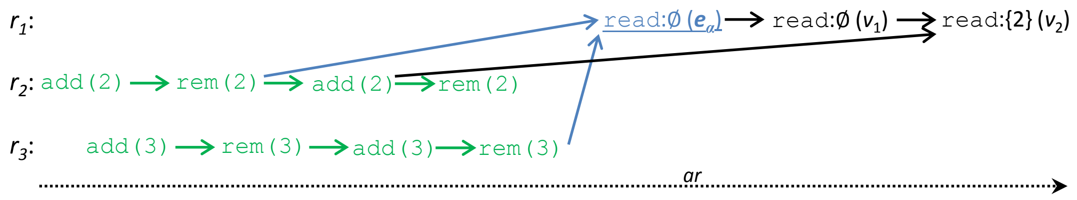
Definition of exp. execution $\alpha \in Q$	$\mathbf{C}_\alpha = \mathsf{exec}(\mathcal{D}_\tau, (R_0, N_0), init; exp(\alpha))$ where $(R_0, N_0) = (\mathcal{D}_\tau.\mathsf{initialize}, \emptyset)$
Definition of read-back function $\mathsf{readback} : \mathcal{D}_\tau.\Sigma \to Q$	$\mathsf{readback}(\sigma) = \lambda r : [2..n].\mathsf{result}(\mathcal{D}_\tau, (R_{init}[1 \mapsto \sigma], N_{init}), test(r))$ where $(R_{init}, N_{init}) = \mathsf{post}(\mathsf{exec}(\mathcal{D}_\tau, (R_0, N_0), init))$

Table 4.4: Experiment family $(Q, n, m, \mathbf{C}, \mathbf{e}, \mathsf{readback})$ used in the lower bound proof for last-writer-wins set (LWWSet).



init
exp(α)
test(2)
1
2
3
del del read^72:∅
e_α
del read^97:∅ del read^98:{2}
(v_1)
(v_2)
α(2) = 1
add(2)^8 send rem(2)^11 send add(2)^14 send rem(2)^17 send
mid_2,1 mid_2,2 mid_2,3 mid_2,4
α(3) = 2
add(3)^9 send rem(3)^12 send add(3)^15 send rem(3)^18 send
mid_3,1 mid_3,2 mid_3,3 mid_3,4
configuration (R_init[1→R_α(1)], N_init)

(A) Concrete execution C'_α.



r_1:
read:∅ (e_α)
read:∅ (v_1)
read:{2} (v_2)
r_2: add(2) rem(2) add(2) rem(2)
r_3: add(3) rem(3) add(3) rem(3)
ar

(B) Witness abstract execution $\mathsf{abs}(C'_\alpha, \mathcal{V}^{\mathsf{state}})$. Solid arrows indicate visibility. Horizontal position indicates arbitration order.

Figure 4.5: Example experiment ($n = 3$ and $m = 8$) and test for last-writer-wins set (LWWSet). Dashed lines and colors indicate phases of the experiment; blue dotted shape represents the configuration where the *test* driver program is applied to read-back $\alpha(2)$; underlined read indicates the culprit read.

Lemma 4.6. *If* $\mathcal{D}_{\mathtt{MVReg}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\mathtt{MVReg}}$, *$n \geq 2$ and $m \geq n$ is such that $(m-1)$ is a multiple of $(n-1)$, then the tuple* $(Q, n, m, \mathbf{C}, \mathbf{e}, \mathsf{readback})$ *in Table 4.5 is an experiment family.*

The experiments are similar to the add-wins set experiments. The idea is to force replica 1 to remember a number of `write` operations from each replica dominated by the current `write`, which then introduces an overhead proportional to $n \lg m$; cf. the optimized implementation in Algorithm 3.4. All experiments start with the *init* phase, where each of the replicas $2..n$ performs $(m-1)/(n-1)$ writes `write`(0) and sends a message after each write. All messages remain undelivered until the second phase, defined by $exp(\alpha)$. There replica 1 delivers exactly one message from each replica $r = 2..n$, selected using $\alpha(r)$, and subsequently overwrites the register with `write`(1). An experiment concludes with the culprit read $\mathbf{e}_\alpha$ on the first replica that returns the result of the last write, {1}.

The read-back works by performing separate tests for each replica $r = 2..n$, defined by $test(r)$. To determine which message was sent by replica r during the experiment, the program $test(r)$ enforces (re)delivery of all messages sent by replica r to replica 1, in the order they were sent, and performs a read after each delivery. By inspecting the return value of each such read, the program identifies the first message that causes read to return both values 0 and 1. By $\mathcal{F}_{\mathtt{MVReg}}$, this corresponds to observing `write`(0) performed at r that was not visible at the time of `write`(1), i.e., that is concurrent to `write`(1); the index of this message corresponds to $\alpha(r)+1$.

For example, consider an experiment and $test(2)$ in Figure 4.6. The read-back delivers the first message from replica 2 at replica 1, and reads value {1}, since all visible `write`(0) are covered by `write`(1); next, it delivers the second message from replica 2 and reads again, which returns value $\{0,1\}$ this time, because the second `write`(0) from replica 2, concurrent to `write`(1), became visible. Therefore, the second send was the first that made value 1 appear, which indicates how many `write`(0) operations from replica 2 were sent to replica 1 during the experiment: $\alpha(2) = 2 - 1 = 1$.

Formal proofs of Lemma 4.6 and Theorem 4.5 are in Appendix B.3.

4.2.6 Last-Writer-Wins Register

Theorem 4.6. *If* $\mathcal{D}_{\mathtt{LWWReg}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\mathtt{LWWReg}}$, *then the complexity of* $\mathcal{D}_{\mathtt{LWWReg}}$ *is* $\widehat{\Omega}(\lg m)$.

A proof is Appendix B.3. It is a specific case of Theorem 4.4, and of minor practical importance.

4.3 Summary

Table 4.6 summarizes our positive and impossibility results for all studied data types. From left to right, the table lists prior implementations and their complexity, as well as our optimizations (if any) and their complexity, and lower bounds on complexity of any implementation. We underline every implementation optimal in the sense of Definition 3.3, i.e., with an upper bound that matches a general lower bound.

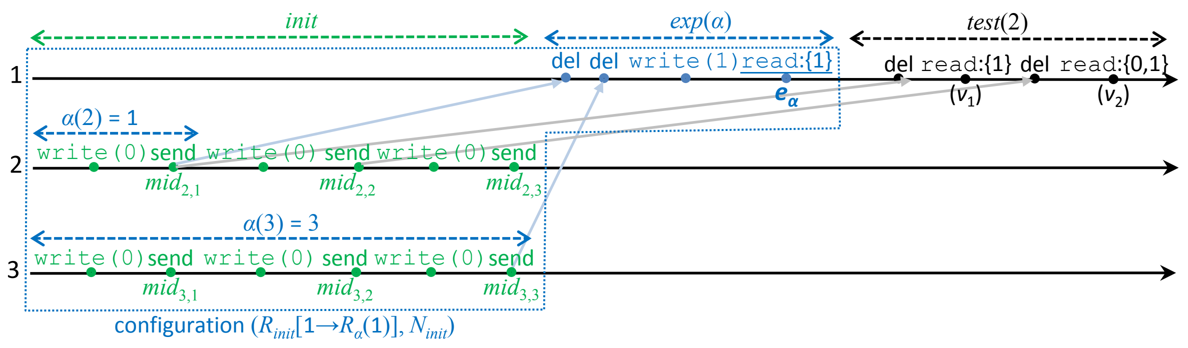
Conditions on #replicas & #updates	$m \geq n \geq 2 \quad \wedge \quad (m-1) \bmod (n-1) = 0$
Index set	$Q = ([2..n] \to [1..\frac{m-1}{n-1}])$
Family size	$\lvert Q \rvert = (\frac{m-1}{n-1})^{n-1}$

Driver programs

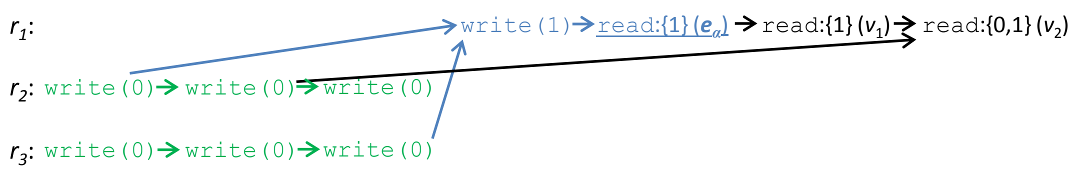
```
procedure $init$
    for all $r \in [2..n]$ do
        for all $i \in [1..\frac{m-1}{n-1}]$ do
            $\mathsf{do}_r$ write(0) $^{rm+i}$
            $\mathsf{send}_r(mid_{r,i})$

procedure $exp(\alpha)$
    for all $r \in [2..n]$ do
        $\mathsf{deliver}_1(mid_{r,\alpha(r)})$
    $\mathsf{do}_1$ write(1) $^{(n+2)m}$
    $\mathsf{do}_1$ read $^{(n+3)m}$  ▷ $\mathbf{e}_\alpha$
    ▷ (culprit read)

procedure $test(r)$
    for all $i \in [1..(\frac{m-1}{n-1})]$ do
        $\mathsf{deliver}_1(mid_{r,i})$
        $v \leftarrow \mathsf{do}_1$ read $^{(n+4)m+i}$
        if $v = \{0,1\}$ then
            return $i-1$
    return $\frac{m-1}{n-1}$
```

Definition of exp. execution $\alpha \in Q$	$\mathbf{C}_\alpha = \mathsf{exec}(\mathcal{D}_\tau, (R_0, N_0), init; exp(\alpha))$ where $(R_0, N_0) = (\mathcal{D}_\tau.\mathsf{initialize}, \emptyset)$
Definition of read-back function $\mathsf{readback} : \mathcal{D}_\tau.\Sigma \to Q$	$\mathsf{readback}(\sigma) = \lambda r : [2..n].\mathsf{result}(\mathcal{D}_\tau, (R_{init}[1 \mapsto \sigma], N_{init}), test(r))$ where $(R_{init}, N_{init}) = \mathsf{post}(\mathsf{exec}(\mathcal{D}_\tau, (R_0, N_0), init))$

Table 4.5: Experiment family $(Q, n, m, \mathbf{C}, \mathbf{e}, \mathsf{readback})$ used in the lower bound proof for multi-value register (MVReg).


init
exp(α)
test(2)
1
2
3
del del write(1) read:{1}
e_α
del read:{1} del read:{0,1}
(v_1)
(v_2)
α(2) = 1
write(0) send write(0) send write(0) send
$mid_{2,1}$ $mid_{2,2}$ $mid_{2,3}$
α(3) = 3
write(0) send write(0) send write(0) send
$mid_{3,1}$ $mid_{3,2}$ $mid_{3,3}$
configuration ($R_{init}[1\to R_\alpha(1)]$, N_{init})

(A) Concrete execution C'_α.


r_1: write(1) → read:{1} (e_α) → read:{1} (v_1) → read:{0,1} (v_2)
r_2: write(0) → write(0) → write(0)
r_3: write(0) → write(0) → write(0)

(B) Witness abstract execution $\mathsf{abs}(C'_\alpha, \mathcal{V}^{\mathsf{state}})$. Arrows indicate visibility. Arbitration is not shown.

Figure 4.6: Example experiment ($n = 3$ and $m = 7$) and test for multi-value register (MVReg). Dashed lines and colors indicate phases of the experiment; blue dotted shape represents the configuration where the *test* driver program is applied to read-back $\alpha(2)$; underlined read indicates the culprit read.

Data type	**Prior implementation**		**Optimized implementation**		**Any impl. lower bound**
	Definition	**Bounds**	**Definition**	**Bounds**	
`Ctr`	Alg. 2.3 [92]	$\widehat{\Theta}(n)$	—	—	$\widehat{\Omega}(n)$
`AWSet`	Alg. 3.1 [93]	$\widehat{\Theta}(m \lg m)$	new: Alg. 3.2	$\widehat{\Theta}(n \lg m)$	$\widehat{\Omega}(n \lg m)$
`RWSet`	Alg. 3.5 [22]	$\widehat{\Omega}(m \lg m)$	—	—	$\widehat{\Omega}(m)$
`LWWSet`	Alg. 3.6 [22, 56]	$\widehat{\Omega}(m \lg m)$	—	—	$\widehat{\Omega}(n \lg m)$
`MVReg`	Alg. 2.5 [77, 93]	$\widehat{\Theta}(n^2 \lg m)$	new: Alg. 3.4	$\widehat{\Theta}(n \lg m)$	$\widehat{\Omega}(n \lg m)$
`LWWReg`	Alg. 2.5 [56, 92]	$\widehat{\Theta}(\lg m)$	—	—	$\widehat{\Omega}(\lg m)$

Table 4.6: Summary of metadata overhead results for different data types. Underlined implementations are optimal; n and m, are, respectively, the number of replicas and updates in an execution; $\widehat{\Omega}, \widehat{O}, \widehat{\Theta}$ are lower, upper and tight bounds on metadata, respectively.

For the counter data type (`Ctr`), the existing state-based implementation is optimal.

The set data type offers a range of different semantics choices, (`AWSet`,`RWSet`,`LWWSet`) which are uneven in terms of the metadata space complexity. Prior implementations all incur overhead in the order of number of updates, at least (recall that for remove-wins and LWW sets there is no upper bound, as actual elements are stored, and their size is variable). However, this is not necessary for all implementations of all variants. Our improved design of add-wins set implementation reduces this cost to the order of number of replicas, which we prove is the optimal achievable cost. Furthermore, this is asymptotically the cheapest known implementation of all three sets. This optimization is impossible for the remove-wins set, as our lower bound shows: the order of number of updates or worse is necessary. The existing implementation of the LWW set incurs similar overhead, but it is an open problem whether this is necessary. We managed neither to improve it, nor to prove it is impossible, although we conjecture the latter.

The register type also comes in different variants (`MVReg`,`LWWReg`), which too, are uneven in the metadata complexity. The existing implementation of the LWW register has negligible overhead and is the optimal one. On the contrary, the existing implementation of the multi-value register has a substantial overhead, in the square of the number of replicas, due to inefficient treatment of concurrent writes of the same value. Our optimizations alleviate the square component, and reach the asymptotically optimal metadata overhead.

Chapter 5

Related Work and Discussion

We know no more than you. . . But maybe it is worth investigating the unknown, if only because the very feeling of not knowing is a painful one.

Krzysztof Kieślowski

Contents

In previous chapters we introduced the metadata complexity problem, presented positive and impossibility results for six data types, and related them to known state-based implementations. In this chapter, we discuss the scope of these results, in reference to other related work. Although a formal model of RDTs is a recent development [34, 92], there is already a sizeable body of work on RDTs, and some older algorithms use similar techniques.

Specifically, in Section 5.1 we discuss the completeness of our results in terms of data type coverage. In Section 5.2, we discuss known and potential optimizations to state-based implementations that are not covered by our metadata metrics. In Section 5.3, we discuss our results for state-based category compared to other approaches proposed in the literature. Finally, in Section 5.4, we compare our complexity proofs to similar ones in the field of distributed computing.

5.1 Other Data Types

Our metadata analysis cannot be exhaustive in terms of data types, although we cover many important *primitive* types, from which other data types can be derived by composition [27, 52].

An important data type absent in our analysis is the *list*. A list object is a dynamically-sized totally ordered list of elements, where an element can be inserted at or removed from a designated position [48, 73, 76, 81, 85, 101]. One of the most important usages of the list type is a model of a shared document in collaborative editing applications [48, 76, 81, 101].

Implementing a list is far from trivial. There is a long history of incorrect designs [74], and specifications are implementation-driven. Moreover, all known implementations suffer from high bounds on metadata overhead, in the order of the number of updates. Every implementation stores some trace of removed elements, in order to position concurrently inserted elements correctly. This manifests in metadata either as tombstones [75, 76, 85], or as position identifiers of unbounded size [73, 81, 101]. We conjecture this is fundamental to the list semantics; the proof is a major direction of future work.

5.2 State-Based Optimizations Beyond Our Metric

Our definitions of metadata metrics (Definition 3.2) and optimality (Definition 3.3) are oriented towards worst-case asymptotic analysis and generality; so are our optimizations and lower bounds. However, there are multiple sensible ways of formulating the concepts of complexity and optimality, some of which can express optimizations that our results do not cover. In this section, we motivate our choice for the definitions, and discuss some of their properties. We identify some optimizations from the literature, and potential optimizations, not covered by our formulation, and we evaluate their significance.

5.2.1 Background Compaction of Stable Metadata

Our definition of maximum metadata overhead (Definition 3.1) concerns the worst situation in an execution. This worst case may involve relatively short-lived peaks in metadata and only a subset of replicas. This perspective is important as it models what could happen under high concurrency or failure scenarios. For instance, the case where a set of replicas is partitioned away, or a replica becomes unresponsive. It shows the capacity that the system should be planned for.

Nevertheless, there exist RDT implementations that can **compact**, or specifically **garbage collect** (GC), metadata that is not useful anymore. In particular, **stable updates** [56, 93] can often be collected. An update is stable if it is known to be replicated at all replicas. For example, in a set implementation, a stable tombstone can be discarded. When all replicas have received a given tombstone instance, then it is guaranteed that no further message will include the tombstone, except for a possibly delayed message. If furthermore, the implementation protects itself from delayed messages, a replica can discard the stable tombstone safely.

Johnson and Thomas [56] use this protocol to discard timestamps of removed entries in a LWW map, whereas Wuu and Bernstein [102] use it for a map where elements are guaranteed to be added no more than once. A list implementation by Roh et al. [85] also uses stability to collect removed list elements.

A **stability protocol** computes stable updates that can be compacted [51, 56, 102]. The stability protocol maintains information about the set of updates that each replica has received. Such information is typically encoded as a one-dimensional vector or a two-dimensional matrix.

Stability-based metadata compaction has its own drawbacks. The data structures of the stability protocol incur an overhead on their own [56, 102]. It takes time until stability is detected by the protocol. Finally, stability protocol is not live in the presence of failures, since it requires to communicate with every replica. Our model assumes a static or grow-only set of replicas and does not explicitly consider their failures. However, even under a model that explicitly considers dynamic set of replicas and their failures, it requires a perfect failure detector to safely eliminate an unresponsive replica from the set of correct replicas that must acknowledge an update.[1] In contrast, with our metadata design from Section 3.2.1, an implementation discards information right away, independent of network conditions, and does not require additional GC. Nevertheless, for some data types (e.g., the remove-wins set or the list), our optimizations are not applicable and the GC approach is the only way to decrease metadata size.

5.2.2 Finer-Grained Optimizations

Our definition of implementation optimality (Definition 3.3) is relatively coarse-grained, in the sense that the optimal implementation is not required to incur minimum overhead in every execution. Instead, it suffices that it performs well enough in all cases, incurring an overhead that

[1]Similarly, there is no other safe known way of removing an entry of a replica from the version vector in our optimizations from Section 3.2.1.

is no higher than any other implementation in the worst-case. In practice, some optimizations for cases other than worst-case are also desirable, albeit they tend to be less fundamental.

Many minor finer-grained optimizations can be considered. For example, a multi-value register does not need to store a complete vector until every replica performs at least one write.

An important exception is the recent work of Almeida et al. [6]. They improve over our optimized multi-value register implementation (Algorithm 3.4) in the case when *different* values are written concurrently. Their implementation uses a new *dotted version vector set* metadata, which consists of a single vector extended with a set of at most n scalar timestamps. The storage requirements of their register is equivalent to a single vector, regardless of the number of concurrent values stored in the state. In contrast, our optimized implementation stores one vector per value stored in the state. Both implementations are asymptotically optimal, i.e., their overhead is bounded by $\widehat{\Theta}(n \lg m)$, but the one of Almeida et al. [6] performs better for common executions. In particular, it incurs a lower size of absolute state.

As a first approach, we chose to not use a fine-grained optimality definition, suitable for such optimizations. Indeed, this could tie the definition too closely to a particular implementation or a data type. The metric would need to be type- or implementation-specific, increasing complexity and decreasing readability. Moreover, it could make optimality proofs more implementation-specific, i.e., difficult and tedious. It is an open question whether an implementation-agnostic metric that captures such optimizations exists. With our coarse-grained definition, lower bound proofs are not specific to any particular implementation (cf. Chapter 4), and the metric is general and concise.

5.2.3 Custom Timestamps

Our model and bounds require that the implementations of an arbitration-based type, such as LWW register or LWW set, use externally-provided timestamps to arbitrate updates, rather than any internal mechanism. This requirement stems from the fixed definition of witness abstract execution (Equation 2.12), which is part of the implementation correctness condition. Thanks to external timestamps all implementations can support any type of timestamp and cross-object arbitration consistency properties [34] (e.g., a consistent choice of the “last” writer among different LWW registers). However, such a lack of flexibility in timestamp choice can appear as a potential implementation restriction. Although we are not aware of any space optimizations based on the use of implementation-assigned timestamps for arbitration, our lower bound proofs are limited to implementations that do respect external timestamps. Note that all of our lower bound proofs provide only causal timestamps, so the implementations do not actually need to handle particularly adversary timestamps.

5.3 Other Implementation Categories

A number of recent implementations or implementation categories extend the concept of the state-based category. Extensions related to metadata optimization include message size optimizations and topology restrictions.

5.3.1 State-Based Implementations With Smaller Messages

Pure state-based implementations have the drawback that they transfer the complete state in each message [7]. Recent extensions address this problem, at a modest expense in state size.

Almeida et al. [7] propose a new implementation category: **state-based with deltas**. The core idea is that each replica maintains not only a complete durable object state (as in state-based), but also a smaller buffer of recent updates, called **deltas**, which does not need to be durable. The expectation is that, in the common case, the system transfers only deltas, and resorts to complete state transfer only as a fallback (e.g., when the buffer has been discarded).[2] In contrast to op-based implementation, the buffer of deltas is not just a log of independent updates, but a compressed representation of updates using their semantics. A custom but type-independent protocol is used to disseminate deltas; optionally, to maintain causality. The challenge for this category of implementations is to design deltas significantly smaller than the complete state.

Almeida et al. present two examples of efficient adaptations of existing state-based implementations to deltas. For example, the delta-based counter buffers, as a delta, only the modified entries of the increments vector (cf. Algorithm 2.3); in the common case, only a small number of vector entries are transferred rather than full vector. In the absence of causal delivery, the optimized add-wins set (Algorithm 3.2) adaptation is more challenging. Recall that the optimization uses a vector to represent a set of element instances with contiguous timestamps, but delta instances might be non-contiguous. A more general representation is needed, e.g., concise version vectors [70], or their interval-based representation [24], which support timestamp gaps in the set.

The deltas category optimizes message size, which is a problem mostly orthogonal to the state optimization problem that we considered. In examples provided by Almeida et al. [7], the reduced message size comes at cost of a slight increase in local storage size: a new non-durable delta buffer and bookkeeping metadata on successful deliveries are needed. These examples do not contradict our complexity bounds, which hold for the complete state and fall-back state transfer.

Deftu and Griebsch [45] propose a somewhat similar extension, tailored specifically to the basic add-wins set (Algorithm 3.1). Their set implementation stores all element instances until discarded by a GC protocol. The GC protocol is not specified formally. Storing instances serves a purpose of naive delta implementation (a replica can request from another replica all new in-

[2]In terms our formal model from Chapter 2, this means that there are two kinds of send and deliver events, and a new visibility witness, $\mathcal{V} \neq \mathcal{V}^{\mathsf{state}}$, treats each kind differently.

stances since a certain logical time), but increases space requirements.[3] To accommodate sets that do not fit on a single machine and/or to improve synchronization efficiency, the implementation of Deftu and Griebsch supports object sharding, which we do not consider.

Our original optimized add-wins set implementation [21] (not presented in this thesis) combined both state- and op-based replication, allowing updates to be delivered either as individual operations, under causal delivery, or as complete state, under any delivery condition. Mukund et al. [72] discuss a follow-up on this idea, and extends it to support a new, weaker delivery network specification for op-based messages, k-causal delivery. Under k-causal delivery, a message can be delivered if the receiver has not missed more than $k-1$ causal dependencies from the sender. Mukund et al. generalize our solution from causal delivery to k-causal delivery, by *(i)* replacing version vectors with an interval-based representation [24]; and by *(ii)* arbitrarily extending specification $\mathcal{F}_{\texttt{AWSet}}$ to handle visibility that is not transitive. The efficiency of their implementation depends on k: the higher k, the higher metadata overhead. With $k = 1$, it reduces to our optimization. The practical significance of the optimization is unknown, since we are unaware of any k-causal delivery protocol.

Similarly to deltas, these designs are also orthogonal to our work, as they concern message size.

5.3.2 Optimizations Based on Topology Restrictions and Delayed Visibility

Our results assume a full network communication topology, where every pair of replicas communicates directly. An important group of optimizations make use of a restricted topology to reduce the metadata overhead; in particular, of variants of the client-server topology.

For example, the metadata overhead of the state-based counter is proportional to the number of vector entries, i.e., the number of replicas. This can be too much in a large or dynamic system. To address this issue, Almeida and Baquero [5] propose counters optimized for a *tiered topology*, where replicas in the lowest tier cannot communicate directly with one another, but only via small set of replicas in a higher tier. In the simplest case, i.e., two tiers, it reduces to client and server replicas. The goal is to use a smaller number of vector entries in the average case, in the order of the number of servers, by eventually representing all client increments in these limited server entries.

The state of client and server replica is different in Almeida's counter. They share a vector of entries for server replicas. Additionally, a client replica maintains a single entry for his own increments that are not included in server entries so far. The increments from this entry are *handed off* to the server tier replicas by means of a fault-tolerant non-blocking two-phase protocol. When a hand-off instance succeeds, the increments are exposed through the server tier to all replicas, without "polluting" their vectors with the client entry, and do not leave any trace on

[3]The implementation of Almeida et al. [7], described earlier, demonstrates that storing indidividual instances is not necessary to generate deltas.

server replicas. However, until a hand-off instance finishes, i.e., between the first and the second round of the protocol, the server replica needs to maintain a dedicated slot for the protocol instance, to tolerate potential message duplicates from the client. In the worst-case, if the client fails after the first round, an unused slot remains opened in the server tier forever.

For such a protocol, it is useful to define metadata overhead separately for client and server replicas. The worst case for a server replica is identical to the normal state-based counter, since every client may fail after the first round of the protocol. Under favorable conditions, the common case is better. The worst case for a client replica is significantly improved, since it requires only one vector entry per server replica. Thus, servers can *amortize* the metadata cost for clients.

Our general lower bound proofs do not cover this protocol, due to different visibility witness. This protocol could be characterized by a visibility witness $\mathcal{V} \neq \mathcal{V}^{\mathsf{state}}$, which exposes updates only after an extra round of messages. We conjecture that the lower bound for server replicas for this category of protocols is the same as in the state-based implementation; the proof for such flexibly delayed visibility witnesses is future work.

Topology restrictions are also successfully applied to variants of op-based implementations. This is the case for our SwiftCloud system from Part III, and for a protocol of Burckhardt et al. [35], which assume multiple servers or a single server, respectively. The protocol of Burckhardt additionally allows to apply semantics-specific *reduction* on the log of stored or transferred operations.[4] In a previous work [103], we also show a protocol for tree-based list implementation, where the tree can be compacted (rebalanced) at a small set of servers, independently from concurrent operations at client replicas, to optimize message size.

5.3.3 Replicated File Systems

A number of replicated file systems, such as Ficus [80], Coda [59], or DFS-R [24], rely on eventually consistent replication. Some of them use the state-based replication model, and share similarities with state-based RDT implementations.

Bjørner [24] describes DFS-R, a replicated file system that ships with Microsoft products. DFS-R can be viewed as a complex RDT object. Its state is a map from unique file identifier to file record (including file name, data etc.), and a version vector. When a file is created, it is assigned a unique id. A file can be also updated, deleted, and moved. DFS-R uses the LWW approach to resolve concurrent updates to the same file identifier, and more complex heuristics at the global file system level. When a file is deleted and created again, it uses a fresh identifier.

DFS-R implementation shares commonalities with the two add-wins set implementations. In fact, we can see part of an DFS-R as an implementation of a *token set*, which supports two updates: creation of a fresh token (file identifier) and removal of an existing token. Token set is almost equivalent to the set of element instances in the add-wins implementations.

[4]Somewhat similar optimization techniques have been exploited in the context of log-based protocols without topology restrictions: Conway et al. [41] propose automated synthesis of optimized general purpose log-based distributed applications, whereas Baquero et al. [17] propose manually optimized op-based RDT implementations.

In this context, it is interesting how DFS-R treats remove operations and tombstones. Bjørner describes two approaches to removal: a standard tombstone-based and a tombstone-free. The latter relies on the knowledge in a version vector, maintained along a map of file identifiers to file records (tokens). By using timestamps as tokens, version vector can serve as a summary of all observed tokens. Our first optimization to the optimized add-wins set in Section 3.2.1.1 can be viewed as an adaptation of the tombstone-free implementation of a token set in Bjørner's file system, to build an implementation of a high level set RDT.

5.4 Lower Bound Proofs in Distributed Computing

The distributed computing community has studied the complexity of implementing different distributed or concurrent abstractions, and has developed techniques to demonstrate asymptotic lower bounds on their complexity. It is natural to compare our lower bound proofs to these works.

According to the survey of Fich and Ruppert [49], most of these works consider either system models or metrics significantly different from ours. Results for message passing models are usually concerned with time and message complexity required to solve a certain problem, or size of messages involved. Results for shared memory models typically evaluate the number of registers or other objects required to solve a problem. Nevertheless, the core counting argument of the lower bound proofs is often similar: the goal is to demonstrate that there is a (large) set of states that are distinguishable, because if they are not, the correctness would be affected. The structure of our experiment family, in particular the states during culprit reads in Definition 4.1, takes a similar approach.

The most relevant is the work of Charron-Bost [39], who proves that the size of vector clocks [71] is optimal to represent the happens-before relation of a computation. Happens-before relation is a counterpart of the visibility relation at the level of concrete executions. Specifications of multi-value register (`MVReg`), sets (`AWSet` and `RWSet`), and other data types, rely crucially on visibility. However, Charron-Bost's result does not translate into a lower bound on their implementation complexity. A specification may not require complete knowledge about the relation for all pair of events, and an implementation may not even need to store information about all events. It is often sufficient to store a fragment of this knowledge, using a semantics-optimized representation. For example, the optimized add-wins set implementation (Algorithm 3.2) does not require to store a complete vector clock about every `rem` event it observed. Instead, it stores only a version vector that summarizes all known operations, and a vector for each visible element (corresponding to `add` event).

Part III

Causally-Consistent Object Database for Client-Side Applications

Chapter 6

Problem Overview

What's normal anyways?

Forrest Gump by Robert Zemeckis

Contents

In this chapter we introduce a client-side replication problem that we study and that we address in the remaining part of the thesis. The problem concerns client-side applications, such as in-browser and mobile *apps*, which are poorly supported by the current technology for sharing mutable data over the wide-area. App developers resort to implementing their own ad-hoc application-level cache and buffer [60, 94], in order to avoid slow, costly and sometimes unavailable round-trips to a data center, but they are not in the position to solve system issues such as fault tolerance, or consistency/session guarantees [34, 96]. Recent client-side systems ensure only some of the desired properties, i.e., either make only limited consistency guarantees (at a granularity of a single object or a small database only), do not tolerate failures, and/or do not scale to large numbers of client devices [18, 30, 37, 40, 69]. Standard algorithms for geo-replication [8, 12, 46, 66, 67] were not designed to support high numbers of client replicas outside of the server-side infrastructure.

Our thesis is that it is possible to build a scalable fault-tolerant system that takes the responsibility of providing client-side applications a local database access that is available, (causally) consistent, and convergent, i.e., that addresses the client-side replication problem.

Before we demonstrate such a system, we define the problem more precisely and highlight the challenges. In Section 6.1, we outline the system model and basic requirements. In Section 6.2, we motivate and define causal consistency model for RDT objects. In Section 6.3, we summarize the requirements with the client API. In Section 6.4, we show why the problem is nontrivial; first, by discussing prior metadata designs, and why it is difficult to apply them in the context of client-side replication; second, why partial client-side replication complicates the problem.

Hereafter, we *depart from the rigorous formalism* of Part II, since our focus is not anymore on the RDT model and RDT implementation metadata itself, but on the update delivery layer. We focus on log-exchange protocols supporting both op-based RDTs, with strong message delivery requirements, and consistency across objects. We believe that a less rigorous description is adequate in the context of research and system engineering challenges of log-exchange protocols.

6.1 System Model and Basic Requirements

We consider support for a variety of client-side applications sharing a database of RDT objects that the client can read and update. We aim to scale to thousands of clients, spanning the whole internet, and to a database of arbitrary size.

Figure 6.1 illustrates our system model. A cloud infrastructure connects a small set (say, tens) of geo-replicated data centers, and a large set (thousands) of clients. A DC has abundant computational, storage and network resources. Similarly to Sovran et al. [95], we abstract a DC as a powerful sequential process that hosts a **full replica** of the database.[1] DCs communicate in a

[1]We refer to prior work for the somewhat orthogonal issues of ensuring parallelism and fault-tolerance within a DC, using sharding and replication, respectively [8, 46, 66, 67]. We discuss them in Chapter 9.

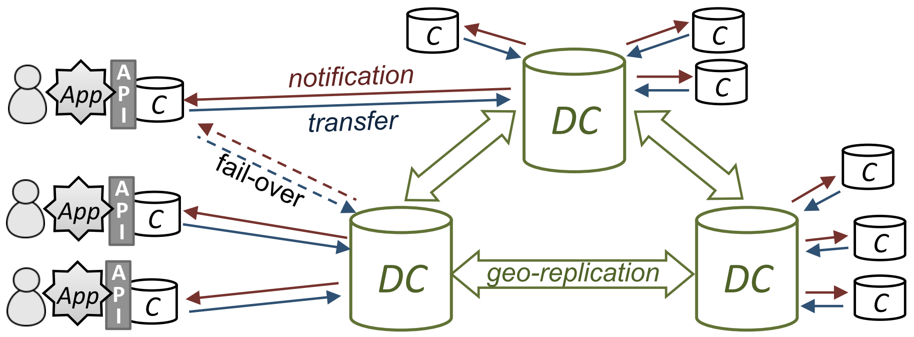


Figure 6.1: System components (Application processes, Clients, Data Centers), and their interfaces. Arrows indicate protocols over wide-area (high latency, unreliable) network connections.

peer-to-peer way. A DC may fail (due to disaster, power outage, WAN partition or misconfiguration, etc. [11, 57]) and recover with its persistent memory intact.

Clients do not communicate directly, but only via DCs.[2] Normally, a client connects to a single DC; in case of failure or roaming, to zero or more. A client may fail and recover (e.g., disconnection during a flight) or permanently (e.g., destroyed phone) without prior warning. We consider only non-byzantine failures.

Client-side apps often require high **availability** and **responsiveness** for good user experience, i.e., to be able to read and update data quickly and at all times. This can be achieved by replicating RDT objects locally, and by synchronizing updates in the background. However, a client has limited resources; therefore, it hosts a **cache** that contains only the small subset of the database of current interest to the local app. It should not have to receive messages relative to objects that it does not currently replicate [18, 88]. For similar reasons, control messages and piggy-backed metadata should have small and bounded size.

Since a client replica is only *partial*, there cannot be a guarantee of complete availability of all operations. The best we can expect is **partial availability**, whereby an operation returns without remote communication if the requested object is cached; and after retrieving the object from a remote node (DC) otherwise. If the object is not there and the network is down, the operation may be unavailable, i.e., it either blocks or returns an error.

6.2 Consistency with Convergence

Application programmers wish to observe a single, consistent view of the database. With availability and convergence requirements, consistency options are limited [50, 68]. In Section 2.1, we outlined why strongly consistent (i.e., linearizable) objects are unachievable in an available system, which is why we resort to RDTs. Similarly, strong consistency across objects, e.g., serializability [20], is unachievable too. Nevertheless, some inter-object guarantees are achievable.

[2]Although auxiliary client-to-client communication is viable in some environments, we do not consider it here.

6.2.1 Causal Consistency

We consider support for the strongest known available and convergent consistency model: causal consistency with RDT objects [4, 68].

Definition 6.1 (Causal Order and Consistency)**.** Let an execution H be a set of sequences (one per application process session) of object operation invocations with their return values. Operations e and f are *potentially causally-related* in H, noted $e \to f$ and called **causal order**, if:
A. An application process session invoked f after it invoked e; or
B. Read f observed update e on the same object; or
C. There exists an operation $g \in H$ such that $e \to g \to f$.
An execution H is **causally consistent** if every read operation in H observes all the updates that causally precede the read, applied in some linear extension of causal order.

Informally, under causal consistency, every process observes a monotonically non-decreasing set of updates, including its own updates, in an order that respects the causality between operations.[3] The following well-known example illustrates it [66]. In a social network, Bob sets permissions to disallow his boss Alice from viewing his photos. Some time later, Bob posts a questionable photo of himself. Without causal consistency, Alice may view the bad photo, delivered before the new permissions. Under causal consistency, the change of permission is guaranteed to be delivered before the post, and Alice cannot view the photo.

More generally, if an application process reads object x, and later reads object y, and if the state of x causally-depends on some update u to y, then the state of y that it reads will include update u. When the application requests y, we say there is a **causal gap** if the local replica has not yet delivered u. The system must detect such a gap, and wait until u is delivered before returning y, or avoid a gap in the first place. Otherwise, reads with a causal gap expose both application programmers and users to ordering anomalies [66, 67].

We consider a transactional variant of causal consistency to better support multi-object operations. A transaction is a sequence of operations issued by a single client session. All the reads of a **causal transaction** come from a same database snapshot, and either all its updates are visible as a group, or none is [14, 66, 67]. Causal transactions facilitate, for instance, maintenance of secondary indices and symmetric relations [14], and object composition [52].

6.2.2 Convergence with Replicated Data Types

Recall from Section 2.3 that **convergence** consists of safety and liveness components:

1. **Eventual visibility / at-least-once delivery** (liveness): an update that is delivered (i.e., is visible by the app) at some node, is eventually delivered to all (interested) connected nodes; for timely convergence, we require that it is delivered after a finite number of message exchanges;

[3]This subsumes the well-known session guarantees, such as *read-your-writes* or *monotonic reads* [34, 96].

2. **Confluence** (safety): two nodes that delivered the same set of updates read the same value.

Causal consistency alone is not enough to guarantee confluence, as two replicas might deliver concurrent updates in different orders. Therefore, we rely on RDT objects for order-insensitive confluence. Specifically, we consider the **op-based** implementation category. Recall that op-based objects can use a common update delivery protocol to enforce consistency, and to share metadata, across objects. The metadata cost lies primarily in the delivery protocol. The delivery protocol can be topology-optimized and can address the problem *once and for all*. In contrast, state-based objects do not easily support cross-object consistency, and require type-specific optimizations.

Implementations of op-based RDTs require adequate support from the system, especially in terms of network layer guarantees (cf. discussion in Section 2.4.3). Many existing causal consistency protocols offer only LWW register support [8, 18, 46, 66, 67, 69], which is simpler than general purpose support of higher-level types such as sets or counters. For instance, as high-level RDT updates are often not idempotent (consider for instance `inc` in counter), safety also demands **at-most-once delivery**. Op-based implementations, especially optimized ones, often require **causal delivery** of object updates. Since an RDT object's value may be defined not just by the last update (as in the case of LWW register), but also depend on earlier updates, causal dependencies may span several different object updates, and need to be encoded and respected.

6.3 Application Programming Interface

To summarize, our client API resembles both modern object stores, such as Riak 2.0 or Redis, and prototype causal transactional stores, such as COPS or Eiger [3, 66, 67, 84]:[4]

• begin_transaction()	Opens a causal transaction (at most one at a time).
• read(object_id) : v	Performs `read` on object object_id with return value v.
• update(object_id,`method`)	Performs an update operation `method` on object_id.
• commit_transaction()	Terminates the open transaction.

Object id embeds an object type. A read can be configured to create a non-existent object. The implementation additionally supports fine control over caching, and a non-blocking API.

6.4 Challenge

Although each of discussed requirements may seem familiar or simple in isolation, the combination with scalability to high numbers of nodes and database size is a novel challenge.

6.4.1 Metadata Design

System metadata serves to identify individual updates, and sets of updates, to ensure correct delivery. Metadata is piggy-backed on update messages, increasing the cost of communication.

[4]Unlike COPS or Eiger, we consider interactive transactions, i.e., accessed objects do not need to be predefined.

One common metadata design assigns each update a timestamp as soon as it is generated on some originating node. This is flexible and supports tracking causality at a fine grain. However, the metadata data structures tend to grow "fat." For instance, dependency lists [66] grow with the number of updates (cf. Du et al. [46], Lloyd et al. [67, Section 3.3]), whereas version vectors [18, 69] grow with the number of clients (indeed, our experiments show that their size becomes unreasonable). We call this the **Client-Assigned, Safe but Fat** approach.

An alternative delegates timestamping to a small number of DC servers [8, 46, 67]. This enables the use of small vectors, at the cost of losing some parallelism. However, this is not fault tolerant if the client does not reside in a DC failure domain. For instance, it may violate at-most-once delivery. Consider a client transmitting update u to be timestamped by DC1. If it does not receive an acknowledgement, it retries, say with DC2 (fail-over). This may result in u receiving two distinct timestamps, and being delivered twice. Duplicate delivery violates safety for many op-based RDTs, or otherwise increases their space and implementation complexity [5, 34, 67]. We call this the **Server-Assigned, Lean but Unsafe** approach.

Clearly, neither "fat" nor "unsafe" is satisfactory.

6.4.2 Causal Consistency with Partial Replication is Hard

Since a partial replica receives only a subset of the updates, and hence of metadata, it could miss some causal dependencies [18]. Consider the following example: Alice posts a photo on her wall (update e). Bob sees the photo and mentions it in a message to Charles (update f), who in turn mentions it to David (update g). When David looks at Alice's wall, he expects to observe update e and view the photo. However, if David's machine does not cache Charles' inbox, it cannot observe the causal chain $e \to f \to g$ and might incorrectly deliver g without e. Metadata design should protect from such causal gaps, caused by transitive dependency over absent objects.

Failures complicate the picture even more. Suppose David sees Alice's photo, and posts a comment to Alice's wall (update h). Now a failure occurs, and David's machine fails over to a new DC. Unfortunately, the new DC has not yet received Bob's update f, on which comment h causally depends. Therefore, it cannot deliver the comment, i.e., fulfill convergence, without violating causal consistency. David cannot read new objects from the DC for the same reason.[5]

Finally, a DC logs an individual update for only a limited amount of time, but clients may be unavailable for unlimited periods. Suppose that David's comment h is accepted by the DC, but David's machine disconnects before receiving the acknowledgement. Much later, after h has been executed and purged away, David's machine comes back, only to retry h. This could violate at-most-once delivery; some previous systems avoid this with fat version vectors [18, 69] or with log compaction protocol that relies on client availability [61].

[5]Note that David can still perform update. However, the system as a whole does not converge.

Chapter 7

The SwiftCloud Approach

I don't have to wait to realize the good old days.

Ziggy Marley

Contents

In this chapter, we present the design, algorithms, and evaluation of SwiftCloud, a distributed RDT object database that addresses the challenges of client-side replication. It efficiently ensures causally consistent, available, and convergent access to high number of client replicas, tolerating failures. To achieve this, SwiftCloud uses a flexible client-server protocol. The client *writes fast* into the local cache, and *reads in the past* (also fast) data that is consistent, but occasionally stale.

In Section 7.1, we present the design of SwiftCloud, based on the *cloud-backed support for partial replicas*. To avoid the complexity of consistent partial replication at the client side and at the scale of client-side devices, we leverage the DC's full replicas to provide a consistent view of the database to the client. The client merges this view with his own updates. To tolerate DC failures, we explore a trade-off between liveness and data freshness, and serve clients a slightly delayed, but consistent and sufficiently replicated version.

In Section 7.2, we show how to realize the abstract design with *concrete protocols using a form of lean and safe metadata*. Thanks to funnelling communication through DCs and to "reading in the past," we can use a metadata design that decouples two aspects: *causality tracking* using small vectors assigned in the background by DCs, and *unique identification* of updates to protect from duplicates, using client-assigned scalar timestamps. This ensures that the size of metadata is small and bounded, and allows DC to prune its log independently of clients' availability.

7.1 Design

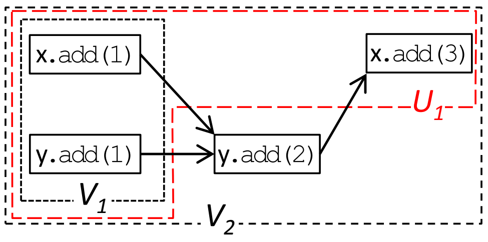
We first describe an abstract design that addresses the client-side replication challenges, starting from the failure-free case, and then, how we support DC failure. Our design demonstrates how to apply principles of causally consistent algorithms for full replication systems to build a cloud-based support for partial client replicas.

7.1.1 Causal Consistency at Full Data Center Replicas

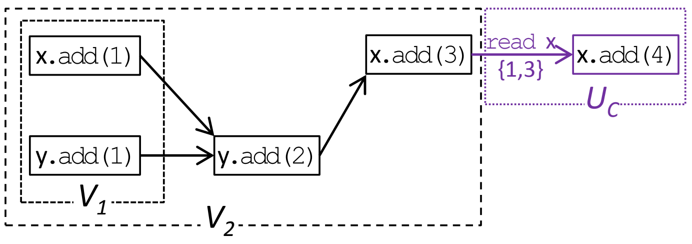
Ensuring causal consistency at full geo-replicated DC replicas is a well-known problem [4, 46, 66, 67]. Our design is primarily based on a log-based approach that stores updates in a log and transfers them incrementally, combined with checkpointing optimizations [18, 79], where the system occasionally stores and transmits the state of an object, called **checkpoint**, in place of the complete log. We focus on the log-based angle, and discuss checkpoints only where relevant.

A **database version** is made of any subset of updates, noted U, ordered by causality. A version maps object identifiers to object state, by applying the relevant subsequence of the updates log. The value of an object in a version is exposed via the read API. Our log-based implementation stores the state in log chunks, one per object, ordered in a linear extension of their causal order.

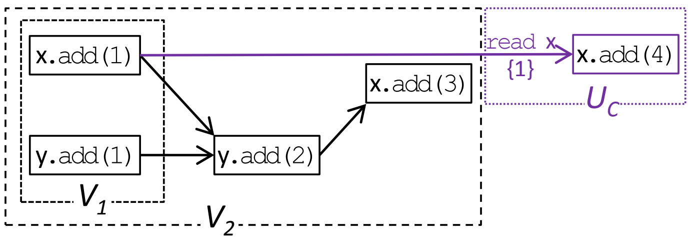
We say that a version U has a **causal gap**, or is **inconsistent** if it is not causally-closed, i.e., if $\exists u, u' : u \rightarrow u' \land u \notin U \land u' \in U$. As we illustrate shortly, reading from an inconsistent version



(A) Initial configuration.



(B) Continuation from 7.1A to risky configuration.



(C) Read-in-the-past: continuation from 7.1A to conservative configuration.

Figure 7.1: Example evolution of configurations for two DCs, and a client. x and y are sets; box = update; arrow = causal dependence (an optional text indicates the source of dependency); dashed box = named database version/state.

should be avoided, because, otherwise, subsequent accesses might violate causality. On the other hand, waiting for the gap to be filled would increase latency and decrease availability. To side-step this conundrum, we adopt the approach of “reading in the past” [4, 66]. Thus, a DC exposes a gapless but possibly delayed state, noted V.

To illustrate, consider the example of Figure 7.1A. Objects x and y are of type set. DC_1 is in state U_1 that includes version $V_1 \subseteq U_1$, and DC_2 in a later state V_2. Versions V_1 with value $[\mathsf{x} \mapsto \{1\}, \mathsf{y} \mapsto \{1\}]$ and V_2 with value $[\mathsf{x} \mapsto \{1,3\}, \mathsf{y} \mapsto \{1,2\}]$ are both gapless. However, version U_1, with value $[\mathsf{x} \mapsto \{1,3\}, \mathsf{y} \mapsto \{1\}]$ has a gap, missing update y.add(2). When a client requests to read x at DC_1 in state U_1, the DC could return the most recent version, $\mathsf{x} : \{1,3\}$. However, if the application later requests y, to return a safe value of y requires to wait for the missing update from DC_2. By “reading in the past” instead, the same replica exposes the older but gapless version V_1, reading $\mathsf{x} : \{1\}$. Then, the second read will be satisfied immediately with $\mathsf{y} : \{1\}$. Once the missing update is received from DC_2, DC_1 may advance from version V_1 to V_2.

A gapless algorithm maintains a *causally-consistent, monotonically non-decreasing progression* of replica states [4]. Given an update u, let us note $u.deps$ its set of causal predecessors, called its **dependency set**. If a full replica, in some consistent state V, receives u, and its dependencies are satisfied, i.e., $u.deps \subseteq V$, then it applies u. The new state is $V' = V \oplus \{u\}$, where we note by $\oplus$ a **log merge operator** that filters out duplicates caused by failures, further discussed in Section 7.2.1. State V' is consistent, and monotonicity is respected, since $V \subseteq V'$.

If the dependencies are not met, the replica buffers u until the causal gap is filled.

7.1.2 Causal Consistency at Partial Client Replicas

As a client replica contains only part of the database and its metadata, this complicates consistency [18]. E.g., encoding and respecting causal dependencies is more difficult. To avoid the complexity of the protocol and to minimize metadata size, we leverage the DC's full replicas to manage large part of gapless versions for the clients.

Given some **interest set** of objects the client is interested in, its initial state consists of the projection of a DC state onto the interest set. This is a causally-consistent state, as shown in the previous section. Client state can change either because of an update generated by the client itself, called an **internal update**, or because of one received from a DC, called **external**. An internal update obviously maintains causal consistency. If an external update arrives, without gaps, from the same DC as the previous one, it also maintains causal consistency.

More formally, consider some recent DC state, which we will call the **base version** of the client, noted V_{DC}. The interest set of client C is noted $O \subseteq \mathsf{x}, \mathsf{y}, \ldots$. The client state, noted V_C, is restricted to these objects. It consists of two parts. One is the projection of base version V_{DC} onto its interest set, noted $V_{DC}|_O$. The other is the log of internal updates, noted U_C. The client state is their merge $V_C = V_{DC}|_O \oplus U_C|_O$. On cache miss, the client adds the missing object to its interest set, and fetches the object from base version V_{DC}, thereby extending the projection.

Safety. Base version V_{DC} is a monotonically non-decreasing causal version (it might be slightly behind the actual current state of the DC due to propagation delays). By induction, internal updates can causally depend only on internal updates, or on updates taken from the base version. Therefore, a hypothetical full version $V_{DC} \oplus U_C$ would be causally consistent. Its projection is equivalent to the client state: $(V_{DC} \oplus U_C)|_O = V_{DC}|_O \oplus U_C|_O = V_C$.

Liveness. This approach ensures partial availability. If a version is in the cache, it is guaranteed causally consistent, although possibly slightly stale. If it misses in the cache, the DC returns a consistent version immediately. Furthermore, the client app can *write fast*, because it does not wait to commit updates, but the client replica transfers them to its DC in the background.

Convergence is ensured, because *(i)* the client's base version is maintained up to date by the DC, in the background, and internal log is propagated to the DC; and *(ii)* RDTs are confluent.

7.1.3 Failing Over: The Issue with Transitive Causal Dependency

The approach described so far assumes that a client connects to a single DC. In fact, a client can switch to a new DC at any time, in particular in response to a failure. Although each DC's state is consistent, an update that is delivered to one is not necessarily delivered in the other (because geo-replication is asynchronous, to ensure availability and performance at the DC level [15]), which may create a causal gap in the client.

To illustrate the problem, return to the example of Figure 7.1A. Consider two DCs: DC_1 is in (consistent) state V_1, and DC_2 in (consistent) state V_2; DC_1 does not include two recent updates of V_2. Client C, connected to DC_2, replicates object x only; its state is $V_2|_{\{\mathsf{x}\}}$. Suppose that the client reads the set $\mathsf{x}:\{1,3\}$, and performs update $u:\mathsf{x.add(4)}$, transitioning to the configuration shown in Figure 7.1B.

In this configuration, if the client now fails over to DC_1, and the two DCs cannot communicate, the system is not live:

(1) *Reads are not available*: DC_1 cannot satisfy a request for y, since the version read by the client is newer than the DC_1 version, $V_2 \not\subseteq V_1$.

(2) *Updates cannot be delivered (divergence)*: DC_1 cannot deliver u, due to a missing dependency: $u.deps \not\subseteq V_1$.

Therefore, DC_1 must reject the client to avoid creating the gap in state $V_1 \oplus U_C$.[1]

7.1.3.1 Conservative Read: Possibly Stale, But Safe

To avoid such gaps that cannot be satisfied, we use an approach similar to Mahajan et al. [69], and depend on updates that are likely to be present in the fail-over DC, called **K-stable** updates.

A version V is K-stable if every one of its updates is replicated in at least K DCs, i.e., $|\{i \in \mathcal{DC} \mid V \subseteq V_i\}| \geq K$, where $K \geq 1$ is a threshold configured w.r.t. expected failure model, and $\mathcal{DC}$ is a set of all data centers. To this effect, in our system every DC maintains a consistent **K-stable version** $V_i^K \subseteq V_i$, which contains the updates for which DC_i has received acknowledgements from at least $K-1$ distinct other DCs.

A client's base version must be K-stable, i.e., $V_C = V_i^K|_O \oplus U_C|_O$, to support failover. In this way, the client depends, either on external updates that are likely to be found in any DC (V_i^K), or internal ones, which the client can always transfer to the new DC (U_C).

To illustrate, let us return to Figure 7.1A, and consider the conservative progression to configuration in Figure 7.1C, assuming $K = 2$. The client's read of x returns the 2-stable version $\{1\}$, avoiding the dangerous transitive dependency via an update on y. If DC_2 is unavailable, the client can fail over to DC_1, reading y and propagating its update remain both live.

Correctness with K-stability. By the same arguments as in Section 7.1.2, a DC version V_i^K is causally consistent and monotonically non-decreasing, and hence the client's version as well.

[1]The DC may accept to *store* client's updates to improve durability, but it cannot *deliver* them or offer notifications.

Note that a client observes his internal updates immediately, even if not K-stable. Our approach is *flexible* w.r.t. which consistent version the DC offers to the client as a base version.

Parameter K can be adjusted dynamically. Decreasing it has immediate effect without impacting correctness. Increasing K has effect only for future updates, to preserve montonicity.

7.1.3.2 Discussion

The source of the problem with the basic algorithm from Section 7.1.2 is an indirect causal dependency on an update that the two replicas do not both know about (y.add(2) in our example). As this is an inherent issue, we conjecture a general impossibility result, stating that genuine partial replication, causal consistency, partial availability and timely at-least-once delivery (convergence) are incompatible. Accordingly, some requirements must be relaxed.

Note that in many previous systems, this impossibility translates to a trade-off between consistency and availability on the one hand, and response time and throughput on the other [43, 66, 95]. By "reading in the past," we displace this to a trade-off between freshness and availability, controlled by adjusting K. A higher K increases availability, but updates take longer to be delivered;[2] in the limit, $K = N$ ensures complete availability, but no client can deliver a new update when some DC is unavailable. A lower K improves freshness, but increases the probability that a client will not be able to fail over, and that it will block until its original DC recovers. In the limit, $K = 1$ is identical to the basic protocol from Section 7.1.2, and is similar to previous blocking session-guarantee protocols [96].

$K = 2$ is a good compromise for deployments with three or more DCs that covers common scenarios of a DC failure or disconnection [43, 57]. Our evaluation with $K = 2$ shows that it incurs a negligible staleness.

Network partitions. Client failover between DCs is safe and generally live, except in the unlikely case when the original set of K DCs were partitioned away from both other DCs and the client, shortly after they delivered a version to the client. In this case, the client blocks. To side-step this unavoidable possibility, we provide an unsafe API to read inconsistent data.

When a set of fewer than K DCs is partitioned from other DCs, the clients that connect to them are available and safe, but do not deliver their mutual updates until the partition heals. To improve liveness in this scenario, SwiftCloud supports two heuristics: *(i)* a partitioned DC announces its "isolated" status, automatically recommending clients to use another DC, and *(ii)* clients who cannot reach another DC that would satisfy their dependencies can use the isolated DCs with K temporarily lowered, risking unavailability if another DC fails.

Precision vs. missing dependencies. The probability of a client blocked due to an unsatisfied transitive causal dependency depends on many factors, such as workload- and deployment-

[2] The increased number of concurrent updates that this causes is not a big problem, thanks to RDTs.

specific ones. Representation of dependencies also contributes. SwiftCloud uses coarse-grained representation of dependencies, at the granularity of the complete base version used by the client. This may cause a *spurious* missing dependency, when a DC rejects a client because it misses some update that is not an actual dependence. Finer-grained dependency representation, such as in causality graphs [66], or resorting to application-provided explicit dependencies under a weaker variant of causal consistency [12], avoid some spurious dependencies at the expense of fatter metadata. However, the missing dependency issue remains under any dependency precision. Thus, our approach is to fundamentally minimize the chances of *any* missing dependency, both genuine and spurious.

7.2 Implementation

We now describe a metadata and concrete protocols implementing the abstract design.

7.2.1 Timestamps, Vectors and Log Merge

The SwiftCloud approach requires metadata: *(1)* to *uniquely identify an update*; *(2)* to *encode its causal dependencies*; *(3)* to *identify* and *compare versions*; *(4)* and to *identify all the updates of a transaction*. We now describe a type of metadata, which fulfills the requirements and has a low cost. It combines the strengths of the two approaches outlined in Section 6.4.1, and is both *lean and safe*

Recall that a logical timestamp is a pair $(i,k) \in (\mathcal{DC} \cup \mathcal{C}) \times \mathbb{N}$, where i identifies the node that assigned the timestamp (either a DC or a client) and k is a sequence number. Similarly to the solution of Ladin et al. [61], our metadata assigned to some update u combines both: *(i)* a single **client-assigned timestamp** $u.t_{\mathcal{C}}$ that uniquely identifies the update, and *(ii)* a set of zero or more **DC-assigned timestamps** $u.T_{\mathcal{DC}}$. Before being delivered to a DC, the update has no DC timestamp; it has one thereafter; it may have more than one in case of delivery to multiple DCs (on failover). The updates in a transaction all have the same timestamp(s), to ensure all-or-nothing delivery [95]. Our approach provides the flexibility to refer to an update via any of its timestamps, which is handy during failover.

We represent a version or a dependency as a **version vector** [77], introduced in Section 2.2.3. Recall that a version vector is a partial map from node id to integer, e.g., $vv = [DC_1 \mapsto 1, DC_2 \mapsto 2]$, which we interpret as a set of timestamps. For example, when vv is used as a dependency for some update u, it means that u causally depends on a set of timestamps $\mathcal{TS}(vv) = \{(DC_1,1),(DC_2,1),(DC_2,2)\}$ ($\mathcal{TS}$ is defined in Equation 3.1). In SwiftCloud protocols, every vector has at most one client entry, and multiple DC entries; thus, its size is bounded by the number of DCs, limiting network overhead. In contrast to a dependence graph [66], a vector compactly represents transitive dependencies and can be evaluated locally by any node.

A **version decoding function** $\mathcal{V}$ of vector vv on a state U selects every update in state U that matches the vector (it is defined for states U that cover all timestamps of vv):

$$\mathcal{V}(vv, U) = \{u \in U \mid (u.T_{\mathcal{DC}} \cup \{u.t_{\mathcal{C}}\}) \cap \mathcal{TS}(vv) \neq \emptyset\}$$

For the purpose of the decoding function $\mathcal{V}$, a given update can be referred equivalently through any of its timestamps, which we leverage for failover. Moreover, $\mathcal{V}$ is stable with growing state U, which is handy to identify a version in a (large) state that undergoes concurrent log appends.

The log merge operator $U_1 \oplus U_2$, which eliminates duplicates, is defined using client timestamps. Two updates $u_1 \in U_1, u_2 \in U_2$ are identical if $u_1.t_{\mathcal{C}} = u_2.t_{\mathcal{C}}$. The merge operator merges their DC timestamps into $u \in U_1 \oplus U_2$, such that $u.T_{\mathcal{DC}} = u_1.T_{\mathcal{DC}} \cup u_2.T_{\mathcal{DC}}$.

7.2.2 Protocols

We are now in the position to describe the concrete protocols of SwiftCloud, by following the lifetime of an update, and with reference to the names in Figure 6.1.

7.2.2.1 State

A DC replica maintains its state U_{DC} in durable storage. The state respects causality and atomicity for each individual object, but due to internal concurrency, this may not be true across objects. Therefore, the DC also has a vector vv_{DC} that identifies a safe, monotonically non-decreasing causal version in the local state, which we note $V_{DC} = \mathcal{V}(vv_{DC}, U_{DC})$. Initially, U_{DC} is an empty log of updates, and vv_{DC} is zeroed.

A client replica stores the commit log of its own updates U_C, and the projection of the base version from the DC, restricted to its interest set O, $V_{DC}|_O$, as described previously in Section 7.1.2. It also stores a copy of vector vv_{DC} that describes the base version.

7.2.2.2 Client-Side Execution

When the application starts a transaction τ at client C, the client replica initializes it with an empty **buffer of updates** $\tau.U = \emptyset$ and a **snapshot vector** of the current base version $\tau.vvDeps = vv_{DC}$; the base version can be updated by the notification protocol concurrently with the transaction execution. A read in transaction τ is answered from the version identified by the snapshot vector, merged with recent internal updates, $\tau.V = \mathcal{V}(\tau.vvDeps, V_{DC}|_O) \oplus U_C|_O \oplus \tau.U$. If the requested object is not in the client's interest set, $x \notin O$, the clients extends its interest set, and returns the value once the DC updates the base version projection.

When the application issues internal update u, it is appended to the transaction buffer $\tau.U \leftarrow \tau.U \oplus \{u\}$, and included in any later read. To simplify the notation, and without loss of generality, we assume hereafter that a transaction performs at most one update.[3] The transaction

[3]This is easily extended to multiple updates, by assigning the same timestamp to all the updates of the same transaction, ensuring the all-or-nothing property of causal transactions [95].

commits *locally* at the client; it never fails. If the transaction made update $u \in \tau.U$, the client replica commits it locally as follows: *(1)* assign it client timestamp $u.t_{\mathcal{C}} = (C, k)$, where k counts the number of committed transactions at the client; *(2)* assign it a **dependency vector** initialized with the transaction snapshot vector $u.\mathit{vvDeps} = \tau.\mathit{vvDeps}$; *(3)* append it to the commit log of local updates on stable storage $U_C \leftarrow U_C \oplus \{u\}$. This terminates the transaction, which becomes irrevocable; the client can start a new one, which will observe the committed updates.

7.2.2.3 Transfer Protocol: Client to Data Center

The transfer protocol transmits committed updates from a client to its current DC, in the background. It repeatedly picks the first unacknowledged committed update u from the log. If any of u's internal dependencies has recently been assigned a DC timestamp, it merges this timestamp into the dependency vector, $u.\mathit{vvDeps}$. Then, the client sends a copy of u to its current DC. The client expects to receive an acknowledgement from the DC, containing the timestamp(s) T that the DC assigned to update u. If so, the client records the DC timestamp(s) in the original update record $u.T_{\mathcal{DC}} \leftarrow T$. T is a singleton set, unless failover was involved.

The client may now iterate with the next update in the log.

A transfer request may fail for three reasons:

A. **Timeout**: the DC is suspected unavailable; the client connects to another DC (failover) and repeats the protocol.

B. The DC reports a **missing internal dependency**, i.e., it has not received some update of the client, as a result of a previous failover. The client recovers by marking as unacknowledged all internal updates starting from the oldest missing dependency, and restarting the transfer protocol from that point.

C. The DC reports a **missing external dependency**; this is also an effect of failover. In this case, the client tries yet another DC. The approach from Section 7.1.3.1 avoids repeated failures.

Upon receiving update u, the DC verifies if its dependencies are satisfied, i.e., if $\mathcal{TS}(u.\mathit{vvDeps}) \subseteq \mathcal{TS}(vv_{DC})$. (If this check fails, it reports an error to the client, indicating either case (B) or (C)). If the DC has not received this update previously, as determined by the client-assigned timestamp, i.e., $\forall u' \in U_{DC} : u'.t_{\mathcal{C}} \neq u.t_{\mathcal{C}}$, the DC does the following: *(1)* assign it a DC timestamp $u.T_{\mathcal{DC}} \leftarrow \{(DC, vv_{DC}(DC)+1)\}$, *(2)* store it in its durable state $U_{DC} \oplus \{u\}$, *(3)* make the update visible in the DC version V_{DC}, by incorporating its timestamp(s) into vv_{DC}. This last step makes u available to the geo-replication and notification protocols, described hereafter. If the update has been received before, the DC only looks up its previously-assigned DC timestamps, $u.T_{\mathcal{DC}}$. In either case, the DC acknowledges the transfer to the client with the DC timestamp(s). Note that some of these steps can be parallelized between transfer requests received from different client replicas, e.g., using batching for timestamp assignment.

7.2.2.4 Geo-replication Protocol: Data Center to Data Center

The geo-replication protocol relies on a uniform reliable broadcast across DCs [36]. An update enters the geo-replication protocol when a DC accepts a fresh update during the transfer protocol. The accepting DC broadcasts it to all other DCs. The broadcast implementation stores an update in a replication log until every DC receives it. A DC that receives a broadcast message containing u does the following: *(1)* if the dependencies of u are not met, i.e., if $\mathcal{TS}(u.vvDeps) \not\subseteq \mathcal{TS}(vv_{DC})$, buffer it until they are; and *(2)* incorporate u into durable state $U_{DC} \oplus \{u\}$ (if u is not fresh, the duplicate-resilient log merge $\oplus$ safely unions all timestamps), and incorporate its timestamp(s) into the DC version vector vv_{DC}. This last step makes it available to the notification protocol. The K-stable version V^K_{DC}, and a corresponding vector vv^K_{DC}, are maintained similarly.

7.2.2.5 Notification Protocol: Data Center to Client

A DC maintains a best-effort **notification session**, over a FIFO channel, to each of its connected clients. The *soft state* of a session at a DC includes a copy of the client's interest set O and the last known base version vector used by the client, vv'_{DC}. Both of them can be recovered from the client when necessary, and the client provides both to initiate a session. The DC accepts a new session only if its own state is consistent with the base version of the client, i.e., if $\mathcal{TS}(vv'_{DC}) \subseteq \mathcal{TS}(vv_{DC})$. Otherwise, the DC would cause a causal gap with the client's state; in this case, the client is redirected to another DC (the solution from Section 7.1.3.1 avoids repeated rejections).

The DC sends over each channel a causal stream of update notifications.[4] Notifications are batched according to either time or to rate [18]. A notification packet consists of a new base version vector vv_{DC}, and a log of all the updates U_δ to the objects of the interest set, between the client's previous base vector vv'_{DC} and the new one. Formally:

$$U_\delta = \{u \in U_{DC}|_O \mid u.T_{\mathcal{DC}} \cap (\mathcal{TS}(vv_{DC}) \setminus \mathcal{TS}(vv'_{DC})) \neq \emptyset\}. \tag{7.1}$$

The client applies the newly-received updates to its local state, described by the old base version, $V_{DC}|_O \leftarrow V_{DC}|_O \oplus U_\delta$, and assumes the new vector vv_{DC}. If any of received updates is a duplicate w.r.t. to the old version or to a local update, the log merge operator handles it safely. Note that transaction atomicity is preserved by Equation 7.1, since all updates of a transaction share a common timestamp.

When the client detects a broken channel, it reinitiates the session, possibly on a new DC.

The interest set can change dynamically. When an object is evicted from the cache, the notifications are lazily unsubscribed to save resources. When it is extended with object x, the DC responds with the current version of x, which includes all updates to x up to the base version vector. To avoid races, a notification includes a hash of the interest set, which the client checks.

[4]Alternatively, the client can ask for invalidations instead, trading responsiveness for lower bandwidth utilization and higher DC throughput [18, 69].

7.2.3 Object Checkpoints and Log Pruning

The log-based representation of database state contributes to substantial storage and, to smaller extent, network costs. This is a reminiscent of the similar problem at the object level of state-based RDT implementations from Part II, which manifests here in the log-exchange protocol for op-based implementations.

To avoid unbounded growth of the log, a background **pruning protocol** replaces the prefix of the log by a *checkpoint*. In the common case, a checkpoint is more compact than the corresponding log of updates. For instance, a log containing one thousand increments to an op-based counter object from Algorithm 2.2, including system timestamps, can be replaced by a checkpoint containing just the number 1000, and system metadata, a version vector.

7.2.3.1 Log Pruning in the Data Center

The log at a DC provides *(i)* unique timestamp identification of each update, which serves to filter out duplicates by $\oplus$ operator, to ensure reliable causal delivery, as explained earlier, and *(ii)* the capability to compute different versions, for application processes reading at different causal times. Conversely, for the protocol described so far, an update u is *expendable* once all of its duplicates have been filtered out, and once u has been delivered to all interested application processes. However, evaluating expendability precisely would require access to the client replica.

In practice, we need to prune *aggressively*, but still avoid the above issues, as we explain next.

In order to reduce the risk of pruning a version not yet delivered to an interested application, we prune only a **delayed version** vv_{DC}^{Δ}, where Δ is a real-time delay [66, 67]. If this heuristic fails, the consequences are not fatal: an ongoing client transaction may need to restart and repeat prior reads or return inconsistent data if desired, but the committed updates are never aborted.

To avoid duplicates, we extend our DC local metadata as follows. DC_i maintains an **at-most-once guard** G_i, which records the sequence number of each client's last pruned update $G_i : \mathcal{C} \to \mathbb{N}$. Whenever the DC receives a transfer request or a geo-replication message for update u with timestamp $(C,k) = u.t_{\mathcal{C}}$ and cannot find it in its log, it checks the at-most-once guard entry $G_i(C)$ whether u is contained in the checkpoint. If the update was already pruned away ($G_i(C) \geq k$), it is recognized as a pruned duplicate. In this case, the DC neither assigns the update a new timestamp (as it normally does in geo-replication protocol), nor stores it in the durable statate, nor delegates the actual update to the notification protocol (as in transfer and geo-replication protocols), but it updates its version vector to include all existing timestamps of the update; in transfer reply, the DC overapproximates u's timestamps by vector vv_{DC}, since the information about the exact set of update's DC timestamps is discarded.

The notification protocol also uses checkpoints. On a client cache miss, the DC does not send updates log for an object, but the object state that consists of the most recent checkpoint of the object and the client's guard entry, so that the client can merge it with his updates safely.

Note that a guard is *local* to and *shared* by all objects at a DC. It is *never* fully transmitted.

7.2.3.2 Pruning the Client's Log

Managing the log at a client is comparatively simpler. A client logs his own updates U_C, which may include updates to object that is currently out of his interest set — this enables the client to read its own updates, and to propagate them lazily to a DC when connected and convenient. An update u can be discarded as soon as it appears in K-stable base version vector vv_{DC}^{K}, i.e., when the client becomes dependent on the presence of u at a DC.

Chapter 8

Experimental Evaluation

If you torture the data enough, nature will always confess.

Ronald H. Costa

Contents

	YCSB [42]	**SocialApp** à la WaltSocial [95]
Type of objects	LWW map	set, counter, LWW register
Object payload	10 × 100 bytes	variable
Read transactions	read fields (load A: 50% / load B: 95%)	read wall[†] (80%) see friends (8%)
Update transactions	update field (load A: 50% / load B: 5%)	message (5%) post status (5%) add friend (2%)
Objects / transactions	1 (non-transactional)	2–5
Database size	50,000 objects	50,000 users / 400,000 objects
Object popularity	uniform / Zipfian	uniform
Session locality	40% (low) / 80% (high)	

[†] *read wall* is an update in metadata experiments, where page view statistics are enabled

Table 8.1: Characteristics of applications/workloads.

We implement SwiftCloud and evaluate it experimentally to determine the quality and the cost of the properties that serve our requirements, in comparison to other protocols. In this chapter, we describe our prototype and applications (Section 8.1), experimental setup (Section 8.2), and experiments (Section 8.3).

In particular, we show that SwiftCloud provides: *(i)* fast response, under 1 ms for both reads and writes to cached objects (Section 8.3.1); *(ii)* scalability of throughput with the number of DCs, and support for thousands of clients with small metadata size, linear in the number of DCs (Section 8.3.2); *(iii)* fault-tolerance w.r.t. client churn (Section 8.3.3) and DC failures (Section 8.3.4); and *(iv)* modest staleness cost, under 3% of stale reads (Section 8.3.5).

8.1 Prototype and Applications

SwiftCloud, and the benchmark applications are implemented in Java. SwiftCloud uses BerkeleyDB for durable storage (turned off in the present experiments), a custom RPC implementation, and Kryo for data marshalling [2]. A client cache has a fixed size and uses an LRU eviction policy; more elaborate approaches, such as object prefetching [18], are feasible. Applications can use predefined RDT types, as well as define their own op-based implementations.

Along the lines of previous studies of causally consistent systems [8, 12, 67, 95], we use two different benchmarks, YCSB and SocialApp, summarized in Table 8.1.

Yahoo! Cloud Serving Benchmark (YCSB) [42] serves as a kind of micro-benchmark, with simple requirements, measuring baseline costs and specific system properties in isolation. It has a simple key-field-value object model, implemented as a LWW map RDT (a generalization of LWW set), using a default payload size of ten fields of 100 bytes each. YCSB issues transactions with single-object reads and writes. We use two of the standard YCSB workloads: update-heavy Workload A, and read-dominated Workload B. The object access popularity can be set to either uniform or Zipfian. YCSB does not rely on transactional semantics or high-level RDTs.

SocialApp is a social network application modelled closely after WaltSocial [95].[1] It employs high-level data types such as sets, for friends and posts, LWW register for profile information, counter for counting profile visits, and inter-object references (implemented at the application level). Many SocialApp objects grow in size over time (e.g., sets of posts).[2] SocialApp accesses multiple objects in a causal transaction to ensure that operations such as reading a wall page and profile information behave consistently. Percentage in parentheses next to the operation type in Table 8.1 indicate the frequency of each operation in the workload. The SocialApp workload is read-dominated, but the ostensibly read-only operation of visiting a wall can actually increment the wall visit counter when statistics are enabled, in metadata experiments. The user popularity distribution is uniform.

We use a 50,000-user database for both applications, except for a smaller 10,000-user database for metadata experiments, to increase the stability of measurements.

In order to model the locality behavior of a client, which is a condition to benefit from any kind of client-side replication, both YCSB and SocialApp workloads are augmented with a facility to control access locality, mimicking social network access patterns [19]. Within a client session, a workload generator draws uniformly from a pool of session-specific objects with either 40% (*low locality*) or 80% (*high locality*) probability. For SocialApp, the pool contains objects of user's friends. Objects not drawn from this local pool are drawn from the global (uniform or Zipfian) distribution described above. The size of the local pool is smaller than the size of cache.

8.2 Experimental Setup

We run three DCs in geographically distributed Amazon EC2 availability zones (Europe, Virginia, and Oregon), and a pool of distributed clients, with the following Round-Trip Times (RTTs):

	Oregon DC	**Virginia DC**	**Europe DC**
nearby clients	60–80 ms	60–80 ms	60–80 ms
Europe DC	177 ms	80 ms	
Virginia DC	60 ms		

Each DC runs on a single m3.m EC2 instance, cheap virtual hardware, equivalent to one core 64-bit 2.0 GHz Intel Xeon virtual processor (2 ECUs) with 3.75 GB of RAM, and OpenJDK7 on Linux 3.2. Objects are pruned at random intervals between 60–120 s, to avoid bursts of pruning

[1]SocialApp implements WaltSocial's user registration operation partially, since user uniqueness with custom user names requires additional support for strong consistency.

[2]In practice, this could be an issue, however, a recent work of Briquemont et al. [26] demonstrates how to implement object sharding in a SwiftCloud-like system.

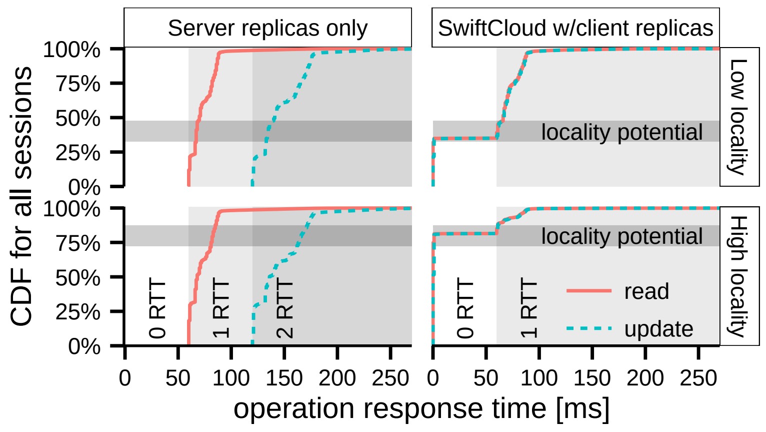


Figure 8.1: Response time for YCSB operations (workload A, Zipfian object popularity) under different system and workload locality configurations, aggregated for all client sessions.

activity. We deploy 500–2,500 clients on a separate pool of 90 m3.m EC2 instances. Clients load DCs uniformly and use the closest DC by default, with a client-DC RTT ranging in 60–80 ms.

For comparison, we provide three protocol modes based on the SwiftCloud implementation:

A. *SwiftCloud mode* (default) with client cache replicas of 256 objects, and refreshed with notifications at a rate of approx. 1 s by default;

B. *Safe But Fat metadata mode* with cache, but with client-assigned metadata only, modeled after PRACTI, or Depot without cryptography [18, 69];

C. *Server-side replication mode* without client caches. In this mode, a read incurs one RTT to a DC, whereas an update incurs two RTTs, modelling the cost of a synchronous writes to a quorum of servers to ensure fault-tolerance comparable to SwiftCloud.

8.3 Experimental Results

8.3.1 Response Time and Throughput

We run several experiments to evaluate SwiftCloud's client-side caching, with reference to the caching locality potential and geo-replication on server-side only. For each workload we evaluate the system stimulated with different rates of aggregated incoming transactions, until it becomes saturated. We use a number of clients that is throughput-optimized for each workload and protocol mode. We report aggregated statistics for all clients.

Response time. Figure 8.1 shows response times for YCSB, comparing server-side (left side) with client-side replication (right side), under low (top) and high locality (bottom), when the system is not overloaded. Recall that in server-side replication, a read incurs a RTT to the DC, whereas a fault-tolerant update incurs 2 RTTs. We expect SwiftCloud to provide much

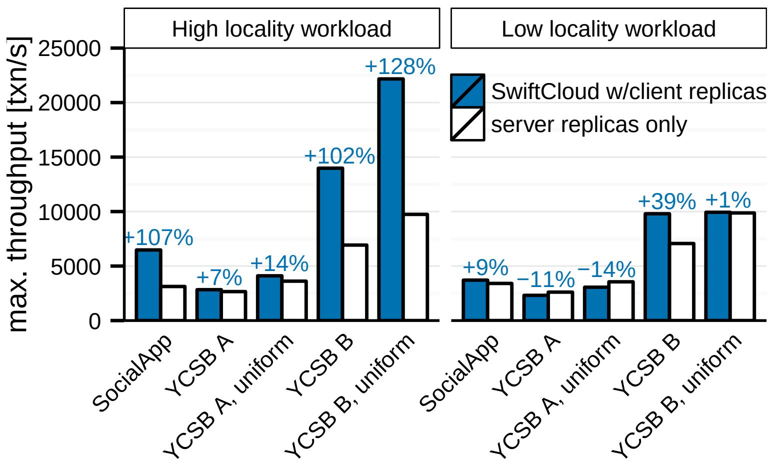


Figure 8.2: Maximum system throughput for different workloads and protocols. Percentage over bars indicates throughput increase/decrease for SwiftCloud compared to server-side replication.

faster response, at least for cached data. Indeed, the figure shows that a significant fraction of operations respond immediately in SwiftCloud mode, and this fraction tracks the locality of the workload (marked "locality potential" on the figure), within a ± 7.5 percentage-point margin.[3] The remaining operations require one round-trip to the DC, indicated as 1 RTT.

These results demonstrate that the consistency guarantees and the rich programming interface of SwiftCloud do *not* affect responsiveness of cached read and updates.

Our measurements for SocialApp show the same message, we do not report them here. An important lesson from a preliminary evaluation of SocialApp (not plotted) was that transactions involving reads on more than one object may incur multiple RTTs, compared to 1 RTT for server-side replication. We optimized our implementation to use predeclared objects and parallel reads to bring the response time down to 1 RTT. However, some transactions involving several predefined objects, yet producing small return value, could benefit from optional server-side execution support.

Maximum throughput. In the next study, we saturate the system to determine its maximum aggregated throughput. Figure 8.2 compares SwiftCloud with server-side replication for all workloads. Client-side replication is a mixed blessing: it lets client replicas absorb read requests that would otherwise reach the DC, but also puts extra load of maintaining client replicas on DCs. SwiftCloud's client-side replication consistently improves throughput for high-locality workloads, by 7% up to 128%. It is especially beneficial to read-heavy workloads. In contrast, low-locality workloads show no clear trend; depending on the workload, throughput either increases by up to

[3]A detailed analysis reveals the sources of this error margin. The default Zipfian object access distribution of YCSB increases the fraction of local accesses due to added "global" locality (up to 82 % local accesses for target 80 % of workload session locality). On the other hand, low locality workload decreases amount of local accesses, due to magnified imperfections of the LRU cache eviction algorithm (down to 34 % local accesses for target 40 %).

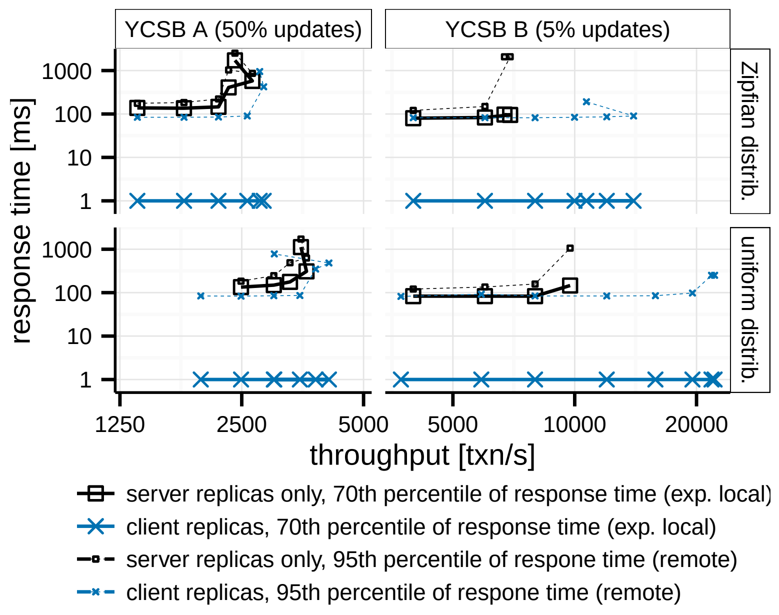


Figure 8.3: Throughput vs. response time for different system configurations and YCSB variants.

38%, or decrease by up to 11% with SwiftCloud.

Throughput vs. response time. Our next experiment studies how response times vary with server load and with the staleness settings. The results show that, as expected, cached objects respond immediately and are always available, but the responsiveness of cache misses depends on server load.

For this study, Figure 8.3 plots throughput vs. response time, for YCSB A (left side) and B (right side), both for the Zipfian (top) and uniform (bottom) distributions. Each point represents the aggregated throughput and latency for a given transaction incoming rate, which we increase until reaching the saturation point. The curves report two percentiles of response time: the lower (70 th percentile) line represents the response time for requests that hit in the cache (the session locality level is 80%), whereas the higher (95 th percentile) line represents misses, i.e., requests served by a DC.

As expected, the lower (cached) percentile consistently outperforms the server-side baseline, for all workloads and transaction rates. A separate analysis, not reported in detail here, reveals that a saturated DC slows down its rate of notifications, increasing staleness, but this does not impact response time, as desired. In contrast, the higher percentile follows the trend of server-side replication response time, increasing remote access time with DC load.

Varying the target notification rate (not plotted) between 500 ms and 1000 ms, reveals the same trend: response time is not affected by the increased staleness. At a longer refresh rate, notification batches are less frequent but larger. This increases throughput for the update-heavy YCSB A (up to tens of percent points), but has no effect on the throughput of read-heavy YCSB B.

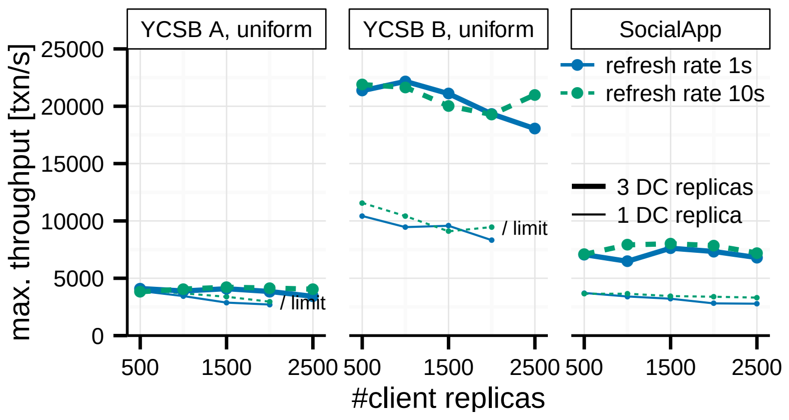


Figure 8.4: Maximum system throughput for a variable number of client and DC replicas.

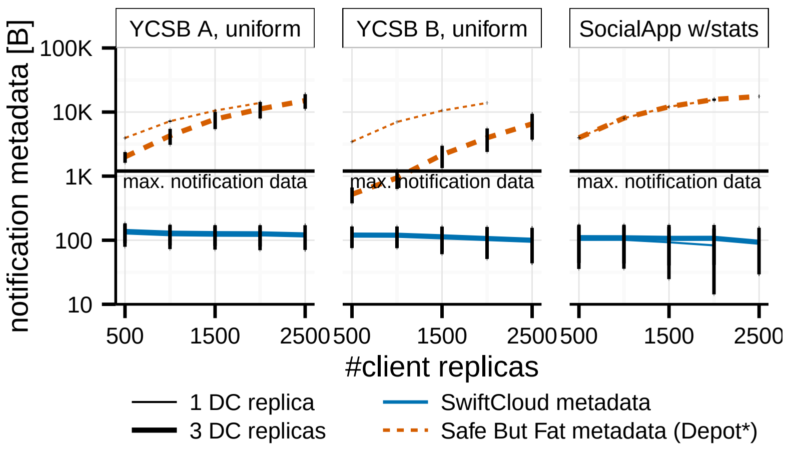


Figure 8.5: Size of metadata in notification message for a variable number of replicas, mean and standard error. Normalized to a notification of 10 updates.

We expect the impact of refresh rate to be amplified for workloads with very low rate of updates.

8.3.2 Scalability

Next, we measure how well SwiftCloud scales with increasing numbers of DC and of client replicas. Of course, performance is expected to increase with more DCs, but most importantly, the size of metadata should be small, should increase only marginally with the number of DCs, and should not depend on the number of clients. Our results support these expectations.

In this experiment, we run SwiftCloud with a variable number of client (500–2500) and DC (1–3) replicas. We report only on the uniform object distribution, because under the Zipfian distribution different numbers of clients skew the load differently, making any comparison meaningless. To control staleness, we run SwiftCloud with two notification rates: 1 s and 10 s.

Throughput. Figure 8.4 shows the maximum system throughput on the Y axis, increasing the number of replicas along the X axis. The thin lines are for a single DC, the bold ones for three DCs. Solid lines represent the fast notification rate, dashed lines the slow one. The figure shows, left to right, YCSB Workload A, YCSB Workload B, and SocialApp.

The capacity of a single DC in our hardware configuration peaks at 2,000 active client replicas for YCSB, and 2,500 for SocialApp. Beyond that, the DC drops excessive number of packets.

As to be expected, additional DC replicas increase the system capacity for operations that can be performed at only one replica such as read operations or sending notification messages. Whereas a single SwiftCloud DC supports at most 2,000 clients, with three DCs SwiftCloud supports at least 2,500 clients for all workloads. Unfortunately, as we ran out of resources for client machines at this point, we cannot report an upper bound.

For some fixed number of DCs, adding client replicas increases the aggregated system throughput, until a point of approximately 300–500 clients per DC, where the cost of maintaining client replicas up to date saturates the DCs, and further clients do not absorb enough reads to overcome these costs. Note that the lower refresh rate can reduce the load at a DC by 5 to 15%.

Metadata. In the same experiment, Figure 8.5 presents the distribution of metadata size notification messages. (Notifications are the most common and the most costly network messages.) We plot the size of metadata (in bytes) on the Y axis, varying the number of clients along the X axis. Left to right, the same workloads as in the previous figure. Thin lines are for one DC, thick lines for three DCs. A solid line represents SwiftCloud "Lean and Safe" metadata, and dotted lines the classical "Safe but Fat" approach. Note that our Safe-but-Fat implementation includes the optimization of sending vector deltas rather than the full vector [69]. Vertical bars represent standard error across clients. As notifications are batched, we normalize metadata size to a message carrying exactly 10 updates, corresponding to under approx. 1 KB of data.

The experiment confirms that the SwiftCloud metadata is small and constant, at 100–150 bytes/notification (10–15 bytes per update); data plus metadata together fit inside a single standard network packet. It is *independent* both from the number of client replicas and from the workload; and from the number of objects in the database, as an additional experiment (not plotted) validates. Increasing the number of DC replicas from one to three causes a negligible increase in metadata size, of under 10 bytes. We attribute some variability across clients (bars) to the data encoding and inaccuracies of measurements, including the normalization process.

In contrast, the classical Safe but Fat metadata grows linearly with the number of clients and exhibits higher variability. Its size reaches approx. 1 KB for 1,000 clients in all workloads, and 10 KB for 2,500 clients. Clearly, metadata being up to 10× larger than the actual data represents a substantial overhead.

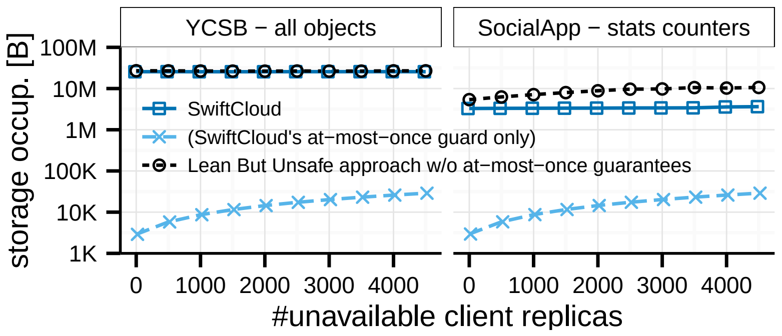


Figure 8.6: Storage occupation at a single DC in reaction to client churn for Lean-and-Safe SwiftCloud and Lean-but-Unsafe alternative.

8.3.3 Tolerating Client Churn

We now turn to fault tolerance. In the next experiment, we evaluate SwiftCloud under client churn, by periodically disconnecting client replicas and replacing them with a new set of clients. At any point in time, there are 500 active clients and a variable number of disconnected clients, up to 5000. Figure 8.6 illustrates the storage occupation of a DC for representative workloads, which is also a proxy for the size of object checkpoints transferred. We compare SwiftCloud's log compaction to a protocol without at-most-once delivery guarantees (Lean But Unsafe).

SwiftCloud storage size is approximately constant. This is achievable safely thanks to the at-most-once guard table per DC. Although the size of the guard (bottom curve) grows with the number of clients, it requires orders of less storage than the actual database itself.

A protocol without at-most-once delivery guarantees can use Lean-but-Unsafe metadata, without SwiftCloud's at-most-once guard. However this requires higher complexity in each RDT implementation, to protect itself from duplicates (cf. state-based implementations from Part II). This increases the size of objects, impacting both storage and network costs. As is visible in the figure, the cost depends on the object type: none for YCSB's LWW map, which is naturally idempotent, vs. linear in the number of clients for SocialApp's Counter objects.

We conclude that the cost of maintaining SwiftCloud's shared at-most-once guard is negligible, and easily amortized by the possible savings and the stable behavior it provides.

8.3.4 Tolerating Data Center Failures

The next experiment studies the behavior of SwiftCloud when a DC disconnects. The scatterplot in Figure 8.7 shows the response time of a SocialApp client application as the client switches between DCs. Each dot represents the response time of an individual transaction. The client runs on a private machine outside of EC2. Starting with a cold cache, response times quickly drops to near zero for transactions hitting in the cache, and to around 110 ms for misses. Some 33 s into the experiment, the current DC disconnects, and the client is diverted to another DC in a different continent. Thanks to K-stability the fail-over succeeds, and the client continues with

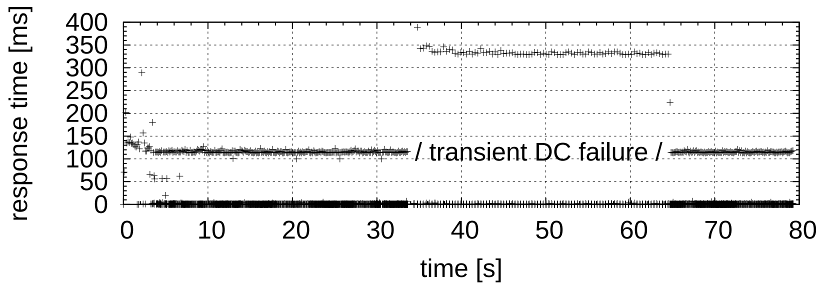


Figure 8.7: Response time for a client that hands over between DCs during a 30 s failure of a DC.

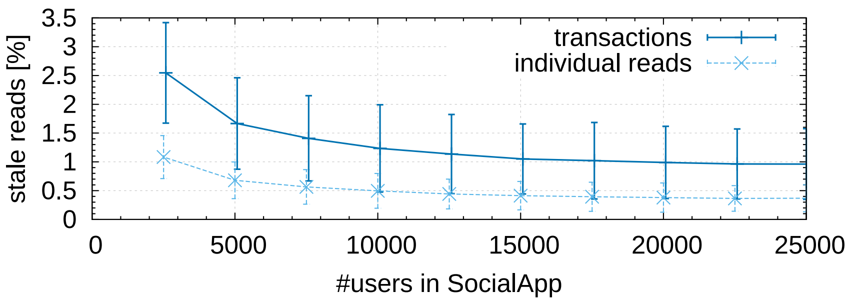


Figure 8.8: K-stability staleness overhead.

the new DC. Its response time pattern reflects the higher RTT to the new DC. At 64 s, the client switches back the initial DC, and performance smoothly recovers to the initial pattern.

Recall that a standard server-side replication system with similar fault-tolerance incurs high response time (cf. Section 8.3.1, or Corbett et al. [43]) and does not ensure at-most-once delivery.

8.3.5 Staleness Cost

The price to pay for our read-in-the-past approach is an increase in staleness. Our next experiment illustrates the impact of K-stability for evaluated workloads. A read incurs *staleness overhead* if a version more recent (but not K-stable) than the one it returns exists at the current DC of a client that performed the read. A transaction is stale if any of its reads is stale. In the experiments so far, we observed a negligible number of stale reads. The reason is that the window of vulnerability (the time it takes for an update to become K-stable) is small: approximately the RTT to the closest DC. For this experiment, we artificially increase the probability of staleness by various means: using a smaller database, setting cache size to zero, and transferring to the farthest DC, with a RTT of around 170 ms. We run the SocialApp benchmark with 1000 clients in Europe connected to the Ireland DC and replicated in the Oregon DC.

Figure 8.8 shows that stale reads and stale transactions remain under 1% and 2.5% respectively. This number decreases as we increment the size of the database. This shows that even under high contention, accessing a slightly stale snapshot has very little impact on the data read by transactions.

Chapter 9

Related Work

If somebody wants a sheep, that is a proof that one exists.

The Little Prince by Antoine de Saint-Exupéry

Contents

In this chapter, we categorize and present the work related to SwiftCloud, and compare to it. We discuss relevant theory, in Section 9.1, and relevant systems and algorithms, in Section 9.2.

9.1 Consistency Models for High Availability

Causal consistency. Ahamad et al. [4] propose the original causal consistency model for shared memory, with highly available read and writes. Their definition concerns only updates ordering, and does not embody the confluence requirement, i.e., replicas may not converge in the presence of concurrent updates (although it is rarely the case for practical implementations). More recently, Lloyd et al. [66] and Mahajan et al. [68] independently strengthen the original definition with the confluence requirement. This model is sometimes called causal+ consistency [8, 66]. Burckhardt et al. [33] provide a formal and abstract treatment of these definitions, by explicitly modelling confluence with RDTs; it is a generalization of the model we use in Chapter 2 to the multi-object setting.

Trade-offs. Mahajan et al. [68] prove that no stronger consistency model than causal consistency is available and convergent, in *full* replication system. They do not consider partial replication. We conjecture, in Section 7.1.3.2, that these properties are not simultaneously achievable under partial replication, and show how to weaken one of them. Bailis et al. [13] also study liveness and limits of a variety of weak consistency models, including causal consistency and weak transactional models. They give an argument for a similar impossibility of highly available causal consistency for a client switching server replicas, but do not take into account the capabilities of a client replica, as we do in our solution.

Transactions. Lloyd et al. propose two extensions to causal consistency that facilitate multi-object programming: read-only and write-only transactions [66, 67] (Burckhardt et al. [31] also propose a similar model). A read-only transaction reads from a consistent snapshot [66]. A write-only transaction, proposed independently of our work, ensures atomic visibility of all updates in a transaction [67]. The transactions of Lloyd are non-interactive, i.e., the application needs to predefine the set of accessed objects. SwiftCloud supports interactive read-update transactions.

Atomic visibility transactions can be considered independently from causal consistency. Bailis et al. [14] argue that atomic transactions without causality incur lower cost and are sufficiently strong. SwiftCloud, similarly to non-interactive transactional systems like Eiger [67], supports transactions that are both atomic and causal, interactively.

9.2 Relevant Systems

In this section, we overview relevant replication systems and algorithms that support consistent, available and convergent data access, at different scales, scopes, and to different extent.

For directly comparable causally-consistent partial replication systems, we evaluate their metadata in Table 9.1. The columns indicate: *(i)* the extent of support for partial replication; *(ii)* which nodes assign timestamps; *(iii)* the worst-case size of causality metadata; *(iv)* the scope of validity of metadata representing a database version (is it valid only in a local replica or anywhere); *(v)* whether it ensures at-most-once delivery; *(vi)* whether it supports general RDTs.

9.2.1 Replicated Databases for Client-Side Applications

Replication systems for client-side applications have been previously proposed in the literature and some systems have been deployed in production. Some of them use client-side replicas.

9.2.1.1 Systems that Support Inter-Object Consistency

The most relevant systems are the ones that provide causal consistency support for a large database with many objects.

PRACTI. PRACTI [18] is a seminal work on causal consistency under partial replication. PRACTI uses Safe-but-Fat client-assigned metadata and an ingenious log-exchange protocol that supports an arbitrary communication topology. It implements flexible replication options including lazy checkpoint transfer, object prefetching etc., absent in SwiftCloud implementation. While the generality of PRACTI has advantages, it is not viable for a large-scale client-side replication deployment: *(i)* its fat metadata approach (version vectors sized as the number of clients) is prohibitively expensive (see Figure 8.5), and *(ii)* any replica can make another one unavailable, because of the indirect dependence issue discussed in Section 7.1.3.2. Our cloud-backed support of client-side replication addresses these issues at the cost of lower flexibility in communication topology. We are considering support for a limited form of peer-to-peer communication that would not cause these issues, e.g., between devices of a same user or a local group of collaborators.

Depot. Our high availability techniques are similar to those of Depot [69]. Depot is a causally-consistent system, which provides client-side applications a reliable storage on top of untrusted cloud replicas. In the normal mode of operation, Depot replicates only metadata at the clients. To tolerate Byzantine cloud behavior, Depot additionally uses cryptographic metadata signatures, in order to detect misbehavior, as well as fat metadata, in order to support direct client-to-client communication. Furthermore, it either exposes updates signed by K different servers or forces clients to receive all transitive causal dependencies of their reads. This conservative approach is at odds with genuine partial replication at the client side. Although Depot does not replicate all data to the client, it does require that every client processes the metadata of *every* update, and puts the burden of computing a K-stable version on the client. In the case of extensive failures and DC partitions, it additionally floods all updates to the client, not only the fat metadata. In contrast, SwiftCloud relies on DCs to compute K-stable and consistent versions with lean metadata. In

	System	Partial replication support	Timestamp assignment	Causality metadata size O(#entries)	Version metadata validity	≤ 1 delivery	RDTs support
Safe but Fat	PRACTI [18]	arbitrary, no sharding	any replica	#replicas $\approx 10^4$–10^5	global	yes	possibly yes
	Depot [69]	partial data at client	any replica	#replicas $\approx 10^4$–10^5	global	yes	possibly yes
	COPS-GT [66]	DC sharding	database client	|causality subgraph|	local (DC)	yes	possibly yes
	Bolt-on [12]	external sharding	DC server	|explicit causality subgraph|	local (DC)	no	LWW only
Lean but Unsafe	Eiger [67]	DC sharding	DC server (shard)	#objects $\approx 10^6$	local (DC)	yes‡	counter, LWW
	Orbe [46]	DC sharding	DC server (shard)	#servers $\approx 10^2$–10^3	global†	no	LWW only
	ChainReaction [8]	DC sharding	DC (full replica)	#DCs $\approx 10^0$–10^1	local (DC)	no	LWW only
	Walter [95]	arbitrary, no sharding	DC	#DCs $\approx 10^0$–10^1	global	no	LWW only
Lean & Safe	SwiftCloud	no DC sharding, partial at client	DC (full replica) + client replica	#DCs $\approx 10^0$–10^1 + 1 client entry	global	yes	yes

† not live during DC failures
‡ only for server-side replicas

Table 9.1: Qualitative comparison of metadata used by causally consistent systems with some support for partial replication. "Timestamp assignment" indicates which nodes assign timestamps. "Causality metadata" indicates the type and maximum cardinality of entries in causality metadata, given as the expected order of magnitude (e.g., we expect hundreds to thousands servers). "Version metadata" indicates if a metadata to represent a consistent version is valid only in a local replica (DC), or globally; for the latter, if it is fault-tolerant or not. "≤ 1 delivery" indicates at-most-once delivery support.

the event of an extensive failure that cannot be handled by K-stability, SwiftCloud provides the flexibility to decrease K dynamically or to weaken consistency. Depot tolerates Byzantine clients by refusing misbehavior, or by representing late-detected misbehavior as concurrent updates. Our current implementation does not address Byzantine failures.

Both PRACTI and Depot use Safe-but-Fat metadata, as indicated in Table 9.1. They support only LWW registers, but their rich metadata could conceivably accommodate other RDTs too.

9.2.1.2 Systems that Support Intra-Object Consistency Only

Some other systems ensure consistency only for individual objects, or for a small fully-replicated database, accessed from client-side applications.

Lazy Replication. Lazy Replication technique of Ladin et al. [61] includes causal consistency protocols for client-side application accessing server replicas, which share many similarities with SwiftCloud. In Lazy Replication, every server node hosts a full replica of the database. There is no client-side replica; server replicas execute operations on behalf of the clients. A client operation issues query or update requests to one or more server replicas; update requests are asynchronously replicated across servers. To implement this scheme, Ladin et al. propose safe and lean metadata similar to the metadata we employed in SwiftCloud. It involves client-assigned timestamps to ensure at-most-once execution of client operations and server-assigned (DC-assigned) timestamps to encode operation dependencies. To achieve high availability, similarly to SwiftCloud, *(i)* an update can reach multiple servers that can assign it multiple timestamps safely; and moreover *(ii)* in a later work, Ladin et al. [62] also suggest reading only stable updates, but that forces Lazy Replication clients to execute them synchronously.

Our work differs in a number of aspects. Lazy Replication considers only global operations, at the scope of the entire database, with possibly complex conflict resolution. Instead, we structure the database into smaller RDT objects, making them the elementary replication and confluence unit, and provide causal transactions across RDT objects to implement global operations, including reads from a snapshot and atomic updates. Primarily, this allows SwiftCloud to offer partial client replicas that host only a subset of objects. We demonstrate how to leverage such client replicas to offer causal consistency with increased availability and responsiveness on cached objects (in contrast, Lazy Replication performs all operations on server replicas), and how to combine local updates with K-stable updates into a consistent view (without resorting to slow and unavailable synchronous updates of Lazy Replication). Moreover, our database structure translates into more structured implementation and provides natural parallelism potential, enabling multi-versioning and sharding techniques (cf. related work in Section 9.2.2).

Ensuring at-most-once execution incurs some cost, both in Lazy Replication and in SwiftCloud. Lazy Replication uses client-driven log compaction protocol that relies on replicas availability and loosely-synchronized clocks for progress. In contrast, SwiftCloud relies on at-most-once guard table and prunes the log aggressively. Our solution is optimized for the worst-case, whereas Lazy Replication is optimized for a favorable case where client replicas do not disappear without warning. The technique of Lazy Replication could be also incorporated into SwiftCloud. However, it comes with a trade-off, caused by clock skew and network delay, between a fraction of rejected transfer requests and log pruning period.

In addition to causal consistency, Lazy Replication offer other consistency modes for operations. In particular, it allows the application to explicitly specify causal dependencies for each operation, using application-specific knowledge. This solutions adds complexity and results in a weaker form of causal consistency, but can potentially increase performance and availability; additionally, it can serve as a tool to implement dependencies across different server-side services [62]. In addition to causal consistency, Ladin et al. also describe a protocol extension that supports strongly consistent, server-ordered or externally-ordered operations.

Bayou. Bayou [79, 97] is an influential weakly-consistent replicated database system for mobile computing environments with weak connectivity. A Bayou node replicates the full database, and communicates in peer-to-peer manner with other nodes. Bayou also predates the RDT concept, and delegates conflict detection and resolution logic to the application. The application defines a write to comprise three parts: an update, a dependency check, which specifies the conditions in which the update can be executed, and a merge procedure. A replicated write is applied locally if the dependency check succeeds. Otherwise, the merge procedure modifies the original update.

A write operation can be submitted to any Bayou node, but a single *primary* node assigns a stable reference execution order followed thereafter by all replicas; this is can be viewed as the stable prefix of an arbitration order in the RDT specification model (Section 2.3) [32]. Each Bayou server maintains two data versions: committed and tentative. The committed version reflects the execution of write operations that have been assigned a fixed position in the reference execution order. The tentative version extends the committed version with non-stable writes, for which the final position in the execution order is not known.

The main advantage of Bayou is high flexibility. It can express consistency stronger than causal, provided the application is willing to wait until its write is committed, i.e., at the expense of responsiveness and availability. However, Bayou does not tolerate all failures, its programming model is not modular, and the system does not scale with database size (due to full replication) or with system size (as it uses fat version vectors).

TouchDevelop. TouchDevelop [30] is a recent mobile application framework, somewhat similar to Bayou, but optimized for a star communication topology. A primary data replica resides in the cloud, and all mobile devices synchronize with that replica. TouchDevelop provides a programming model based on a set of predefined Cloud Types [32], which is a category of RDTs similar to op-based. It provides facilities to compose and to couple objects, using a few predefined patterns. From a consistency perspective, the database can be considered a single large object. Just like in Bayou, to implement stronger consistency, an application can enforce and await synchronous synchronization with the cloud, in order to determine the final position of recent operations in the execution order (arbitration order).

A recent work of Burckhardt et al. [35] presents TouchDevelop protocols. Although independent from SwiftCloud, both designs have major similarities. Likewise to SwiftCloud, TouchDevelop also relies on a hybrid of client-assigned and cloud-assigned update timestamps, and on a shared at-most-once guard on the cloud replica. However, it assumes only a single cloud replica (DC). Therefore, TouchDevelop *(i)* assigns at most one cloud timestamp per update, meaning that a version is identified by a scalar timestamp rather than a vector; *(ii)* can use the presence of a central replica to apply an optimized log reduction scheme that saves bandwidth and storage; and *(iii)* does not tolerate cloud replica failures (or assumes a synchronous fault-tolerant protocol). Similarly to Bayou, TouchDevelop does not support partial replication, nor a large database.

Database of independent objects. A number of other systems provide partial database replication without cross-object guarantees, i.e., they provide several independent small objects. The Google Drive Realtime API [37] is a framework for programming responsive web applications with replicated objects backed by the cloud. It provides op-based RDTs with a fixed structure, based on Operational Transformation functions to resolve concurrent updates [48]. Mobius [40] provides both data replication and messaging facilities for mobile applications backed by the cloud, with a range of cloud-controlled caching and freshness policies. Cimbiosys [83] supports partial replication with content-based filtering, supporting various topologies, such as a hierarchy of personal devices and a cloud.

Compared to SwiftCloud, the implementations of these system can be simpler or more flexible, at the expense of consistency (programming) model limited to the reduced scope of an object.

9.2.1.3 Session Guarantees

Session guarantees are desirable client-centric consistency properties for systems where a client may use several servers [96]. They include *monotonic reads*, *read-your-writes*, and others. In our context, their conjunction is equivalent to causal consistency [28, 34]. The original session guarantees algorithm of Terry et al. [96] is not available in the presence of faults, which corresponds to $K = 1$ parameter in SwiftCloud. Brzeziński et al. [29] describe a fault-tolerant protocol that also relies on K-stability. However, their protocol assumes that updates are synchronous and must be atomically accepted by K servers. In contrast, updates in SwiftCloud are asynchronous, and the transfer protocol tolerates partial failures.

9.2.2 Geo-replicated Databases for Server-Side Applications

A number of server-side geo-replicated systems support variants of causal consistency across DCs. We first present unique aspects of these approaches, and then evaluate them in the context of applicability to client-side applications, comparing them to SwiftCloud.

9.2.2.1 Approaches

Several recent geo-replication systems scale-out by *sharding* [8, 12, 46, 66, 67], which is a specific case of partial replication, where full replica is split into disjoint shards. Each shard can be replicated by a different server within a DC to increase overall throughput. However, shards require additional metadata and communication to ensure consistency.

COPS-GT. COPS-GT, a variant of COPS [66] with read-only causal transactions, is an influential design with sharding. COPS-GT assigns metadata to updates at database clients; it records fine-grained dependencies of all operations (versions) directly or indirectly observed by the client, on all shards. The dependencies have two roles. First, they ensure monotonic causal progression of DC states (cf. Section 7.1.1). COPS-GT geo-replicates an update to a shard in a remote DC together with the dependencies. The receiving server applies the update only when all dependencies are satisfied in a local DC. To verify dependencies on other shards, the server contacts respective shards servers in its local DC. Second, dependencies provide a way to compute a consistent database snapshot. COPS-GT stores dependencies with updates; therefore, servers in a DC collectively store the complete dependency graph, which they use to construct a consistent snapshot across shards.

Later publications show that using a complete dependency graph is costly [8, 12, 46, 67].

Bolt-on. Bailis et al. [12] reduce the number of dependencies by forcing application to provide (fewer) explicit causal dependencies, like in a variant of Lazy Replication. They build a weaker variant of causal consistency with a safety layer on top of a scalable eventually consistent sharded database, e.g., Cassandra [63]. A background process computes causal cut of the database, avoiding races at the underlying database.

Eiger. Eiger [67] proposes several optimizations to avoid the cost of maintaining a complete causality graph: *(i)* it assigns timestamps at a limited number of shard servers rather than clients; *(ii)* it transfers only direct update dependencies; and *(iii)* it reduces the number of dependencies on updates on a same object with coarser-grained dependencies. To compute a consistent database snapshot without a dependency graph, Eiger relies on logical timestamps assigned in a DC, and valid only locally. Eiger is the first published system that provides atomic write causal transactions. It uses a non-blocking variant of two-phase commit protocol with a coordinator. Bailis et al. [14] later generalize the commit protocol and propose decentralized alternatives.

Orbe. Orbe [46] also assigns timestamps at shard servers. Orbe generalizes COPS's depedency representation with a sparsely encoded matrix clock, with one entry per shard replica. By assigning update timestamps contiguously at each shard replica, multiple dependencies on a shard replica can be encoded with a single matrix entry, reducing the size of transmitted metadata. Orbe uses dependencies only to ensure monotonic causal progression of states, and implements read-only transactions using another mechanism. To compute a snapshot, it relies on a global real-time clock. When it reads from a snapshot at a shard server, it may need to wait out some uncertainty period to ensure consistency, risking unavailability.

ChainReaction. ChainReaction [8] takes a different approach to sharding and replication. ChainReaction uses dedicated replication protocols within a DC, based on a variant of chain replication [98].[1] The protocol is optimized for reads; it is slower for updates, as it accepts an update only after all of its dependencies are fully replicated in the local DC. For geo-replication, ChainReaction relies on a geo-replication service that is shared between all shards. This service enforces coarse-grained causal dependencies independent of shards, using version vectors, similar to the DC-assigned metadata in SwiftCloud. In order to avoid the bottleneck of sequential processing at the shared geo-replication service, ChainReaction additionally uses bloom filters to encode finer-grained dependencies, which enables more processing parallelism. Similarly, ChainReaction relies on a shared operation sequencer within a DC to implement read-only transactions. The sequencer logically serializes all updates and read transactions in a DC.

Causal dependencies and stability. All these algorithms make use of *full stability* condition to spare unnecessary causal dependencies on stable updates. Further, Du et al. [47] propose to expose only n-stable updates and client's own updates (as in SwiftCloud), to reduce dependency messages and increase throughput.

Walter. Walter [95] is a partial geo-replication system, i.e., a system without a DC sharding support, but with arbitrary partial replication at a global scale. Walter supports causal consistency with a single predefined RDT counting-set type (cf. Section 2.3.2.3) and an additional support for *strongly* consistent register type, mixing weakly and strongly consistent transactions.[2] Walter provides general purpose read-update transactions, using a metadata based on timestamp and version vectors, the same as DC-assigned metadata of SwiftCloud.

9.2.2.2 Comparison and Applicability to Client-Side Replication

Geo-replication systems offer causal consistency only to a client tied to a specific DC replica in its failure domain. None of the existing designs is directly applicable to client-side replication, both due to metadata design, and other issues, summarized in Table 9.1 and below.

Geo-replication protocols do not tolerate client or DC failures. All existing designs suffer from the transitive dependency issue from Section 7.1.3.2, violating consistency or availability for

[1]Other work treat replication in a DC as an orthogonal aspect, and assume linearizable object in a DC [46, 66, 67].
[2]More precisely, Walter offers a Parallel/Non-Monotonic variant of Snapshot Isolation [87, 95].

a remote client. They mostly rely on Lean-but-Unsafe metadata (Eiger, Orbe, ChainReaction, and Walter), which does not offer at-most-once delivery for operations on client-side replicas and assumes that data is updated by overwriting, or otherwise rely on the overly costly Safe-but-Fat metadata (COPS-GT and Bolt-on). This is a consequence of designing for LWW register support only, which makes implementing other RDTs more complex and costly (see, for example, Figure 8.6).

Another type of metadata used by these systems is a version representation, which identifies a snapshot to read, and can also serve to compare states when replicas synchronize. Many geo-replication systems use a version representation valid only in a local DC. This makes implementation of failover between DCs difficult. Orbe is one exception, as it uses a globally valid real-time clock, but its read protocol is prone to DC outages. Walter and SwiftCloud represent a version with a globally valid version vector, the same that they use for causality metadata.

The exact size of causality metadata depends on runtime factors, whereas the worst-case size is primarily impacted by the location of timestamp assignment. When fewer nodes assign metadata, it tends to be smaller, but this may limit throughput scalability with database size. For instance, SwiftCloud, Walter and ChainReaction offer the smallest guaranteed size of causality metadata, but do not provide sharding, or rely on a centralized component to support sharding.

SwiftCloud support for sharding is limited compared to the most scalable decentralized server-side designs. The presented design does not support sharding; our earlier prototype [104] (not evaluated in this thesis) offered sharding with a centralized component, similar to ChainReaction. Reconciling client-side replication with a more decentralized sharding support, without growing metadata size (like some decentralized sharding protocols, e.g., Eiger or Orbe), is future work.

Chapter 10

Conclusion

This chapter concludes the thesis with a summary of findings from both parts of the thesis, and of their limitations, and identifies areas for future work.

10.1 Summary

In this thesis, we studied the design of dependable and efficient algorithms for RDTs and causal consistency, and explored the trade-offs between dependability and metadata optimizations. Our two main contributions are: a comprehensive study of metadata space optimality for RDT implementations, including new optimized designs and impossibility results, and the design of a scalable causally-consistent RDT database for client-side applications.

Metadata optimization for state-based RDT implementations. In the first part of the thesis, we considered the problem of minimizing metadata incurred by state-based RDT implementations, i.e., objects communicating by exchanging their complete state. They can be found, for instance, in server-side geo-replicated object databases without cross-object consistency guarantees. The design of optimized implementations, to reduce the size of metadata, is nontrivial, as it creates a tension between correctness w.r.t. type semantics, efficiency, and fault-tolerance.

We formalized **worst-case metadata overhead analysis** and applied it to six existing data type implementations. We found that most of them incur substantial overhead, linear in the number of updates, or polynomial in the number of replicas. Moreover, we found that the concurrent semantics of a data type has a critical impact on the extent of the overhead.

Our first major contribution are **optimized designs of add-wins set and multi-value register** RDTs. Existing implementations store a trace of removed elements (tombstones) or of concurrently assigned values. We proposed safe optimizations based on algorithms that efficiently aggregate removed information using a variant of version vectors [77]. These optimizations reduce the metadata overhead by orders of magnitude, down to the number of replicas.

The second contribution are **lower bound proofs** on metadata overhead of any state-based implementation of a data type. A lower bound sets the limits of possible optimizations and can prove that an implementation is asymptotically optimal. We applied a common technique for proving lower bounds to six data types using a semantics-specific argument, proving optimality of four implementations (including our two optimizations), and near-optimality of another one.

These results are complementary and together offer a comprehensive view of the metadata optimization problem. It is natural to search for optimizations first, although it is a laborious process that requires developing nontrivial solutions. Our lower bounds results help to save designers from seeking the unachievable, and to lead them towards new design assumptions.

Causally-consistent object database for client-side applications. In the second part of the thesis we studied the problem of providing extended causal consistency guarantees, across object boundaries, and beyond server-side infrastructure, i.e., at the client-side. Client-side (e.g., in-browser and mobile) apps are poorly supported by the current technology for sharing mutable data over the wide-area. Existing systems ensure only some of the desired properties. They offer limited consistency or fault-tolerance guarantees, and/or do not scale.

We presented the design of SwiftCloud, the system that offers client-side apps a causally-consistent, available and convergent local access to partial database replica, i.e., an interface comparable to server-side systems. SwiftCloud relies on op-based RDT implementations, i.e., objects that transfer individual operations via log-exchange protocol. The main challenge was the design of a log-exchange protocol that ensures causal consistency and supports RDTs.

Experimental evaluation demonstrates that our **fault-tolerant cloud-backed support for partial client replicas** is able to provide immediate response on reads and updates on local objects, and to maintain consistency and throughput of a server-side replication system. Our protocols handle failures nearly transparently, at the only cost of approx. 1 % of stale reads.

SwiftCloud leverages **metadata that separates causality tracking from update identification** [61]. Our experiments demonstrated that with this metadata, a configuration with 1–3 DCs (servers) scales up safely to thousands of concurrent clients with metadata size of 15 bytes per update, independent of the number of clients (both available and unavailable ones).

SwiftCloud's design leverages a common principle that helps to achieve several goals: client buffering and controlled staleness can absorb the cost of scalability, availability, and consistency.

10.2 Limitations and Perspectives

Several aspects remain open for improvements and investigation.

Open metadata optimization problems. Some questions regarding the metadata overhead remained open. We would like to take into account a wider range of implementation categories in our metadata complexity analysis and optimizations. We also wish to improve lower bounds, or implementations, for costly data types that use arbitration to resolve concurrency, such as the LWW set (Algorithm 3.6) or the list (Section 5.1). For example, several existing list implementations suffer from high metadata size [48, 73, 76, 81, 85, 101], in the order of the number of updates. We conjecture this is necessary in a peer-to-peer communication topology, and avoidable for restricted topologies. To this end, we study variants of the list specification.

Optimized composite objects, object shards. We witness growing importance of object composition techniques [27, 52]. Composition at the specification level can translate directly into composition of object implementations [52]. However, we observe that composition permits optimizations based on composed semantic and on *metadata sharing* between composed objects (cf. our optimized add-wins set vs. a naive register-based implementation); object shards are a specific case of composite object implementations that could leverage similar metadata sharing optimizations. We are looking into ways of using faimiliar optimization patterns to facilitate, or automate, synthesis of efficient composite type implementations and shard implementations.

Scalable Data Center implementation in SwiftCloud. Our DC implementation of SwiftCloud has a centralized architecture, with limited throughput scalability. We wish to extend it with decentralized sharding support [46, 66, 67], without overly increasing the size of metadata. We believe this is possible to achieve by, once again, trading data freshness for performance, i.e., by managing a slightly stale consistent version at a high throughput, with small metadata [47].

Support for untrusted clients. Database clients cannot be trusted. An adequate protection, both for data and metadata, is missing in SwiftCloud. We are considering an adaptation of Depot's support for Byzantine clients, to detect forged metadata at the DC perimeter, and to translate any Byzantine behavior into concurrency [69]. Moreover, standard access control algorithms and new privacy mechanisms are required. For the latter, partial RDTs [26] or lenses [54] could address the problem at the object level. They are also useful independent of security concerns, as they help to save client bandwidth and to integrate server- and client-side application logic [26, 99].

Beyond causal transactions. Causal transactions with RDTs is a powerful model, but some rare operations require strongly consistent objects. For instance, we observed that our SocialApp port would benefit from strongly consistent user registration. Similarly, conflict handling (even strong) at the granularity of an object is sometimes insufficient, e.g., to enforce cascading deletes. It currently requires ad-hoc solutions in the application logic. Prior work demonstrates that combining strong and weak consistency is possible on shared data [65, 95]. We are looking into ways of integrating server-side strong consistency support, akin to the models of Burckhardt et al. [32], Ladin et al. [61], Red-Blue [65], or Walter [95].[1] Determining adequate programming models and tools, to address such challenges of eventual consistency, is an open problem [9].

[1]Server-side execution can be also beneficial for transactions that involve a lot of data to produce a small result.

Part IV

Appendix

Appendix A

Additional Material on Replicated Data Types

A.1 Formal Network Layer Specifications

Table A.1 defines formally network specifications discussed in Section 2.4.3.

A.2 Optimized Op-Based Implementations

We present some optimized op-based implementations here to better illustrate mechanisms used by the op-based category, compared to the state-based category, and their cost.

Multi-value register. Algorithm A.1 defines an optimized op-based multi-value register implementation . The optimized op-based implementations does *not* rely on vectors as the state-based implementation does (which can be costly in size, as we show in Chapter 4), but on the network layer *delivery guarantees*.

The state of an object is a set of active entries, pairs of values and unique IDs (timestamps), and a buffer of recently written or overwritten values. Read operation simply returns the values of active entries. Write operation records the value and all overwritten entries in the buffer, and replaces the set of active entries with the newly written value. Replication protocol flushes the buffer. The receiver integrates the message by removing ever local entry that was overwritten by a remote replica, and storing a new remotely written value, if any. The protocol *requires* that the latter is always a fresh value that is never overwritten locally. This can be achieved with reliable causal delivery network specification $\mathcal{T}^{1c}$ (defined in Table A.1).

Note that the state incurs almost no overhead due to removed elements or vectors, compared to the state-based implementation from Algorithm 2.5 or Algorithm 3.4. However, the correctness of the replication protocol requires stricter, more costly delivery conditions.

Specification	Symbol	Condition on executions
Any delivery	$\mathcal{T}^{\mathsf{any}}$	—
At-most-once delivery	$\mathcal{T}^{\leq 1}$	$(\forall e,f,g \in C.E : e \xrightarrow{\mathsf{del}(C)} f \wedge e \xrightarrow{\mathsf{del}(C)} g \wedge C.\mathsf{replica}(f) = C.\mathsf{replica}(g) \implies f = g)$ $(\forall e,f \in C.E : \quad e \xrightarrow{\mathsf{del}(C)} f \implies C.\mathsf{replica}(e) \neq C.\mathsf{replica}(f))$
At-least-once delivery	$\mathcal{T}^{\geq 1}$	$(\forall e \in C.E : \forall r,r' : C.\mathsf{action}(e) = \mathsf{send} \wedge C.\mathsf{replica}(e) = r \neq r' \implies \exists f \in C.E : C.\mathsf{replica}(f) = r' \wedge e \xrightarrow{\mathsf{del}(C)} f)$
Reliable delivery	$\mathcal{T}^{1}$	$\mathcal{T}^{\geq 1} \cap \mathcal{T}^{\leq 1}$
Causal delivery	$\mathcal{T}^{\mathsf{c}}$	$\forall e,f \in C.E : C.\mathsf{action}(e) = \mathsf{send} \wedge e \xrightarrow{(\mathsf{ro}(C) \cup \mathsf{del}(C))^+} f \implies e \xrightarrow{\mathsf{ro}(C)} f \vee \exists g \in C.E : e \xrightarrow{\mathsf{del}(C)} g \xrightarrow{\mathsf{ro}(C)^*} f$
Rel. causal delivery	$\mathcal{T}^{1\mathsf{c}}$	$\mathcal{T}^{1} \cap \mathcal{T}^{\mathsf{c}}$
Causal timestamps	$\mathcal{T}^{\mathsf{ct}}$	$(\mathsf{ro}(C) \cup \mathsf{del}(C))^+ \subseteq \mathsf{ar}(C)$
Eventual flush	$\mathcal{T}^{\mathsf{f}}$	$\forall e \in C.E : C.\mathsf{action}(e) = \mathsf{do} \implies \exists f \in C.E : C.\mathsf{action}(f) = \mathsf{send} \wedge e \xrightarrow{\mathsf{ro}(C)} f$

Table A.1: Catalog of popular network specifications (variation of Burckhardt et al. [34]).

Algorithm A.1 Optimized op-based multi-value integer register implementation (`MVReg`).

```
$\Sigma = \mathcal{P}(\mathbb{Z} \times \mathsf{Timestamp})^3 \quad M = \mathcal{P}(\mathbb{Z} \times \mathsf{Timestamp})^2$
$\mathsf{initialize}(r_i) : (W, b_w, b_R)$
    **let** $W = \emptyset$                  ▷ set of non-overwritten values with their unique IDs (timestamps)
    **let** $b_w = \emptyset$                                              ▷ buffer of the latest written value
    **let** $b_R = \emptyset$                                          ▷ buffer of the latest overwritten values
$\mathsf{do}(\texttt{read},\ t_o) : V$
    **let** $V = \{a \mid \exists t : (a,t) \in W\}$      ▷ return the current state modulo timestamps and the buffer
$\mathsf{do}(\texttt{write}(a),\ t_o)$
    $b_w \leftarrow \{(a,t_o)\}$                                           ▷ record the last write in the buffer
    $b_R \leftarrow b_R \cup W$                                        ▷ add overwritten values to the buffer
    $W \leftarrow \{(a,t_o)\}$                                                   ▷ update the current state
$\mathsf{send}() : (w_m, R_m)$
    **let** $(w_m, R_m) = (b_w, b_R)$                                                  ▷ flush the buffer
    $(b_w, b_R) = (\emptyset, \emptyset)$
$\mathsf{deliver}((w_m, R_m))$
    $W \leftarrow (W \setminus R_m) \cup w_m$                        ▷ keep only overwritten values; add new values
```

Similar optimizations are possible for other data types, such as add-wins set. We adopted them in our RDT implementations for the SwiftCloud system from Part III. In addition to the presented optimizations of state size, further optimizations of message size are also feasible [17].

Appendix B

Metadata Overhead Proofs

B.1 Standard Encoding

Table B.1 presents standard recursive encoding schemes for values from different domains and their asymptotic cost. These schemes are employed to encode replica state and return values throughout all our complexity proofs.

B.2 Metadata Overhead of Specific Implementations

Tight bound for an implementation is a conjunction of an upper and lower bound. In order to demonstrate upper bounds of a specific implementation, we need to make a general argument about every execution. On the other hand, to demonstrate an implementation of a data type is suboptimal, we need to show a lower bound higher than the upper bound of the optimal implementation, using a specific counter-example execution for a given n and m The following proofs follow this scheme. We do not prove implementation-specific lower bounds for optimal implementations, since they are covered by a general lower bound theorems from Section 4.2.

Theorem B.1. *Let $\mathcal{D}_{\texttt{AWSet}}$ be the naive add-wins set implementation defined in Algorithm 3.1, such that $\mathcal{D}_{\texttt{AWSet}}\ \mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]\ \mathcal{F}_{\texttt{AWSet}}$ (by [34]). The complexity of $\mathcal{D}_{\texttt{AWSet}}$ is $\widehat{O}(m \lg m)$.*

Proof. Consider any execution $C \in [\![\mathcal{D}_{\texttt{AWSet}}]\!]$ with n replicas, $m \geq n$ updates, and any `read` event e in this execution (assuming there is at least one).

Recall that the state of a set replica at the time of read is a tuple $(r, k, A, T) = \mathsf{state}(e)$, where r is a replica ID, k is a counter of locally assigned timestamps, A is a set of active instances $(a, r', k') \in S$ with element a and timestamp (r', k'), and T is a set of timestamps of removed elements. We first establish some properties of state. By definition of add, $k \leq m$ for any state and timestamp in this execution. By inspection of add, `rem`, and $\mathsf{deliver}$, the only mutators of sets A and T, an invariant $(a, r', k') \in A \implies (r', k')) \notin T$ holds. By definitions of add and $\mathsf{deliver}$, $|A| \leq m$. Similarly, by definitions of `rem` and $\mathsf{deliver}$, $|T| \leq m$. Moreover, since only add generate

Encoded object & domain $s \in S$	**Encoding scheme $\mathsf{enc}_S(s)$ and asymp. length $\mathsf{len}_S(s)$**
$u \in U$ with t.o. $\leq_U$ (e.g. $\mathbb{N}$)	standard variable length encoding $\mathsf{len}(u) \in \Theta(1+\lg\lvert U^{-1}(u)\rvert)$ where $U^{-1}(u)=\{u' \mid u' \leq_U u\}$
$(u_1,\ldots,u_k) \in U_1 \times \cdots \times U_k$	$\mathsf{enc}(u_1,\ldots,u_k)=\mathsf{enc}_{\mathbb{N}}(k)\cdot\mathsf{enc}_{U_1}(u_1)\cdot\ldots\cdot\mathsf{enc}_{U_k}(u_k)$ $\mathsf{len}(u_1,\ldots,u_k) \in \Theta(1+\sum_{i=1}^{k}\mathsf{len}_{U_i}(u_i))$
$U \subseteq U'$ s.t. $U=\{u_1,\ldots,u_k\}$	$\mathsf{enc}(U)=\mathsf{enc}(u_1,\ldots,u_k)$ $\mathsf{len}(U) \in \Theta(1+\sum_{u\in U}\mathsf{len}_{U'}(u))$
$f: U \to V$ s.t. $U=\{u_1,\ldots,u_k\}$	$\mathsf{enc}(f)=\mathsf{enc}(f(u_1),\ldots,f(u_k))$ $\mathsf{len}(f) \in \Theta(1+\sum_{u\in U}\mathsf{len}_V(f(u)))$
$f: U \rightharpoonup V$ s.t. $\mathsf{dom}(f)=\{u_1,\ldots,u_k\}$	$\mathsf{enc}(f)=\mathsf{enc}(u_1,f(u_1)),\ldots,(u_k,f(u_k)))$ $\mathsf{len}(f) \in \Theta(1+\sum_{u\in\mathsf{dom}(f)}(\mathsf{len}_U(u)+\mathsf{len}_V(f(u))))$

Table B.1: Standard encoding schemes for components of replica state, return values, and their cost. We denote concatenation of strings by $\cdot$ symbol.

new timestamps and remaining functions only transfer them between components A and T, $|A|+|T| \leq m$. Let $U(A)=\{a \mid \exists(a,r',k') \in A\}$. Then, for some constants K_1, K_2, K_3 and K_4, the state can be encoded with at most:

$$\begin{aligned}\mathsf{len}(A,T) &\leq K_1(1+\lg n+\lg m+\sum_{(a,r',k')\in A}(\mathsf{len}_{\mathbb{Z}}(a)+\lg n+\lg m)+|T|(\lg n+\lg m))\\ &\leq K_2(1+m\lg m+\sum_{(a,\_,\_)\in A}\mathsf{len}_{\mathbb{Z}}(a))\\ &\leq K_3(1+m\lg m+m\sum_{a\in U(A)}\mathsf{len}_{\mathbb{Z}}(a))\\ &\leq K_4 m\lg m(1+\sum_{a\in U(A)}\mathsf{len}_{\mathbb{Z}}(a)).\end{aligned}$$

For some constant K_5, return value $V=C.\mathsf{rval}(e)$ of `read` consumes no less than:

$$\mathsf{len}_{\mathcal{P}(\mathbb{Z})}(V) \geq K_5(1+\sum_{a\in V}\mathsf{len}_{\mathbb{Z}}(a))$$

.

By the definition of `read`, the set of returned values V matches the active elements, i.e., $V=U(A)$. Thus, for some constant K the overhead is:

$$\frac{\mathsf{len}(r,A,T)}{\mathsf{len}(V)} \leq K\frac{m\lg m(1+\sum_{a\in U(A)}\mathsf{len}_{\mathbb{Z}}(a))}{1+\sum_{a\in U(A)}\mathsf{len}_{\mathbb{Z}}(a)}=Km\lg m,$$

which satisfies the $\widehat{O}$ definition. □

Theorem B.2. *Let $\mathcal{D}_{\mathtt{AWSet}}$ be the naive add-wins set implementation defined in Algorithm 3.1, such that $\mathcal{D}_{\mathtt{AWSet}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}},\mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\mathtt{AWSet}}$ (by [34]). The complexity of $\mathcal{D}_{\mathtt{AWSet}}$ is $\widehat{\Omega}(m\lg m)$.*

Proof. Consider the following driver program (introduced in Section 4.1) defined for any positive n and m such that $m \bmod 2 = 0$:

```
1: procedure $inflate(n,m)$
2:     for all $i \in [1..(m/2)]$ do
3:         $\mathsf{do}_1$ add($i$) $^{2i}$
4:         $\mathsf{do}_1$ rem($i$) $^{2i+1}$
5:     $\mathsf{do}_1$ read $^{m+2}$                                  ▷ read $e^*$
6:     for all $i \in [2..n]$ do
7:         $\mathsf{do}_i$ read $^{m+1+i}$        ▷ dummy read on other replicas to fulfill wcmo definition
```

Given any number of replicas n_0 and any number of updates $m_0 \geq n_0$, we pick $n = n_0$ and $m = 2m_0$ to demonstrate overhead on execution:

$$C = \mathsf{exec}(\mathcal{D}_{\mathtt{AWSet}}, (\mathcal{D}_{\mathtt{AWSet}}.\mathsf{initialize}, \emptyset), \mathit{inflate}(n,m)).$$

Clearly, the execution has n replicas and m updates. The execution is well-defined since the driver program uses message identifiers and timestamps correctly. Let $(r,k,A,T) = \mathsf{state}(e^*)$ be the state of replica 1 at the time of read e^*, where A is a set of active instances and T is a set of timestamps of removed instances. By definitions of add and rem in $\mathcal{D}_{\mathtt{AWSet}}$, counter $k = m$, the set of active instances is empty for this execution, $A = \emptyset$, whereas the set of removed instances contains tombstones for all removed elements, $|T| = m/2$. Thus, for some constants K_1 and K_2, state (r,k,A,T) needs to occupy at least:

$$\mathsf{len}(r,k,A,T,) \geq K_1 \sum_{i=1}^{m/2} (\lg n + \lg m + \mathsf{len}_{\mathbb{Z}}(i)) \geq K_2 m \lg m.$$

By definition of read, the return value of read e^* is an empty set, $C.\mathsf{rval}(e^*) = \emptyset$, which can be encoded with a constant length string. Therefore, for some constant K the overhead at the time of read e^* fulfills the definition of $\widehat{\Omega}$:

$$\mathsf{wcmo}(\mathcal{D}_{\mathtt{AWSet}}, n, m) \geq \mathsf{mmo}(\mathcal{D}_{\mathtt{AWSet}}, C) \geq \frac{\mathsf{len}(r,k,A,T)}{\mathsf{len}(C.\mathsf{rval}(e^*))} \geq K m \lg m.$$

□

Theorem B.3 (Originally, Theorem 3.1). *Let $\mathcal{D}_{\mathtt{AWSet}}$ be the naive add-wins set implementation defined in Algorithm 3.1, such that $\mathcal{D}_{\mathtt{AWSet}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\mathtt{AWSet}}$ (by [34]). The complexity of $\mathcal{D}_{\mathtt{AWSet}}$ is $\widehat{\Theta}(m \lg m)$.*

Proof. By Theorem B.1 and Theorem B.2. □

Theorem B.4. *Let $\mathcal{D}^*_{\mathtt{AWSet}}$ be the optimized add-wins set implementation defined in Algorithm 3.2, such that $\mathcal{D}^*_{\mathtt{AWSet}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\mathtt{AWSet}}$ (by [34]). The complexity of $\mathcal{D}^*_{\mathtt{AWSet}}$ is $\widehat{O}(n \lg m)$.*

Proof. Consider any execution $C \in [\![\mathcal{D}^*_{\texttt{AWSet}}]\!]$ with n replicas, $m \geq n$ updates, and any `read` event e in this execution (assuming there is at least one).

Recall that the state of a set replica at the time of read is a tuple $(r, vv, A) = \mathsf{state}(e)$, where r is a replica ID, vv is a version vector (a map from replica ID to integers), and A is a set of active instances. Each instance $(a, r', k') \in A$ is made of an element a, a replica ID r', and an integer k', where (r', k') is a timestamp. We first establish some properties of state. By definition of add, send and deliver, the only mutators of vector vv, vector entries are bounded by number of updates, i.e., $vv(r') \leq m$ in any state for any replica r'. Similarly, by definition of mutators of the set of instances A, $\forall (a, r', k') \in A : k' \leq m$. Finally, thanks to the definition of operation add, there is at most n different instances of any element a in A (the optimization noted as coalescing adds). Let $U(A) = \{a \mid \exists (a, r', k') \in A\}$. Then, for some constants K_1 and K_2, component A of the state can be encoded with a string no bigger than:

$$\mathsf{len}(A) \leq K_1(1 + \sum_{(a,r',k') \in A} (\mathsf{len}_{\mathbb{Z}}(a) + \lg n + \lg m)) \leq K_2(1 + n \sum_{a \in U(A)} (\mathsf{len}_{\mathbb{Z}}(a) + \lg m)).$$

Therefore, for some constants K_3 and K_4, a complete state occupies at most:

$$\mathsf{len}(r, vv, A) \leq K_3(\lg n + n \lg m + n \sum_{a \in U(A))} (\mathsf{len}_{\mathbb{Z}}(a) + \lg m)) \leq K_4 n \lg m (1 + \sum_{a \in U(A)} \mathsf{len}_{\mathbb{Z}}(a)).$$

For some constant K_5, return value $V = C.\mathsf{rval}(e)$ of `read` consumes no less than:

$$\mathsf{len}_{\mathcal{P}(\mathbb{Z})}(V) \geq K_5(1 + \sum_{a \in V} \mathsf{len}_{\mathbb{Z}}(a)).$$

From the definition of `read`, we observe that $V = U(A)$. Thus, for some constant K, the overhead is bounded by:

$$\frac{\mathsf{len}(r, vv, A)}{\mathsf{len}(V)} \leq K \frac{n \lg m (1 + \sum_{a \in U(A)} \mathsf{len}_{\mathbb{Z}}(a))}{1 + \sum_{a \in U(A)} \mathsf{len}_{\mathbb{Z}}(a)} = K n \lg m,$$

which satisfies the $\widehat{O}$ definition. □

Theorem B.5 (Originally, Theorem 3.2)**.** *Let $\mathcal{D}^*_{\texttt{AWSet}}$ be the optimized add-wins set implementation defined in Algorithm 3.2, such that $\mathcal{D}_{\texttt{AWSet}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\texttt{AWSet}}$ (by [34]). The complexity of $\mathcal{D}^*_{\texttt{AWSet}}$ is $\widehat{\Theta}(n \lg m)$.*

Proof. By Theorem B.4 and Theorem 4.2. □

Theorem B.6. *Let $\mathcal{D}_{\texttt{MVReg}}$ be the basic multi-value register implementation defined in Algorithm 2.5, such that $\mathcal{D}_{\texttt{MVReg}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\texttt{MVReg}}$. The complexity of $\mathcal{D}_{\texttt{MVReg}}$ is $\widehat{O}(n^2 \lg m)$.*

Proof. Consider any execution $C \in [\![\mathcal{D}_{\texttt{MVReg}}]\!]$ with n replicas, $m \geq n$ updates, and any `read` event e in this execution (assuming there is at least one).

Recall that the state of a register replica used in the read is a tuple $(r, A) = \mathsf{state}(e)$, where r is a replica ID and A is a set of active entries; each entry $(a, vv) \in A$ is made of value a and a version vector vv (a mapping from a replica ID to an integer). By definitions of `write` operation and deliver function, the two mutators of entries in A, $\forall(a,vv)\in A, r'\in \mathsf{ReplicaID}.vv(r')\leq m$. Also, by definitions of `write` and deliver (alternatively, by specification $\mathcal{F}_{\mathtt{MVReg}}$ applied to visibility witness), the number of concurrent entries in the state cannot exceed the number of sources of concurrency: $|A|\leq n$. Let $U(A)=\{a\mid \exists(a,vv)\in A\}$. Then, for some constants K_1, K_2 and K_3, state (r,A) can be encoded with a string no bigger than:

$$\begin{aligned}\mathsf{len}(r,A) &\leq K_1(\lg n+\sum_{(a,vv)\in A}(\mathsf{len}(a)+n\lg m))\\ &\leq K_2(\lg n+n\sum_{a\in U(A)}(\mathsf{len}(a)+n\lg m))\\ &\leq K_3 n^2\lg m(1+\sum_{a\in U(A)}\mathsf{len}(a)).\end{aligned}$$

For some constant K_4, return value $V=C.\mathsf{rval}(e)$ of `read` consumes no less than:

$$\mathsf{len}_{\mathcal{P}(\mathbb{Z})}(V)\geq K_4(1+\sum_{a\in V}\mathsf{len}(a)).$$

From the definition of `read`, we observe $V=U(V)$. This leads to the overhead limit that holds for some constant K:

$$\frac{\mathsf{len}(r,A)}{\mathsf{len}(V)}\leq K\frac{n^2\lg m(1+\sum_{a\in U(A)}\mathsf{len}(a))}{1+\sum_{a\in U(A)}\mathsf{len}_{\mathbb{Z}}(a)}=Kn^2\lg m,$$

which satisfies the definition of $\widehat{O}$. □

Theorem B.7. *Let $\mathcal{D}_{\mathtt{MVReg}}$ be the baisc multi-value register implementation defined in Algorithm 2.5, such that $\mathcal{D}_{\mathtt{MVReg}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}},\mathcal{T}^{\mathsf{any}}]\,\mathcal{F}_{\mathtt{MVReg}}$. The complexity of $\mathcal{D}_{\mathtt{MVReg}}$ is $\widehat{\Omega}(n^2\lg m)$.*

Proof. Consider the following driver program (introduced in Section 4.1) defined for any positive n and m such that $\frac{m-n}{n}\geq 1$ and $(m-n)\bmod n=0$:

```
1: procedure inflate($n,m$)
2:     for all $r\in[1..n]$ do
3:         for all $i\in[1..(\frac{m-n}{n})]$ do
4:             do$_r$ write(0) $^{r\frac{m-n}{n}+i}$
5:         send$_r$($mid_{r,0}$)
6:     for all $r\in[1..n]$ do
7:         for all $r'\in[1..n]\setminus\{r\}$ do
8:             deliver$_r$($mid_{r',0}$)
9:         do$_r$ write(1) $^{2m+r}$
10:        send$_r$($mid_{r,1}$)
11:        if $r\neq 1$ then
12:            deliver$_1$($mid_{r,1}$)
```

```
13:    do$_1$ read $^{2m+n+1}$                                              ▷ read $e^*$
```

Given any number of replicas n_0 and any number of updates $m_0 \geq n_0$, we pick $n = n_0$ and $m \geq n_0$ such that $m-n$ is divisible by n, $(m-n) \geq n^2$ and $(m-n)^2 \geq m$. We demonstrate overhead on execution:

$$C = \mathsf{exec}(\mathcal{D}_{\mathtt{MVReg}}, (\mathcal{D}_{\mathtt{MVReg}}.\mathsf{initialize}, \emptyset), \mathit{inflate}(n,m)).$$

Clearly, the execution has n replicas and m updates. The execution is well-defined since the driver program uses message identifiers and timestamps correctly. Let $(r,A) = \mathsf{state}(e^*)$ be the state of replica 1 at the time of read e^*, where r is a replica ID and A is a set of entries (a, vv) with a value a and a version vector vv. By definition of implementation and program execution, A contains n entries $(1,vv_1),(1,vv_2),\ldots,(1,vv_n)$ with concurrent vectors, such that each vector vv_i has values: $\forall r'.vv_i(r') \geq \frac{m-n}{n}$. Therefore, for some constants K_1, K_2, K_3, K_4, K_5, K_6 encoded state (r,A) string has a length bounded by:

$$\begin{aligned}\mathsf{len}(r,V) &\geq K_1 \lg n + \sum_{i=1}^{n}(\mathsf{len}_{\mathbb{Z}}(1) + \sum_{j=1}^{n} \lg \frac{m-n}{n}) \geq K_2 n^2 \lg \frac{m-n}{n}\\ &\geq K_3 n^2 \lg \sqrt{m-n} \geq K_4 n^2 \lg(m-n) \geq K_5 n^2 \lg \sqrt{m} \geq K_6 n^2 \lg m.\end{aligned}$$

By definition of `read`, the return value V of read e^* is a singleton set, $C.\mathsf{rval}(e^*) = \{1\}$, which can be encoded with a constant length string.[1] Therefore, for some constant K the overhead at the time of read e^* fulfills the definition of $\widehat{\Omega}$:

$$\mathsf{wcmo}(\mathcal{D}_{\mathtt{MVReg}}, m, n) \geq \mathsf{mmo}(\mathcal{D}_{\mathtt{MVReg}}, C) \geq \frac{\mathsf{len}(r,A)}{\mathsf{len}(C.\mathsf{rval}(e^*))} \geq K \frac{n^2 \lg m}{1} = K n^2 \lg m.$$

□

Theorem B.8 (Originally, Theorem 3.3). *Let $\mathcal{D}_{\mathtt{MVReg}}$ be the multi-value register implementation defined in Algorithm 2.5, such that $\mathcal{D}_{\mathtt{MVReg}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\mathtt{MVReg}}$. The complexity of $\mathcal{D}_{\mathtt{MVReg}}$ is $\widehat{\Theta}(n^2 \lg m)$.*

Proof. By Theorem B.6 and Theorem B.7. □

Theorem B.9. *Let $\mathcal{D}^*_{\mathtt{MVReg}}$ be the basic multi-value register implementation defined in Algorithm 3.4, such that $\mathcal{D}^*_{\mathtt{MVReg}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\mathtt{MVReg}}$. The complexity of $\mathcal{D}^*_{\mathtt{MVReg}}$ is $\widehat{O}(n \lg m)$.*

Proof. Consider any execution $C \in [\![\mathcal{D}^*_{\mathtt{MVReg}}]\!]$ with n replicas, $m \geq n$ updates, and any `read` event e in this execution (assuming there is at least one).

Recall that the state of a register replica used in the read is a tuple $(r,A) = \mathsf{state}(e)$, where r is a replica ID and A is a set of active entries; each entry $(a,vv) \in A$ is made of value a and a version vector vv (a mapping from a replica ID to an integer). By definitions of `write` operation

[1]If the multi-value register would return a multi-set of values, it would also require a maximum of $\lg n$ extra overhead.

and deliver function, the two mutators of entries in A, $\forall(a,vv) \in A, r' \in \mathsf{ReplicaID}.vv(r') \leq m$. Let $U(A) = \{a \mid \exists(a,vv) \in A\}$. The definition of deliver ensures that every a is unique in A, i.e., $|U(A)| = |A|$. Then, for some constants K_1, K_2 and K_3, state (r,A) can be encoded with a string no bigger than:

$$\begin{aligned}\mathsf{len}(r,A) &\leq K_1(\lg n + \sum_{(a,vv)\in A} (\mathsf{len}(a) + n\lg m)) \\ &\leq K_2(\lg n + \sum_{a\in U(A)} (\mathsf{len}(a) + n\lg m)) \\ &\leq K_3 n\lg m(1 + \sum_{a\in U(A)} \mathsf{len}(a)).\end{aligned}$$

For some constant K_4, return value $V = C.\mathsf{rval}(e)$ of `read` consumes no less than:

$$\mathsf{len}_{\mathcal{P}(\mathbb{Z})}(V) \geq K_4(1 + \sum_{a\in V} \mathsf{len}(a)).$$

From the definition of `read`, we observe $V = U(V)$. This leads to the overhead limit that holds for some constant K:

$$\frac{\mathsf{len}(r,A)}{\mathsf{len}(V)} \leq K\frac{n\lg m(1 + \sum_{a\in U(A)} \mathsf{len}(a))}{1 + \sum_{a\in U(A)} \mathsf{len}_{\mathbb{Z}}(a)} = Kn\lg m,$$

which satisfies the definition of $\widehat{O}$. □

Theorem B.10 (Originally, Theorem 3.4). *Let $\mathcal{D}^*_{\mathtt{MVReg}}$ be the multi-value register implementation defined in Algorithm 3.4, such that $\mathcal{D}^*_{\mathtt{MVReg}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\mathtt{MVReg}}$. The complexity of $\mathcal{D}^*_{\mathtt{MVReg}}$ is $\widehat{\Theta}(n\lg m)$.*

Proof. By Theorem B.9 and Theorem 4.5. □

Theorem B.11 (Originally, Theorem 3.6). *Let $\mathcal{D}_{\mathtt{RWSet}}$ be the basic add-wins set implementation defined in Algorithm 3.5, assuming that $\mathcal{D}_{\mathtt{RWSet}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\mathtt{RWSet}}$. The complexity of $\mathcal{D}_{\mathtt{RWSet}}$ is $\widehat{\Omega}(m\lg m)$.*

Proof. Consider the following driver program (introduced in Section 4.1) defined for any positive n and m:

```
1: procedure inflate($n,m$)
2:     for all $i \in [1..m]$ do
3:         do$_1$ rem($i$) $^i$
4:     do$_1$ read $^{m+1}$                                  ▷ read $e^*$
5:     for all $i \in [2..n]$ do
6:         do$_i$ read $^{m+i}$      ▷ dummy read on other replicas to fulfill wcmo definition
```

Given any number of replicas n_0 and any number of updates $m_0 \geq n_0$, we pick $n = n_0$ and $m = m_0$ to demonstrate overhead on execution:

$$C = \mathsf{exec}(\mathcal{D}_{\texttt{RWSet}}, (\mathcal{D}_{\texttt{RWSet}}.\mathsf{initialize}, \emptyset), \mathit{inflate}(n, m)).$$

Clearly, the execution has n replicas and m updates. The execution is well-defined since the driver program uses message identifiers and timestamps correctly. Let $(A, T) = \mathsf{state}(e^*)$ be the state of replica 1 at the time of read e^*, where A is a set of add instances and T is a set of removed instances. By definitions of add and rem in $\mathcal{D}_{\texttt{RWSet}}$, the set of add instances is empty for this execution, $A = \emptyset$, whereas the set of remove instances contains all removed elements with timestamps of remove operations, $|T| = m$. We assume Lamport clock [64] as a timestamp t, the best known space-efficient timestamp implementation, made of a pair $t = (k, r)$, where k is a natural number, r is a replica ID, and $k \leq m$. Thus, for some constants K_1 and K_2, state (A, T) needs to occupy at least:

$$\mathsf{len}(A, T,) \geq K_1 \sum_{i=1}^{m} (\lg n + \lg m + \mathsf{len}(i)) \geq K_2 m \lg m.$$

By definition of read, the return value of read e^* is an empty set, $C.\mathsf{rval}(e^*) = \emptyset$, which can be encoded with a constant length string. Therefore, for some constant K the overhead at the time of read e^* fulfills the definition of $\widehat{\Omega}$:

$$\mathsf{wcmo}(\mathcal{D}_{\texttt{RWSet}}, n, m) \geq \mathsf{mmo}(\mathcal{D}_{\texttt{RWSet}}, C) \geq \frac{\mathsf{len}(A, T)}{\mathsf{len}(C.\mathsf{rval}(e^*))} \geq K m \lg m.$$

□

Theorem B.12 (Originally, Theorem 3.7)**.** *Let $\mathcal{D}_{\texttt{LWWSet}}$ be the last-writer-wins set implementation defined in Algorithm 3.6, assuming that $\mathcal{D}_{\texttt{LWWSet}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]\, \mathcal{F}_{\texttt{LWWSet}}$. The complexity of $\mathcal{D}_{\texttt{LWWSet}}$ is $\widehat{\Omega}(m \lg m)$.*

Proof. The proof is *identical* to the proof of Theorem 3.6 above, except for a constant factor introduced by visibility flag that does not play any role in $\widehat{\Omega}$ definition. □

Theorem B.13. *Let $\mathcal{D}^*_{\texttt{Ctr}}$ be the counter implementation defined in Algorithm 2.3, such that $\mathcal{D}^*_{\texttt{Ctr}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]\, \mathcal{F}_{\texttt{Ctr}}$ (by [34]). The complexity of $\mathcal{D}^*_{\texttt{Ctr}}$ is $\widehat{O}(n)$.*

Proof. Consider any execution $C \in [\![\mathcal{D}^*_{\texttt{Ctr}}]\!]$ with n replicas, $m \geq n$ updates, and any read event e in this execution (assuming there is at least one). Let m' be the number of inc operations visible to the read in the witness abstract execution of this execution, i.e., $m' = |\{e' \mid e' \in (\mathcal{V}^{\mathsf{state}}(C))^{-1}(e) \wedge C.\mathsf{op}(e) = \texttt{inc}\}|$; clearly, $m' \leq m$.

Recall that the state of a counter replica used in the read is a tuple $(r, v) = \mathsf{state}(e)$, where r is a replica ID and v is a map (vector) from replica ID to the number of increments made at that replica.

By definitions of `inc` and deliver, the two mutators of component v, $\forall s \in \mathsf{ReplicaID}.\, v(s) \le m'$. Therefore, for some constants K_1 and K_2, state (r,v) can be encoded as a string bounded by:

$$\mathsf{len}(r,v) \le K_1(\lg n + n\lg(m'+1)) \le K_2 n \lg(m'+1)$$

By $\mathcal{F}_{\texttt{Ctr}}$, the value of a counter must be m', hence the return value $v = C.\mathsf{rval}(e)$ encoding is bounded by: $\mathsf{len}_{\mathbb{N}}(v) \ge K_3(\lg(m'+1))$ for some constant K_3. Finally, for some constant K, metadata overhead is bounded by:

$$\frac{\mathsf{len}(r,v)}{\mathsf{len}(v)} \le K\frac{n\lg(m'+1)}{\lg(m'+1)}) = Kn,$$

which satisfies the $\widehat{O}$ definition. □

Theorem B.14 (Originally, Theorem 3.8)**.** *Let $\mathcal{D}^*_{\texttt{Ctr}}$ be the counter implementation defined in Algorithm 2.3, such that $\mathcal{D}^*_{\texttt{Ctr}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]\, \mathcal{F}_{\texttt{Ctr}}$ (by [34]). The complexity of $\mathcal{D}^*_{\texttt{Ctr}}$ is $\widehat{\Theta}(n)$.*

Proof. By Theorem B.13 and Theorem 4.1. □

Theorem B.15. *Let $\mathcal{D}^*_{\texttt{LWWReg}}$ be the last-writer-wins register implementation defined in Algorithm 2.4, such that $\mathcal{D}^*_{\texttt{LWWReg}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]\, \mathcal{F}_{\texttt{LWWReg}}$ (by [34]). The complexity of $\mathcal{D}^*_{\texttt{LWWReg}}$ is $\widehat{O}(\lg m)$.*

Proof. Consider any execution $C \in [\![\mathcal{D}^*_{\texttt{LWWReg}}]\!]$ with n replicas, $m \ge n$ updates, and any `read` event e in this execution (assuming there is at least one).

Recall that the state of a register replica used in the read is a tuple $(a,t) = \mathsf{state}(e)$, where a is a register value and t is a timestamp of the value. We assume Lamport clock [64] as a timestamp t, where timestamp is a pair $t = (k,r)$ with a natural number $k \le m$, and with replica ID r. For some constant K_1, state (a,t) encoding is bounded by: $\mathsf{len}(a,t) \le K_1(\mathsf{len}_{\mathbb{Z}}(a) + \lg m + \lg n)$

Note that return value $v = C.\mathsf{rval}(e)$ of `read` matches the register value stored in the state (cf. the definition of `read` for $\mathcal{D}^*_{\texttt{LWWReg}}$), $v = a$. Therefore, for some constants K_2, K (picked independently of a, n and m) we obtain the following bound on metadata overhead:

$$\frac{\mathsf{len}(a,t}{\mathsf{len}(v)} \le K_2\frac{\mathsf{len}(a) + \lg m + \lg n}{\mathsf{len}(a)} \le K\lg m,$$

which satisfies the $\widehat{O}$ definition. □

Theorem B.16 (Originally, Theorem 3.5)**.** *Let $\mathcal{D}^*_{\texttt{LWWReg}}$ be the last-writer-wins register implementation defined in Algorithm 2.4, such that $\mathcal{D}^*_{\texttt{LWWReg}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]\, \mathcal{F}_{\texttt{LWWReg}}$ (by [34]). The complexity of $\mathcal{D}^*_{\texttt{LWWReg}}$ is $\widehat{\Theta}(\lg m)$.*

Proof. By Theorem B.15 and Theorem 4.6. □

B.3 Lower Bound Proofs

This section contains proofs of theorems and lemmas from Section 4.2.

B.3.1 Add-Wins Set

Theorem B.17 (Originally, Theorem 4.2). *If* $\mathcal{D}_{\texttt{AWSet}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\texttt{AWSet}}$*, then the complexity of* $\mathcal{D}_{\texttt{AWSet}}$ *is* $\widehat{\Omega}(n \lg m)$*.*

Proof of Theorem 4.2. Given n_0, m_0, we pick $n = n_0$ and some $m \geq n_0$ such that $(m-1)$ is a multiple of $(n-1)$ and $m \geq n^2$. Take the experiment family $(Q, n, m, \mathbf{C}, \mathbf{e}, \mathsf{readback})$ given by Lemma 4.3. For any $\alpha \in Q$, $\mathbf{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha) = \emptyset$, which can be encoded with a constant length. Using Lemma 4.1 and $m \geq n^2$, for some constants K_1, K_2, K we get:

$$\mathsf{wcmo}(\mathcal{D}_{\texttt{AWSet}}, n, m) \geq \lfloor \lg_{|\Lambda|} |Q| \rfloor / (\max_{\alpha \in Q} \mathsf{len}(\mathbf{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha))) \geq K_1 n \lg(m/n) \geq K_2 n \lg \sqrt{m} \geq K n \lg m.$$

□

B.3.2 Remove-Wins Set

Lemma B.1 (Originally, Lemma 4.4). *If* $\mathcal{D}_{\texttt{RWSet}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\texttt{RWSet}}$*,* $m \geq n \geq 3$ *and* m *is a multiple of* 4*, then the tuple* $(Q, n, m, \mathbf{C}, \mathbf{e}, \mathsf{readback})$ *in Table 4.3 is an experiment family.*

Proof. Some parts of the proof are straightforward, such as checking that n and m match the number of replicas/updates, and that the driver programs use message identifiers and timestamps correctly. The only nontrivial obligation is to prove $\mathsf{readback}(\mathsf{state}(\mathbf{e}_\alpha)) = \alpha$. Let $(R_\alpha, N_\alpha) = \mathsf{final}(\mathbf{C}_\alpha)$. Then

$$\begin{aligned}
\alpha(r) &\overset{\text{(i)}}{=} \mathsf{result}(\mathcal{D}_{\texttt{RWSet}}, (R_0, N_0), (\mathit{init}; \mathit{exp}(\alpha); \mathit{test}(r))) \\
&= \mathsf{result}(\mathcal{D}_{\texttt{RWSet}}, (R_\alpha, N_\alpha), \mathit{test}(r)) \\
&\overset{\text{(ii)}}{=} \mathsf{result}(\mathcal{D}_{\texttt{RWSet}}, (R_{\mathit{init}}[1 \mapsto R_\alpha(1)], N_{\mathit{init}}), \mathit{test}(r)) \\
&= \mathsf{readback}(R_\alpha(1))(r) \;=\; \mathsf{readback}(\mathsf{state}(\mathbf{e}_\alpha))(r),
\end{aligned}$$

where:

(i) This is due to $\mathcal{D}_{\texttt{RWSet}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\texttt{RWSet}}$, as we explained informally before. Let

$$C'_\alpha = \mathsf{exec}(\mathcal{D}_{\texttt{RWSet}}, (R_0, N_0), (\mathit{init}; \mathit{exp}(\alpha); \mathit{test}(r)))$$

be an extension of experiment $\mathbf{C}_\alpha$ with the steps of *test* program, and $\mathsf{abs}(C'_\alpha, \mathcal{V}^{\mathsf{state}})$ be the witness abstract execution of this extension. Let v denote the value of a read in the program *test*, which is also the return value of that program. By $\mathcal{F}_{\texttt{RWSet}}$ the value of v is determined by the set of visible add and `rem` operations in the operation context, and their relation. The operation context contains all add and `rem` operations. We analyze the relation of operation in the context element-wise, since the specification $\mathcal{F}_{\texttt{RWSet}}$ is also formulated element-wise. Consider element a. The read observes the following operations on a:

(a) a sequence of `rem`(a) followed by add(a), both performed at replica 3, and concurrent

(b) $\mathtt{rem}(a)$ operation performed at replica 2, either followed by, or concurrent with, $\mathsf{add}(a)$ at replica 1, depending whether $a \in \alpha$ or not.

If $\mathtt{rem}(a)$ and $\mathsf{add}(a)$ in (b) are sequential, then $a \in v$ according to $\mathcal{F}_{\mathtt{RWSet}}$, since all $\mathtt{rem}(a)$ in the context are covered by some $\mathsf{add}(a)$. Otherwise, $a \notin v$, since there is a $\mathtt{rem}(a)$ that is not covered by any $\mathsf{add}(a)$. Therefore, by lifting the argument to all elements, $v = \alpha$.

(ii) We have $N_{init} = N_\alpha$ because $exp(\alpha)$ does not send any messages. Besides, $R_{init}[1 \mapsto R_\alpha(1)]$ and R_α can differ only in the states of the replicas $2..n$. These cannot influence the run of $test(r)$, since it performs execution steps on replica 1 only.

□

Theorem B.18 (Originally, Theorem 4.3). *If* $\mathcal{D}_{\mathtt{RWSet}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\mathtt{RWSet}}$, *then the complexity of* $\mathcal{D}_{\mathtt{RWSet}}$ *is* $\widehat{\Omega}(m)$.

Proof. Given n_0, m_0, we pick $n = \max\{n_0, 3\}$ and some $m \geq n$ such that m is a multiple of 4. Take the experiment family $(Q, n, m, \mathbf{C}, \mathbf{e}, \mathsf{readback})$ given by Lemma 4.4. For any $\alpha \in Q$, $\mathbf{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha) = \emptyset$ by $\mathcal{F}_{\mathtt{RWSet}}$ applied to the witness, which can be encoded with a constant length. Using Lemma 4.1 and $m \geq n^2$, for some constant K we get:

$$\mathsf{wcmo}(\mathcal{D}_{\mathtt{RWSet}}, n, m) \geq \lfloor \lg_{|\Lambda|} |Q| \rfloor / (\max_{\alpha \in Q} \mathsf{len}(\mathbf{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha))) \geq Km.$$

□

B.3.3 Last-Writer-Wins Set

Lemma B.2 (Originally, Lemma 4.5). *If* $\mathcal{D}_{\mathtt{LWWSet}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\mathtt{LWWSet}}$, $n \geq 2$ *and* $m \geq n$ *is such that* m *is a multiple of* $2n-2$, *then the tuple* $(Q, n, m, \mathbf{C}, \mathbf{e}, \mathsf{readback})$ *in Table 4.4 is an experiment family.*

Proof. Some parts of the proof are straightforward, such as checking that n and m match the number of replicas/updates, and that the driver programs use message identifiers and timestamps correctly. The only nontrivial obligation is to prove $\mathsf{readback}(\mathsf{state}(\mathbf{e}_\alpha)) = \alpha$. Let $(R_\alpha, N_\alpha) = \mathsf{final}(\mathbf{C}_\alpha)$. Then

$$\begin{aligned}
\alpha(r) &\overset{(i)}{=} \mathsf{result}(\mathcal{D}_{\mathtt{LWWSet}}, (R_0, N_0), (init; exp(\alpha); test(r))) \\
&= \mathsf{result}(\mathcal{D}_{\mathtt{LWWSet}}, (R_\alpha, N_\alpha), test(r)) \\
&\overset{(ii)}{=} \mathsf{result}(\mathcal{D}_{\mathtt{LWWSet}}, (R_{init}[1 \mapsto R_\alpha(1)], N_{init}), test(r)) \\
&= \mathsf{readback}(R_\alpha(1))(r) \;=\; \mathsf{readback}(\mathsf{state}(\mathbf{e}_\alpha))(r),
\end{aligned}$$

where:

(i) This is due to $\mathcal{D}_{\texttt{LWWSet}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\texttt{LWWSet}}$, as we explained informally before. Let

$$C'_\alpha = \mathsf{exec}(\mathcal{D}_{\texttt{LWWSet}}, (R_0, N_0), (\mathit{init}; \mathit{exp}(\alpha); \mathit{test}(r)))$$

be an extension of experiment $\mathbf{C}_\alpha$ with the steps of *test* program, and $\mathsf{abs}(C'_\alpha, \mathcal{V}^{\mathsf{state}})$ be the witness abstract execution of this extension. Let v_i denote value of a read into v in the i-th iteration of the loop in the program $\mathit{test}(r)$ (Line 15). By $\mathcal{F}_{\texttt{LWWSet}}$ the value of v_i is determined by the set of visible add and rem operations in the operation context, and their arbitration relation. The operation context contains:

(a) first $\max(2\alpha(r), 2i-1)$ operations ($\mathsf{add}(r)$ and $\texttt{rem}(r)$) from replica r, and

(b) unrelated operations on other elements from other replicas.

By $\mathcal{F}_{\texttt{LWWSet}}$, v must include element r if among visible updates concerning r, $\mathsf{add}(r)$ is the latest in the arbitration order $\mathsf{ar}(C'_\alpha)$. Therefore, we will now consider the arbitration relation between visible $\mathsf{add}(r)$ and $\texttt{rem}(r)$ events. If an even number of operations form r is visible ($2\alpha(r)$), then the latest one must be $\texttt{rem}(r)$. If an odd number of operations form r is visible ($2i+1$), then the latest one must be $\mathsf{add}(r)$, By specification $\mathcal{F}_{\texttt{LWWSet}}$ the return value of the read is in this case:

$$v_i = \begin{cases} \{r\} & \text{if } i - \alpha(r) > 0 \\ \emptyset & \text{otherwise} \end{cases}$$

i.e., $\{0\}$ appears in the return value when $\texttt{read}$ observes $\mathsf{add}(r)$ that was not observed in the culrpit read. Therefore:

$$\alpha(r) = \min\{i \mid v_{i+1} = \{r\} \vee i = \frac{m}{2n-2}\}.$$

(ii) We have $N_{\mathit{init}} = N_\alpha$ because $\mathit{exp}(\alpha)$ does not send any messages. Besides, $R_{\mathit{init}}[1 \mapsto R_\alpha(1)]$ and R_α can differ only in the states of the replicas $2..n$. These cannot influence the run of $\mathit{test}(r)$, since it performs execution steps on replica 1 only.

□

Theorem B.19 (Originally, Theorem 4.4). *If* $\mathcal{D}_{\texttt{LWWSet}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\texttt{LWWSet}}$*, then the complexity of* $\mathcal{D}_{\texttt{LWWSet}}$ *is* $\widehat{\Omega}(n \lg m)$*.*

Proof. Given n_0, m_0, we pick $n = n_0$ and some $m \geq n_0$ such that m is a multiple of $2(n-1)$ and $m \geq n^2$. Take the experiment family $(Q, n, m, \mathbf{C}, \mathbf{e}, \mathsf{readback})$ given by Lemma 4.5. For any $\alpha \in Q$, $\mathbf{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha) = \emptyset$ by $\mathcal{F}_{\texttt{LWWSet}}$ applied to the witness abstract execution; empty set can be encoded with a constant length. Using Lemma 4.1 and $m \geq n^2$, for some constants K_1, K_2, K we get:

$$\mathsf{wcmo}(\mathcal{D}_{\texttt{LWWSet}}, n, m) \geq \lfloor \lg_{|\Lambda|} |Q| \rfloor / (\max_{\alpha \in Q} \mathsf{len}(\mathbf{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha))) \geq K_1 n \lg(m/n) \geq K_2 n \lg \sqrt{m} \geq K n \lg m.$$

□

B.3.4 Multi-Value Register

Lemma B.3 (Originally, Lemma 4.6). *If* $\mathcal{D}_{\texttt{MVReg}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\texttt{MVReg}}$, $n \geq 2$ *and* $m \geq n$ *is such that* $(m-1)$ *is a multiple of* $(n-1)$, *then the tuple* $(Q, n, m, \mathbf{C}, \mathbf{e}, \mathsf{readback})$ *in Table 4.5 is an experiment family.*

Proof. Some parts of the proof are straightforward, such as checking that n and m match the number of replicas/updates, and that the driver programs use message identifiers and timestamps correctly. The only nontrivial obligation is to prove $\mathsf{readback}(\mathsf{state}(\mathbf{e}_\alpha)) = \alpha$. Let $(R_\alpha, N_\alpha) = \mathsf{final}(\mathbf{C}_\alpha)$. Then

$$\begin{aligned}
\alpha(r) &\overset{\text{(i)}}{=} \mathsf{result}(\mathcal{D}_{\texttt{MVReg}}, (R_0, N_0), (\mathit{init}; \mathit{exp}(\alpha); \mathit{test}(r))) \\
&= \mathsf{result}(\mathcal{D}_{\texttt{MVReg}}, (R_\alpha, N_\alpha), \mathit{test}(r)) \\
&\overset{\text{(ii)}}{=} \mathsf{result}(\mathcal{D}_{\texttt{MVReg}}, (R_{\mathit{init}}[1 \mapsto R_\alpha(1)], N_{\mathit{init}}), \mathit{test}(r)) \\
&= \mathsf{readback}(R_\alpha(1))(r) \;=\; \mathsf{readback}(\mathsf{state}(\mathbf{e}_\alpha))(r),
\end{aligned}$$

where:

(i) This is due to $\mathcal{D}_{\texttt{MVReg}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\texttt{MVReg}}$, as we explained informally before. Let v_i denote value of a read into v in the i-th iteration of the loop in the program $\mathit{test}(r)$. By $\mathcal{F}_{\texttt{MVReg}}$ the value of v_i is determined by the set and relation of visible writes in the operation context. Let

$$C'_\alpha = \mathsf{exec}(\mathcal{D}_{\texttt{MVReg}}, (R_0, N_0), (\mathit{init}; \mathit{exp}(\alpha); \mathit{test}(r))).$$

Then the operation context in $\mathsf{abs}(C'_\alpha, \mathcal{V}^{\mathsf{state}})$ of the read into v_i in $\mathit{test}(r)$ contains:

(a) first $\max(\alpha(r), i)$ operations $\texttt{write}(0)$ from replica r, and

(b) first $\alpha(r')$ operations $\texttt{write}(0)$ from every replica $r' \neq r$, and

(c) and a single $\texttt{write}(1)$ made on replica 1.

We will now analyze the visibility relation between writes $\texttt{write}(0)$ and $\texttt{write}(1)$. None of writes $\texttt{write}(0)$ observed $\texttt{write}(1)$ in its operation context in $\mathsf{abs}(C'_\alpha, \mathcal{V}^{\mathsf{state}})$. The operation context of $\texttt{write}(1)$ includes first $\alpha(r')$ operations $\texttt{write}(0)$ from every replica r', including replica r. That means, read into u_i observes $\max(0, i - \alpha(r))$ operations $\texttt{write}(0)$ from r that were not visible to $\texttt{write}(1)$. By specification $\mathcal{F}_{\texttt{MVReg}}$ the return value of the read is in this case:

$$v_i = \begin{cases} \{0,1\} & \text{if } i - \alpha(r) > 0 \\ \{1\} & \text{otherwise} \end{cases}$$

i.e., 0 appears in the return value whenever there is $\texttt{write}(0)$ concurrent to $\texttt{write}(1)$ visible. Therefore:

$$\alpha(r) = \min\{i \mid v_{i+1} = \{0,1\} \lor i = \frac{m-1}{n-1}\}.$$

Conditions on #replicas & #updates	$m \geq n \geq 2$
Index set	$Q = [1..m]$
Family size	$\lvert Q \rvert = m$

Driver programs

```
1: procedure $init$
2:     for all $i \in [1..\frac{m}{2n-2}]$ do
3:         $\mathsf{do}_2$ write($i$ mod 2) $^i$
4:         $\mathsf{send}_2(mid_i)$
5:     for all $r \in [3..n]$ do ▷ dummy
6:         $\mathsf{do}_r$ read $^{m+r}$      ▷ reads

7: procedure $exp(\alpha)$
8:     $\mathsf{deliver}_1(mid_\alpha)$
9:     $\mathsf{do}_1$ read $^{m+n-1}$ ▷ $\mathbf{e}_\alpha$
10:    ▷ (culprit read)

11: procedure $test(r)$
12:     $v \leftarrow \mathsf{do}_1$ read $^{2m}$
13:     for all $i \in [1..m)]$ do
14:         $\mathsf{deliver}_1(mid_{r,i})$
15:         $v' \leftarrow \mathsf{do}_1$ read $^{2m+i}$
16:         if $v' \neq v$ then
17:             return $i-1$
18:     return $m$
```

Definition of exp. execution $\alpha \in Q$	$\mathbf{C}_\alpha = \mathsf{exec}(\mathcal{D}_\tau, (R_0, N_0), init; exp(\alpha))$ where $(R_0, N_0) = (\mathcal{D}_\tau.\mathsf{initialize}, \emptyset)$
Definition of read-back function $\mathsf{readback} : \mathcal{D}_\tau.\Sigma \to Q$	$\mathsf{readback}(\sigma) = \mathsf{result}(\mathcal{D}_\tau, (R_{init}[1 \mapsto \sigma], N_{init}), test)$ where $(R_{init}, N_{init}) = \mathsf{post}(\mathsf{exec}(\mathcal{D}_\tau, (R_0, N_0), init))$

Table B.2: Experiment family $(Q, n, m, \mathbf{C}, \mathbf{e}, \mathsf{readback})$ used in the lower bound proof for last-writer-wins register (LWWReg).

(ii) We have $N_{init} = N_\alpha$ because $exp(\alpha)$ does not send any messages. Besides, $R_{init}[1 \mapsto R_\alpha(1)]$ and R_α can differ only in the states of the replicas $2..n$. These cannot influence the run of $test(r)$, since it performs events on replica 1 only.

□

Theorem B.20 (Originally, Theorem 4.5). *If $\mathcal{D}_{\mathtt{MVReg}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\mathtt{MVReg}}$, then the complexity of $\mathcal{D}_{\mathtt{MVReg}}$ is $\widehat{\Omega}(n \lg m)$.*

Proof. Given n_0, m_0, we pick $n = n_0$ and some $m \geq n_0$ such that $(m-1)$ is a multiple of $(n-1)$ and $m \geq n^2$. Take the experiment family $(Q, n, m, \mathbf{C}, \mathbf{e}, \mathsf{readback})$ given by Lemma 4.6. Then for any α, $\mathbf{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha) = \{1\}$ by $\mathcal{F}_{\mathtt{MVReg}}$ applied to the witness abstract execution; singleton set can be encoded with a constant length string. Using Lemma 4.1 and $m \geq n^2$, for some constants K_1, K_2, K_3, K independent from n_0, m_0 we get:

$$\mathsf{wcmo}(\mathcal{D}_{\mathtt{MVReg}}, n, m) \geq \lfloor \lg_{|\Lambda|} |Q| \rfloor / (\max_{\alpha \in Q} \mathsf{len}(\mathbf{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha))) \geq K_1 \frac{\lg_{|\Lambda|} (\frac{m-1}{n-1})^{n-1}}{\mathsf{len}_{\mathcal{P}(\mathbb{Z})}(\{0\})}$$
$$\geq K_2 \frac{n \lg(m/n)}{1} \geq K_3 n \lg \sqrt{m} \geq K n \lg m.$$

□

B.3.5 Last-Writer-Wins Register

Theorem B.21 (Originally, Theorem 4.6). *If $\mathcal{D}_{\mathtt{LWWReg}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathcal{T}^{\mathsf{any}}]$ $\mathcal{F}_{\mathtt{LWWReg}}$, then the complexity of $\mathcal{D}_{\mathtt{LWWReg}}$ is $\widehat{\Omega}(\lg m)$.*

Lemma B.4. *If* $\mathcal{D}_{\mathtt{LWWReg}}\ \mathsf{sat}[\mathcal{V}^{\mathsf{state}},\mathcal{T}^{\mathsf{any}}]\ \mathcal{F}_{\mathtt{LWWReg}}$*,* $n \geq 2$ *and* $m \geq n$*, then tuple* $(Q,n,m,\mathbf{C},\mathbf{e},\mathsf{readback})$ *as defined in Table B.2 is an experiment family.*

The idea of the experiments is to force replica 1 to remember the number of writes that the write of the current value dominates, which then introduces an overhead proportional to the $\lg m$, the minimum cost of storing a timestamp; cf. the implementation in Algorithm 2.4. Only two replicas are involved the experiment, since the overhead is not related to the number of replicas. Other replicas perform only dummy reads to ensure that execution uses at least n replicas, and fulfills wcmo definition.

All experiments start with a common initialization phase, defined by *init*, where replica 2 performs m `write` operations, writing interchangeably values 0 and 1, starting with 1, and sending a message after each write. For each write operation we define a timestamp such that the execution of driver program consists of the arbitration order consistent with the order of operations on replica 2. All messages send in the *init* phase of the program are undelivered until the second phase, defined by $exp(\alpha)$. There replica 1 receives exactly one message from replica 2, selected using α. An experiment concludes with the read $\mathbf{e}_\alpha$ on the first replica.

The read-back works by performing a test defined by *test* program. To determine which message was delivered by replica 1 during the experiment, it first performs a reference read on the first replica, and then forces (re)delivery of all messages sent by replica 2 to replica 1, in the order they were sent, performing a read after each message delivered. By comparing return value of each such read with the reference read, the program identifies the first message that makes the read include an outcome of `write` operation dominating in arbitration order all `write` operations observed in the reference read; the index of this message corresponds to $\alpha+1$.

Proof of Lemma B.4. Some parts of the proof are straightforward, such as checking that n and m match the number of replicas/updates, and that the driver programs use message identifiers and timestamps correctly. The only nontrivial obligation is to prove $\mathsf{readback}(\mathsf{state}(\mathbf{e}_\alpha)) = \alpha$. Let $(R_\alpha, N_\alpha) = \mathsf{final}(\mathbf{C}_\alpha)$. Then

$$\begin{aligned}
\alpha(r) &\overset{\text{(i)}}{=} \mathsf{result}(\mathcal{D}_{\mathtt{LWWReg}},(R_0,N_0),(init;exp(\alpha);test(r)))\\
&= \mathsf{result}(\mathcal{D}_{\mathtt{LWWReg}},(R_\alpha,N_\alpha),test(r))\\
&\overset{\text{(ii)}}{=} \mathsf{result}(\mathcal{D}_{\mathtt{LWWReg}},(R_{init}[1\mapsto R_\alpha(1)],N_{init}),test(r))\\
&= \mathsf{readback}(R_\alpha(1))(r) \ =\ \mathsf{readback}(\mathsf{state}(\mathbf{e}_\alpha))(r),
\end{aligned}$$

where:

(i) This is due to $\mathcal{D}_{\mathtt{LWWReg}}\ \mathsf{sat}[\mathcal{V}^{\mathsf{state}},\mathcal{T}^{\mathsf{any}}]\ \mathcal{F}_{\mathtt{LWWReg}}$, as we explained informally above. Let

$$C'_\alpha = \mathsf{exec}(\mathcal{D}_{\mathtt{LWWReg}},(R_0,N_0),(init;exp(\alpha);test)).$$

Then the operation context in $\mathsf{abs}(C'_\alpha,\mathcal{V}^{\mathsf{state}})$ of the reference read into v in *test* contains α first `write` operations. By $\mathcal{F}_{\mathtt{LWWReg}}$, v must reflect value of the visible `write` that is the latest in the arbitration order $\mathsf{ar}(C'_\alpha)$. By assigned timestamps the arbitration order corresponds to the order of `write` operations on replica 2. Hence, $v = \alpha \bmod 2$.

Let v'_i denote value of a read into v' in the i-th iteration of the loop in the program *test*. The operation context of read into v'_i in $\mathsf{abs}(C'_\alpha, \mathcal{V}^{\mathsf{state}})$ contains $\max(i,\alpha)$ first `write` operations performed at replica 2. Again, by specification $\mathcal{F}_{\texttt{LWWReg}}$ the value must be $v'_i = \max(i,\alpha) \bmod 2$.

By comparing values of v and all v'_i, we obtain:

$$\alpha = \min\{i \mid v \neq v'_{i+1} \vee i = m\}.$$

(ii) We have $N_{init} = N_\alpha$ because $exp(\alpha)$ does not send any messages. Besides, $R_{init}[1 \mapsto R_\alpha(1)]$ and R_α can differ only in the states of the replicas $2..n$. These cannot influence the run of *test*, since it performs events on replica 1 only.

□

Proof of Theorem 4.6. Given n_0, m_0, we pick $n = n_0$ and $m = m_0$. Take the experiment family $(Q, n, m, \mathbf{C}, \mathbf{e}, \mathsf{readback})$ given by Lemma B.4. Then for any α, return value $\mathbf{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha) \in \{0,1\}$ by $\mathcal{F}_{\texttt{LWWReg}}$ applied on the witness execution; set $\{0,1\}$ can be encoded with a constant length string. Using Lemma 4.1 for some constants K_1, K independent from n_0, m_0 we get:

$$\mathsf{wcmo}(\mathcal{D}_{\texttt{LWWReg}}, n, m) \geq \lfloor \lg_{|\Lambda|} |Q| \rfloor / (\max_{\alpha \in Q} \mathsf{len}(\mathbf{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha))) \geq K_1 \frac{\lg_{|\Lambda|}(m)}{1} \geq K \lg m.$$

□

Appendix C

Résumé de la thèse

La haute disponibilité, ainsi que la réactivité sont les qualités essentielles des applications web interactifs partageant les données modifiables. Pour assurer ces exigences, les applications comptent sur les bases de données *géo-réplicatives* répliquant les données dans des endroits géographiquement dispersés à travers le monde. La géo-réplication permet aux utilisateurs d'accéder à la réplique locale des données située dans le centre de donnée le plus proche ou sur leur propre station. De plus, l'accès aux données locales n'est pas entravé par le coût de la latence du réseau connectant les répliques, ni par l'impact des défaillances de l'infrastructure [43, 44, 95]. Lorsque les performances et la disponibilité des opérations de mise à jour sont cruciales pour le fonctionnement de l'application, les opérations de mises à jour doivent être également réalisées localement, sans la coordination avec les répliques distantes. Cela implique une acceptation des *mises à jours simultanées* se reproduisant d'une façon asynchrone. Malheureusement, cela est en conflit avec les modèles de forte cohérence de données, notamment avec l'atomicité ou la sérialisation [20, 53]. La forte cohérence offre aux applications une vue unique de la base de données distribuée, mais nécessite une exécution des opérations suivant un ordre global sur toutes les répliques à l'aide de la réplication synchrone. L'incompatibilité entre la disponibilité, la tolérance aux pannes et la forte cohérence, connue sous le nom du théorème CAP, force la réplication asynchrone à se recourir aux plus faibles *modèles de cohérence à terme* [44, 50, 79].

Dans le cas de la cohérence à terme, les répliques sont autorisées à s'écarter de façon transitoire, par exemple, au cours des mises à jour simultanées ou pendant une panne de réseau. Cependant, les répliques doivent, à terme, converger vers un état commun intégrant toutes les mises à jour [79, 100]. Les états intermittents exposent l'application au risque des anomalies de cohérence, qui sont non seulement embarrassantes pour les utilisateurs, mais aussi complique la réalisation de l'application, ainsi que de la base de données [43]. Les défis comprennent surtout la détection des mises à jours simultanés étant en conflit, leur résolution convergente, ainsi que le traitement des livraisons asynchrones des mises à jour et des échecs.

Deux abstractions complémentaires ont été proposées pour atténuer les problèmes de la réplication dans le cas de la cohérence à terme. Ces abstractions prennent en compte la complexité de la réplication asynchrone et les échecs derrière une interface *fiable*, à savoir une interface avec un comportement bien défini, qui est garanti quel que soit le comportement de l'infrastructure sous-jacente.

Tout d'abord, les *Types de Données Répliquées* (RDTs) exposent les données de la base de données comme des objets classifiés possédant des méthodes de haut niveau. Elle se basent sur le type et la sémantique de la méthode pour assurer la convergence vers un état raisonnable [34, 92]. La base de données comprenant des RDTs contient des objets de types tels que le compteur, l'ensemble où le registre, possédant des méthodes de l'écriture et de mise à jour, tel que l'incrémentation pour le compteur ou l'ajout et la suppression pour l'ensemble. Un type de données encapsule la réplication et s'oocupe de la résolution des mises à jour simultanées en se basant sur la sémantique définie. Ainsi, les RDTs fournissent un fiable bloc de construction.

Deuxièmement, la *cohérence transactionnelle causale* offre un ordonnancement partiel des mises à jour et des garanties de visibilité indépendantes des frontières de l'objet [4, 66, 67]. Officieusement, sous cohérence causale, tous les processus de l'application observent un ensemble non-décroissant de mises à jour, qui comprend ses propres mises à jour, dans l'ordre respectant les causalités entre les opérations. Les applications sont protégées des violations de causalité, par exemple d'une situation où la méthode d'écriture pourrait observer la mise à jour de b, mais ne pourrait pas mettre à jour a sur laquelle b se baserait.

Ces abstractions fiables facilitent la programmation de la base de données cohérente à terme, mais entrennent un cout de stockage et un surcoût des métadonnées dans le réseau causé par l'information de la causalité et l'historique des opérations. Cette thèse étudie la conception des fiables RDTs et des fiables algorithmes de cohérence causale utilisant des métadonnées minimisées, ainsi que les limites et les compromis de leur espace de conception.

Contributions. Cette thèse apporte deux contributions dans le domaine de la fiabilité des algorithmes de la cohérence à terme: une étude de l'espace optimale des implémentations des RDTs, y compris des mises en œuvre optimisées et des résultats d'impossibilité, ainsi que la conception des bases de données causalement cohérentes incluant les RDTs pour les applications côté client. Nous présentons un aperçu de ces résultats dans le reste de ce résumé.

C.1 L'optimalité des types les données répliquées

Dans la première partie de la thèse, nous considérons le problème de minimisation des métadonnées encourues par les implémentations des RDTs. En plus des données observables par le client, les métadonnées sont utilisées par les réalisations des RDTs fiables pour assurer l'exactitude dans le cas des mises à jour simultanées et des échecs. Pour beaucoup de réalisations des RDTs, le surcoût causé par les métadonnées est important et impacte le rendement, le stockage et le coût de la bande passante, ou encore la viabilité de la réalisation. Malheureusement, l'optimisation des réalisations dans le but de réduire la taille des métadonnées est non-triviale. Elle crée des tensions entre l'exactitude de la sémantique des types, de l'efficacité et de la tolérance aux pannes.

Nous formulons et étudions la minimisation des métadonnées pour une catégorie de réalisations des RDTs *basée sur l'état*, par exemple sur des objets communiquant en échangeant leur état complet [92]. Des réalisations similaires sont utilisées, par exemple, par les bases de données d'objets [3]. Nous définissons une métrique du surcoût des métadonnées nous permettant de conduire une analyse du pire des cas des réalisations, ainsi que d'exprimer le surcoût asymptotique en fonction du nombre de répliques et des mises à jour. Une analyse des réalisations de type de données existantes indique que beaucoup d'entre eux encourent un surcoût important, linéaire en fonction du nombre des mises à jour, des répliques où des deux éléments. Nous observons que la sémantique simultanée d'un type de données, à savoir le comportement sous mises à jour simultanées, possède un impact critique sur l'étendue exacte de la surcharge, même parmi les variantes de types partageant la même interface (par exemple, différentes variantes de la sémantique répliquée de l'ensemble).

Les deux principales contributions de cette partie de la thèse ont mené aux *résultats positifs*, à savoir aux réalisations optimisées de type de données, ainsi qu'aux *résultats négatifs*, à savoir aux preuves du minorant du surcoût des métadonnées pour un certain type de données. Il est naturel de rechercher tout d'abord des résultats positifs. Par exemple, nous avons constaté, que l'un des types de métadonnées commun constitue une trace des éléments supprimés (des pierres tombales) ou des valeurs écrasées. Nous proposons deux optimisations en se basant sur un algorithme subtil, mais efficace, qui rassemble les informations sur les données supprimées/écrasées en utilisant une variante des vecteurs de version [77]. L'optimisation est un processus intellectuel laborieux, qui nécessite le développement des solutions non triviales, et qui expose au risque la justesse de la solution existante. Nos résultats négatifs peuvent aider à identifier la fin du processus de l'optimalisation. Une fin bien définie empêcherait les concepteurs de mener une recherche irréalisable et pourrait les guider dans le développement de nouvelles hypothèses de

Type de données	**Ancienne réalisation**		**Réal. optimisé**	**Tout réal.**
	Source	**Bornes**	**Bornes**	**Minorant**
compteur	$\underline{\text{[92]}}$	$\underline{\widehat{\Theta}(n)}$	—	$\widehat{\Omega}(n)$
ensemble ajout-gagnant	[93]	$\widehat{\Theta}(m \lg m)$	$\underline{\widehat{\Theta}(n \lg m)}$	$\widehat{\Omega}(n \lg m)$
ensemble supréssion-gagnante	[22]	$\widehat{\Omega}(m \lg m)$	—	$\widehat{\Omega}(m)$
ensemble dérnier-écrivaint-gagnant	[22, 56]	$\widehat{\Omega}(m \lg m)$	—	$\widehat{\Omega}(n \lg m)$
registre multi-valeurs	[77, 93]	$\widehat{\Theta}(n^2 \lg m)$	$\underline{\widehat{\Theta}(n \lg m)}$	$\widehat{\Omega}(n \lg m)$
registre dérnier-écrivaint-gagnant	$\underline{\text{[56, 92]}}$	$\underline{\widehat{\Theta}(\lg m)}$	—	$\widehat{\Omega}(\lg m)$

Table C.1: Sommaire des résultats de surcharge pour différents types de données. Les réalisations surlignées sont optimales; n est le nombre de répliques, m est le nombre de mises à jour dans une exécution; $\widehat{\Omega}, \widehat{O}, \widehat{\Theta}$ notées, respectivement, minorant, majorant et borne exacte.

conception. Une preuve du minorant, qui s'applique à *tout* réalisation basée sur l'état d'un certain type, établie un limite d'optimalisations possibles et peut prouver qu'une réalisation particulière est asymptotiquement optimale. Nos preuves du minorant utilisent une construction générique non-triviale basée sur la sémantique du spécifique du type de données concerné . Réunis, nos résultats positifs, ainsi que négatifs, donnent une vision globale du problème d'optimisation des métadonnées.

Table C.1 résume nos résultats positifs et négatifs pour tous les types de données étudiés. De gauche à droite, le tableau présente les implémentations antérieures et leurs complexités, la complexité de nos optimisations (dans le cas échéant), ainsi que les bornes inférieures de la complexité. Nous soulignons chaque réalisation asymptotiquement optimale, soit une optimalisation avec une borne supérieure correspondante à la borne inférieure générale.

Pour le type de données des compteurs, la réalisation existante basée sur l'état est optimale. Une telle réalisation exige l'utilisation des vecteurs d'entiers pour tenir compte des augmentations simultanées, ainsi que des messages simultanée ou réordonnées.

L'interface de type de données de l'ensemble offre une large gamme de choix sémantique en accord avec la façon avec laquelle les opérations concurrentes sont traitées sur le même élément: la priorité devrait être accordée pour certain type d'opérations (ajout ou suppression), par contre d'autre opération nécessites d'être arbitrées par des horodateurs (dernier–écrivain-gagnant) [23, 56]. Ces différents types de procédure sont inégaux en termes de coût des métadonnées. Toutes les réalisations antérieures subissent le problème du surcoût de l'ordre de grandeur de nombres de mises à jour correspondant au moins aux traces des éléments supprimés. Cependant, cela n'est pas nécessaire dans tous les cas de variante d'ensemble. Notre optimisation d'un ensemble ajout-gagnant réduit ce coût au nombre des répliques en utilisant une adaptation non-triviale des vecteurs de versions. De plus, cela est (asymptotiquement) la moins cher des réalisations connues pour toutes sortes d'ensembles. La borne inférieure pour l'ensemble suppression-gagnante montre, que telle optimalisation n'est pas possible dans le cas d'une sémantique suppression-gagnante. La question si cela est possible pour un ensemble dernier-écrivain-gagnant reste ouverte.

Le type de registre possède aussi de nombreuses variantes, qui eux aussi sont inégaux en terme de coût des métadonnées encouru. Une variante du registre de type dernier-écrivain-gagnant utilise le horodatage pour arbitrer les attributions simultanées, tandis que le registre multi-valeur identifie les valeurs de tous les conflits d'écriture pour les présenter à l'application. La réalisation existante du registre dernier-écrivain-gagnant est caractérisée par un surcoût négligeable et est considérée comme optimale. Au contraire, la réalisation actuelle du registre à valeurs multiples possède un surcoût important en terme de carré de nombre de répliques due à un traitement inefficace des écritures simultanées de la même valeur. Nos optimisations atténuent la composante carré, et atteignent la surcharge de métadonnées asymptotiquement optimal en utilisant une règles de fusion pour les vecteurs de versions.

C.2 Une base de données causalement cohérente pour les applications coté client

Dans la deuxième partie de la thèse, nous étudions le problème de la fourniture des garanties étendues de la cohérence (causale), à travers les frontières de l'objet et au-delà de l'infrastructure côté serveur, pour les applications côté client.

Nous considérons le problème de la réplication côté client. La technologie actuelle est peu adaptée pour supporter le partage de données envers une large zone dans des applications côté client, comme par exemple dans le navigateur ou les applications mobiles. Les développeurs d'applications font recours à la mémoire cache et aux tampons, afin d'éviter des lents, coûteux et parfois indisponible allers-retours au centre de données. Cependant, ils ne sont pas capables de résoudre les problèmes du système tels que la tolérance aux pannes, ou les garanties de cohérence / session [34, 96]. Les systèmes côté client assurent seulement quelques-unes des propriétés souhaitées. Notamment, ils assurent des garanties de cohérence limitées (seulement à la granularité d'un objet unique ou celle d'une petite base de données), ne tolèrent pas les défaillance et/ou ne peuvent pas intégrer un grand nombre de périphérique clients.

Notre thèse est que le système doit être responsable d'assurer un accès correct et évolutif à la base de données aux applications côté client. Il doit répondre aux exigences (parfois contradictoires) de la cohérence, de la disponibilité et de la convergence [68] au moins aussi bien que les systèmes de géo-réplication récents. Dans ces conditions, le modèle de cohérence le plus fort est une variante de la *cohérence causale pour les objets de RDTs*.

La possibilité d'offrir des milliers ou mêmes des millions de répliques côté client avec la cohérence causale conteste les hypothèses standard. Pour tracer avec précision et par client la causalité dans et entre les objets il faut utiliser un nombre inacceptable de métadonnées. De l'autre côté, un management compact des métadonnées du côté serveur possède une tolérance aux pannes plus faible. De plus, une réplication complète dans un grand nombre de dispositifs pauvres en ressources serait inacceptable [18]. Mais aussi une réplication partielle des données et des métadonnées pourrait être la source d'indisponibilité ou des anomalies de livraison de messages. Par ailleurs, il n'est pas possible de supposer, comme dans nombreux systèmes antérieurs, que la tolérance aux pannes et la cohérence sont résolues, seulement parce que l'application est localisé à l'intérier d'un centre de données (DC), ou parce qu'elle possède une session collante avec un seule DC [13, 96].

Dans cette partie de la thèse, nous abordons les défis mentionnés. Nous présentons les algorithmes, la conception et l'évaluation du SwiftCloud — la première base de données d'objets distribuée conçue pour un grand nombre de répliques. Elle assure d'une façon efficace un accès constant, disponible, et convergent aux nœuds clients, supporte les défaillances et consume peut de métadonnées. Pour atteindre ce but, le SwiftCloud utilise une topologie de client-serveur flexible, et découple l'écriture et la lecture. Le client *écrit rapidement* dans le cache local, et *lit dans le passé* (aussi rapidement) les données qui sont compatibles, mais parfois altérées.

C.2.1 Présentation du problème

Nous considérons le support pour une variété d'applications côté clients partageant une base de données des objets de RDTs, que le client peut lire ou modifier. Nous visons à s'adapter aux milliers de clients couvrant l'ensemble de l'internet, ainsi qu'à une base de données de taille arbitraire.

Figure C.1 présente le modèle de notre système. Une infrastructure “cloud” relie un petit ensemble (par exemple, une dizaine) de centres de données géo-réplicatifs, ainsi qu'un grand nombre (des milliers) de clients. Un DC dispose d'abondantes ressources de calcul, de stockage et de réseau. De même que Sovran et al. [95], nous faisons abstractions d'un DC comme un processus séquentiel hébergeant des **répliques complètes** de la base de données.[1] Le DC

[1] Les travaux antérieurs abordent les questions un peu orthogonales du parallélisme et de la tolérance aux pannes dans un DC [8, 46, 66, 67].

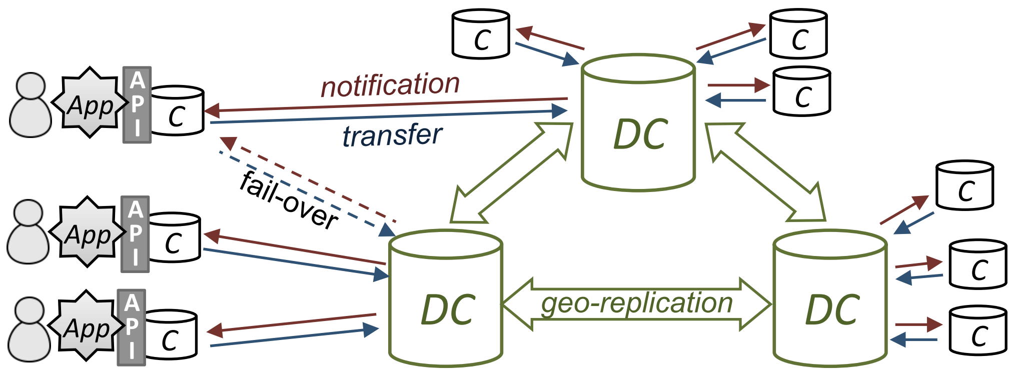


Figure C.1: Les composantes du système (Les processus d'application, Clients, Centre de Donnés) et leurs interfaces.

communique d'une manière pair à pair. Il peut tomber en panne et se rétablir en laissant sa mémoire persistante intacte.

Les clients ne communique pas directement entre eux, mais ils passent par les DCs. D'habitude, un client se connecte à un seul DC ; dans le cas d'une panne ou d'un client migrant , à zéro ou plus. Un client peut tomber en panne et se rétablir (par exemple, durant un vol) ou être détruit (par exemple, un téléphone détruit) sans avertissement.

Les applications côté client souvent exige une haute **disponibilité**, ainsi qu'une haute **réactivité** pour satisfaire pleinement la demande de l'utilisateur. Elles doivent être capables de lire et écrire les données rapidement et à tous moments. Cela peut être réaliser en répliquant les données en local, et en synchronisant les mis à jour dans la tâche de fond. Cependant, un client possède des ressources limitées. En cette raison, il dispose d'une **mémoire cache** contenant seulement un sous-ensemble de la base de données intéressant de point de vue de l'application courante. Il n'est pas supposé de recevoir des messages relatifs aux objets non-répliqués au moment donné. [88]. Finalement, les messages de control, ainsi que les métadonnées venant avec, devrait être d'une taille petite et délimitée.

Puisque la réplique du client et seulement *partielle*, nous ne pouvons pas avoir une garantie d'une disponibilité complète. Le meilleur que nous pouvons expecter est la **disponibilité partielle**, dans le cas de laquelle l'opération retourne le résultat sans communication distante, si les données demandés sont dans le cache, et récupère les données d'un nœud distant dans le cas contraire. Si les données ne sont pas disponibles et le réseau est en panne, l'opération peut s'avérer non-disponible. Dans ce cas là elle serra bloquée ou elle retournera une erreur.

C.2.1.1 La cohérence et la convergence

Les développeurs d'application souhaitent accéder à une vue consistante de la base de données globale. Néanmoins, vu l'exigence de la disponibilité, les possibilités de consistance sont limitées selon le théorème CAP mentionnée avant [50, 68].

Dans cette partie de la thèse, nous considérons un support pour le modèle de cohérence le plus fort et convergeant : le modèle de la cohérence causale pour les objets de RDTs [4, 68].[2] Dans le cas de la cohérence causale, si un processus d'application lit un objet x, et après lit un objet y, et que l'état de x dépend causalement de certaines mises à jour u de y, alors l'état y lu par le processus inclus la mise à jour u.

Quand l'application demande y, nous disons qu'il y a une **lacune causale** si la réplique locale n'a pas encore reçue u. Le système doit détecter cette lacune et attendre le moment de la livraison de u avant retourner y. Cependant, il est conseillé tout d'abord d'éviter la lacune. Autrement, la lecture dans la présence d'une lacune causale expose au risque d'anomalie les développeurs, ainsi que les utilisateurs [66, 67].

Nous considèrons une variante transactionelle de la cohèrence causale pour un meilleurs support des opérations multi-objects : tous les opérations de lecture pendant une **transaction causale** viennent du même aperçu de la base de données et soit tous les mises à jour sont visibles autant que groupe d'une façon atomique ou aucune ne l'est [14, 66, 67].

Une autre exigence constitue la **convergence**, qui est construit de deux propriétés : *(i)* **Au moins une livraison** (vivacité): une mise à jour délivrée (à savoir visible par l'application) dans un nœud, doit être délivrée dans chaque nœud (intéressé) après un nombre finis d'échange de messages ; *(ii)* **Confluence** (sûreté): deux nœuds ayant délivrer le même ensemble de mises à jour lisent la même valeur.

La cohérence causale n'est pas suffisante pour garantir la confluence, car deux répliques peuvent recevoir les mêmes mises à jour dans différents ordres. En conséquence, nous nous fions sur les objets de RDTs pour la confluence. Spécifiquement, nous utilisons une réalisation RDTs la catégorie *basée sur les opérations* [92] qui dépend des protocoles externes d'échange de logs, permettant d'utiliser un système globale de livraison des mises à jour et des protocoles de consistance pour les objets, ainsi que de partager les métadonnées. L'implémentation des objets de RDTs est facilitée par un support adéquat de la part du système. Par exemple, une valeur d'objet est définie non seulement par la dernière mise à jour, mais aussi elle dépend des mises à jour précédentes ; la consistance causale s'avère utile, car elle assure que les mises à jour ne sont

[2]Cela englobe aussi les garanties de sessions [34].

pas perdues ou délivrées hors ordre. Comme les mise à jour de RDTs de haut niveau sont souvent idempotentes (par exemple incrementation()), la sûreté exige **au maximum une livraison**.

Même si en isolation chaque exigence semble simple, la combinaison avec l'adaptabilité à un grand nombre de nœuds et à taille de la base de données constitue un nouveau défi.

C.2.1.2 La conception des métadonnées

Les métadonnées servent à identifier les mises à jour et à assurer l'exactitude. Les métadonnées sont associées aux messages de mises à jour en augmentant le coût de la communication.

Une méthode populaire de conception des métadonnées associe à chacune des mises à jour un horodatage dans le moment de la génération dans un des nœuds. La structure des métadonnées de causalité a tendance à devenir "lourde". Notamment, la liste des dépendances [66] grandi avec le nombre des mises à jour (cf. Du et al. [46], Lloyd et al. [67]), alors que les vecteurs de version [18, 69] grandissent avec le nombre de clients (en effet, nos experiments montrent que leur taille devient déraisonnable). Nous nommons cela l'approche **Attribuée Client, Sécurisé mais Gros**.

Une alternative délègue l'horodatage à un nombre de serveurs du DC [8, 46, 67]. Cela permet d'utiliser des vecteurs de plus petite taille, mais implique une perte de parallélisme. Cependant, cette configuration n'est pas tolérante aux pannes si le client réside en dehors du DC. Par exemple, la contrainte d'au maximum une livraison peut être violée. Considérons un client transmettant la mise à jour u au DC1 pour la horodatée. S'il ne reçoit pas d'acquittement, il réessaye en l'envoyant au DC2 (basculement sur erreur). En résultat, u peut recevoir deux estampilles différentes et être délivrée deux fois. La livraison dupliquée viole la contrainte de sûreté de plusieurs types d'objets de RDTs, et dans autre cas, complique l'implémentation. [5, 34, 67]. Nous nommons cela l'approche **Attribuée Serveur, Maigre mais Non Sécurisée**.

Clairement, aucune des approches, ni "gros", ni "maigre", est satisfaisantes.

C.2.1.3 La cohérence causale avec une réplication partielle est dur

Puisque une réplique partielle reçoit seulement une partie des mises à jour, ainsi qu'en conséquence des métadonnées, elle peut manquée de quelques dépendances causales [18]. Considérons l'exemple suivant : Alice télécharge une photo d'elle sur le mur d'un réseau social populaire (mise à jour a). Bob voit la photo et la mentionne dans un message destiné à Charles (mise à jour b), qui à son tour là mentionne à David (mise à jour c). Quand David regarde le mur d'Alice, il attend de trouver la mise à jour a et voir la photo. Cependant, si la machine de David ne sauvegarde pas le message de Charles dans sa mémoire cache, elle ne pourra pas observer la chaine causale $a \to b \to c$ et peut délivrer le message c sans a. La conception des métadonnées devrait protéger d'une lacune causale pareille, causé par la dépendance transitive entre des objets absents.

Cependant, les pannes compliquent la situation encore plus. Supposons que David voit la photo d'Alice et ajoute commentaire sur le mur d'Alice (mise à jour d). Maintenant, une panne se produit et la machine de David bascule vers un autre DC. Malheureusement, le nouveau DC n'a pas encore reçue le message de Bob avec la mise à jour b, de laquelle d dépend causalement. Par conséquence, le DC ne peut pas délivrer le commentaire, ainsi que respecter la convergence sans la violation de la cohérence causale. David ne pourra pas lire les nouveaux objets du DC pour la même raison.[3]

C.2.2 L'approche SwiftCloud

Nous proposons un modèle soulevant les défis mentionnés. Premièrement, nous le décrivons dans un cas sans pannes. Deuxièmement, nous traitons le cas de la panne du DC. Finalement nous introduisons une nouvelles forme de métadonnées utilisée dans notre solution.

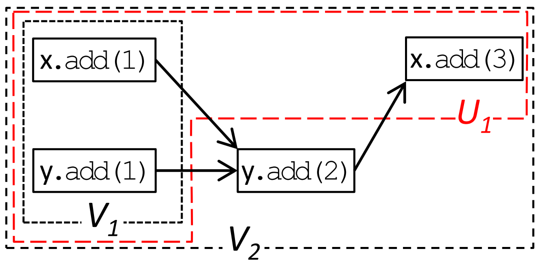
C.2.2.1 Cohèrence causale dans les répliques complètes des Centre de Données

Le problème de la garantie de la cohérence dans les DCs complétement répliqués est bien connu [4, 46, 66, 67]. Nous se concentrons sur les protocoles basées sur les logs, transmettant les opérations d'une façon incrémentale [18, 79].

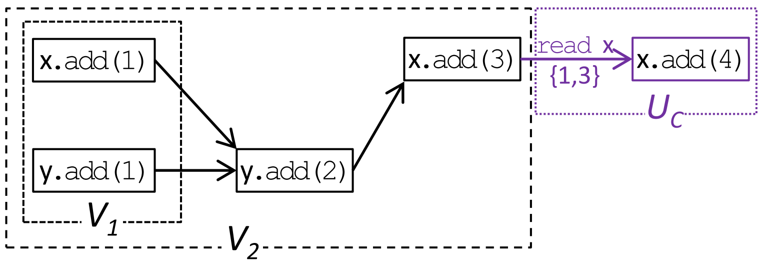
Une **version de base de données**, notée U, est un sous-ensemble des mises à jour ordonné selon la causalité. Une version associe les identifiants des objets à leurs valeurs (exposé par les méthodes `read`) en appliquant une subséquence pertinente de log au état initial de chaque objet. Nous disons qu'une version U possède une **lacune causale** ou est **incohérente** lorsqu'elle n'est pas causalement fermée, soit si $\exists \mathsf{u}, \mathsf{u}' : \mathsf{u} \to \mathsf{u}' \wedge \mathsf{u} \notin U \wedge \mathsf{u}' \in U$. Comme nous l'avons illustré, la lecture d'une version incohérente doit être évitée afin d'éviter des violation de causalité. D'autre part, l'attente de la disparition de la lacune peut augmenter la latence et diminuer la disponibilité. Pour éluder cette énigme, nous adoptons une approche de “lecture dans le passé” [4, 66]. Ainsi, le DC expose un état V, sans lacune mais retardé.

Pour illustrer, nous considérons l'exemple de Figure C.2A. Les objets x et y sont du type ensemble. DC_1 est dans l'état U_1 incluant la version $V_1 \subseteq U_1$, et DC_2 dans un état ultèrieur V_2. Les versions V_1 avec la valeur $[x \mapsto \{1\}, y \mapsto \{1\}]$ et V_2 avec la valeur $[x \mapsto \{1,3\}, y \mapsto \{1,2\}]$ ne possèdent pas de lacune.

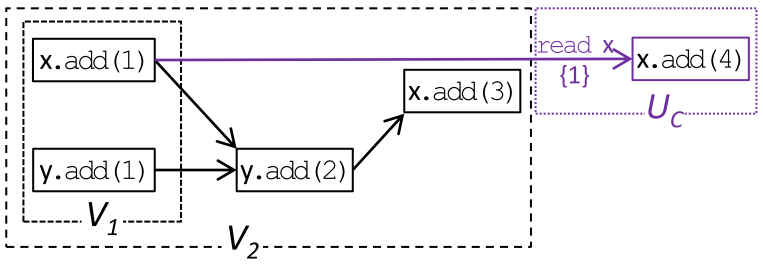
[3] Du point de vue de David, l'écriture reste disponible. Cependant, le système dans son ensemble ne converge pas.



(A) L'état initial.



(B) Continuation depuis C.2A vers un état à risque.



(C) Lecture dans le passé: continuation depuis C.2A vers un état prudent.

Figure C.2: L'exemple d'évolution d'états de deux DCs et un client. x et y sont les objets; boîte = mise à jour; flèche = dépendance causale (un texte facultatif indique la source de la dépendance); boîte pointillée = versions de bases de données nommées /les états.

Cependant, la version U_1, avec la valeur $[x \mapsto \{1,3\}, y \mapsto \{1\}]$ possède une lacune, une mise à jour manquante y.add(2). Quand le client demande de lire x au DC_1 dans l'état U_1, le DC pourrait retourner la version la plus récente, $x = \{1,3\}$. Cependant, si l'application demande ultérieurement y, pour retourner une valeur y sécurisée, il devra attendre la mise à jour manquante de la part de DC_2. En choisissant de "la lecture dans le passé", la même réplique expose la version ancienne V_1 mais sans la lacune, en lisant $x = \{1\}$. Puis, la seconde lecture sera satisfaite immédiatement avec $y = \{1\}$. Une fois la mise à jour manquante est reçue de la part du DC_2, le DC_1 peut passer de la version V_1 à la version V_2.

Un algorithme sans lacune maintient un état de réplique *causalement cohérent avec progression monotone non décroissante* [4]. Donné la mise à jour u, nous notons $u.deps$ son ensemble de prédécesseurs causales, dit son **ensemble de dépendances**. Si une réplique complète étant dans un état cohérant V, reçoit u, et si ses dépendances sont satisfaites, soit $u.deps \subseteq V$, elle applique u. Le nouveau état est $V' = V \oplus \{u\}$, où nous notons $\oplus$ un **opérateur de fusion de log**, qui filtre tous les doubles survenus pendant la transmission. L'état V' est cohérent, ainsi que son monotonie est satisfaite, car $V \subseteq V'$.

Si les dépendances ne sont pas remplies, la réplique met u dans la mémoire du tampon jusqu'au remplissement de la lacune.

C.2.2.2 Cohèrence causale dans les répliques client partielles

La cohérence se complique, quand la réplique client possède seulement une partie de la base de données et de ses métadonnées [18]. Pour éviter la complexité du protocole et pour mimiser la taille des métadonnées requis, nous nous s'appuyons sur les répliques complètes des DCs pour gérer des versions sans lacune pour le client.

Donné un **ensemble d'intérêt** des objets intéressants le client, son état initiale consiste d'une projection d'un état du DC envers l'ensemble. Celui-ci est un état causalement consistent comme démontré dans les sections précédentes.

L'état client peut change à cause d'une mise à jour générée par le client lui même, notée une **mise à jour interne**, ou à cause d'une mise à jour reçue du DC, notée **externe**. Evidement, une mise à jour interne maintient la cohérence causale. Si une mise à jour externe arrive, sans lacune, du même DC que la précédente, elle maintient aussi la cohérence causale.

Plus formellement, nous considérons un état récent du DC, que nous nommons **l'état de base** du client, noté V_{DC}. L'ensemble d'intérêt du client C est noté $O \subseteq x, y, \ldots$. L'état du client, noté V_C est limité à ces objets. Il se compose de deux parties. La première est la projection de la version de base V_{DC} sur l'ensemble d'intérêt, noté $V_{DC}|_O$. L'autre est le log des mises à jour interne, noté U_C. L'état client est leur fusion $V_C = V_{DC}|_O \oplus U_C|_O$. Dans le cas de manque de mémoire cache, le client ajoute les objets manquant à l'ensemble d'intérêt et récupère les objets à partir de la version de base V_{DC}, étendant ainsi la projection.

La version de base V_{DC} est une version monotone, non-décroissante et causale (elle peut être un peu retardée en comparaison avec la version actuelle du DC en raison des délais de propagation). Par induction, les mises à jour internes peuvent dépendre causalement uniquement des mises à jour internes ou des mises à jour prises de la version de base. Ainsi, une version hypothétique complète $V_{DC} \oplus U_C$ serrait causalement cohérente. Sa projection est équivalente au état client : $(V_{DC} \oplus U_C)|_O = V_{DC}|_O \oplus U_C|_O = V_C$.

Cette approche assure la disponibilité partielle. Si une version se trouve dans la mémoire cache, elle a la garantie d'être causalement cohérente, bien qu'elle peut être une peu viciée. Si la version ne se trouve pas dans la mémoire cache, le DC retourne immédiatement une version cohérente. De plus, la réplique client peut *écrire vite*, car elle ne doit pas attendre les mises à jour à venir (elle les transmet au DC dans la tâche de fond).

La convergence est assurée vu que la version de base du client est maintenue à jour par le DC dans la tâche de fond.

C.2.2.3 Le basculement sur erreur: Le problème avec la dépendance causale transitive

L'approche décrite assume que le client se connecte à un seul DC. Cependant, le client peut basculer vers un autre DC à tout moment, particulièrement dans une situation de panne. Bien que chaque état du DC est cohérent, une mise à jour délivrée à un DC n'est pas nécessairement délivrée aux autres (puisque la géo-réplication est asynchrone pour assurer la disponibilité et la performance au niveau de DC [15]). Cela peut créer une lacune causal chez le client.

Pour illustrer ce problème, nous retournons vers l'exemple présenté sur Figure C.2A. Considérons deux DCs: DC_1 est dans un état (cohérent) V_1, et DC_2 est dans un état (cohérent) V_2; DC_1 n'inclus pas les deux dernières mises à de V_2. Le client C, connecté au DC_2, réplique seulement l'objet x; son état est $V_2|_{\{x\}}$. Supposons que le client lit l'ensemble $x = \{1,3\}$, et effectue la mise à jour $\mathsf{u} = \mathsf{add}(4)$, en évoluant ainsi vers l'état présenté sur Figure C.2B.

Si en ce moment le client bascule vers DC_1, et que les des DCs ne peuvent pas communiquer, le système n'est pas vivant:

(1) *La lecture n'est pas disponible*: le DC_1 ne peut satisfaire une requête de y, car la version écrite par le client est plus récente que celle du DC_1, $V_2 \not\subseteq V_1$.

(2) *Les mises à jour ne peuvent pas être livrées (la divergence)*: le DC_1 ne peut pas délivrer u, en raison de manque de dépendance: $\mathsf{u}.deps \not\subseteq V_1$.

Par conséquent, DC_1 doit rejeter le client pour éviter la création d'une lacune dans l'état $V_1 \oplus U_C$.

La lecture prudente: éventuellement viciée, mais sécurisée. Pour éviter la formation des lacunes ne pouvant pas être satisfaites, l'idée est de dépendre des mises à jour probablement présentes pendant le basculement du DC, notées K**-stable**.

Une version V est K-stable si toutes ses mises à jour sont répliquées dans aux moins K DCs, soit, $|\{i \in \mathcal{DC} \mid V \subseteq V_i\}| \geq K$, où $K \geq 1$ est un seuil configuré selon les modèles d'échec. Dans ce sens, le SwiftCloud maintient une version cohérente K**-stable** $V_i^K \subseteq V_i$, contenant les mises à jour pour lesquelles le DC_i a obtenu des acquittements de la part d'au moins $K-1$ DCs différents.

La version de base du client doit être K-stable, soit, $V_C = V_i^K|_O \oplus U_C|_O$, pour supporter le basculement sur erreur.

De cette façon, le client dépend des mises à jour externe qu'il peut trouver dans presque chaque DC (V_i^K) ou des mises à jour internes qu'il peut toujours transférer vers le nouveau DC (U_C).

Pour illustrer, nous retournons vers Figure C.2A, et considérons la progression prudente vers l'état présentée sur Figure C.2C, en assumant $K = 2$. La lecture client du x retourne la 2-stable version {1} en évitant la dépendance dangereuse via une mise à jour de y. Si le DC_2 n'est pas disponible, le client peut basculer vers le DC_1, la lecture de y et la propagation de sa mise à jour restent toujours en vie.

En raison des mêmes arguments que dans Section C.2.2.2, une version de DC V_i^K est causalement cohérente et monotonement non-décroissante, d'où la version cliente aussi possède ces caratéristiques. Notons que le client observe ses mises à jour internes immédiatement, même si il n'est pas K-stable.

La discussion. La source de problème est dans la dépendance causale indirecte d'une mise à jour de laquelle les deux répliques ne sont pas conscientes (y.add(2) dans notre exemple). Comme cette question est inhérente, nous supposons un résultat général d'impossibilité, déclarant que la réplication véritablement partielle, la cohérence causale, la disponibilité partielle et la livraison au-moins-une-fois rapide (convergence) sont incompatibles. En conséquence, certaines conditions doivent être assouplies.

Remarquons que dans beaucoup de systèmes précédents, cette impossibilité impliqué un compromis entre, d'une part, la cohérence et la disponibilité et performance de l'autre [43, 66, 95]. En "lisant le passé," nous déplaçons cela vers un compromis entre la fraîcheur et la disponibilité, contrôlées par l'ajustement de K. Un K plus haut augmente la disponibilité, mais les mises à jour prennent plus de temps pour être livrées;[4] Dans une certaine limite, $K = N$ assure une complète disponibilité, mais aucun client ne peut délivrer une mises à jour si un DC n'est pas disponible. Un K inférieur améliore la fraîcheur, mais augmente la probabilité que le client ne pourra pas basculer sur erreur et qu'il attendra la récupération du DC. Dans une certaine limite, $K = 1$ est identique que le protocole de base de Section C.2.2.2, ainsi que similaire au protocoles précédents bloquant des garanties session [96].

$K = 2$ constitue un bon compromis pour le déploiement de trois ou plus DCs couvrant les scénarios commun incluant la panne du DC ou la déconnexion [43, 57]. Notre évaluation avec $K = 2$ montre qu'il encourt un manque de fraîcheur négligeable.

[4] L'augmentation du nombre des mises à jour concurrentes, que cela implique, n'est pas un problème grâce aux RDTs.

C.2.2.4 Protocoles avec les métadonnées découplées et délimitées

Grace à la communication canalisante entre DCs et la “lecture dans le passé,” notre implémentation de l’approche SwiftCloud peut utiliser un nouveau modèle de métadonnées qui découple deux aspect. Il *trace la causalité* dans le but de renforcer la cohérence, en utilisant des petits vecteurs attribués dans la tâche de fond par le DCs, et il *identifie de manière unique* les mises à jour à protéger des doublures causé par la retransmission,[5] en utilisant un horodatage scalaire assigné par le client. En conséquence, une mise à jour possède deux types d’estampilles : une estampille assignée par un simple client et une estampille assignée par un ou plus DCs. Avant d’être délivrées dans un DC, les mises à jour ne possèdent pas d’estampille ; elle posséderont une par la suite ; une mise à jour peut avoir plus qu’une estampille dans le cas d’une livraison aux plusieurs DCs (pendant le basculement, Section 7.1.3.1). Une fois l’estampille d’un DC assignée, une mise à jour peut être efficacement référencée en utilisant les vecteurs de causalité. Un vecteur de causalité associe l’identifiant du DC à un nombre naturel k, en indiquant les dépendances des k mises à jour transmises à ce DC. La taille des métadonnées causales est petite et délimitée, étant donné qu’elles possèdent une entré pour chaque DC. De plus, grace à l’extention décrite dans cette thèse, un DC peut élaguer ces logs indépendamment de la disponibilité client, en assurant la sûreté en stockant un sommaire local des mises à jour délivrées.

C.2.3 La mise en œuvre et l’évaluation

Nous implémentons le SwiftCloud et démontrons expérimentalement que notre modèle atteint ses objectifs avec un coût modeste de manque de fraîcheur. Nous évaluons le SwiftCloud dans Amazon EC2, contre un port de WaltSocial [95] et contre YCSB [42]. Quand la mémoire cache est utilisée, le temps de réponse est d’un ordre de grandeur deux fois plus faible que pour les protocoles basés sur le serveur avec des garanties de disponibilité similaires. Avec trois serveurs de DC, le système peut accueillir des milliers de répliques client. La taille des métadonnées ne dépend pas du nombre des clients, du nombre des panne ou de la taille de la base de données. Elle augmente légèrement seulement avec le nombre des DCs: en moyenne, 15 octets de métadonnées par mise à jour, avec 3 DCs, comparé aux kilooctets des algorithmes précédents aves les mêmes garanties de sûreté. Le débit est comparable à la réplication côté serveur pour une faible localité de charge, et amélioré pour une haute localité. Quand un DC tombe en panne, ses clients basculent vers un nouveau DC dans les 1000 ms, ainsi que restent cohérents. Sous conditions normales, la version 2-stable est la cause de moins d’1% de lectures viciées.

[5] Causé par les pannes réseaux, le DC, les pannes du client, le basculement sur erreur.

Bibliography

[1] Riak distributed database, 2010. URL `http://basho.com/riak/`.

[2] Kryo Java serialization library, version 2.24, 2014. URL `https://github.com/EsotericSoftware/kryo`.

[3] Data Types in Riak, October 2014. URL `http://docs.basho.com/riak/latest/theory/concepts/crdts/`.

[4] Mustaque Ahamad, James E. Burns, Phillip W. Hutto, and Gil Neiger. Causal memory. In *Proc. 5th Int. Workshop on Distributed Algorithms*, pages 9–30, Delphi, Greece, October 1991.

[5] Paulo Sérgio Almeida and Carlos Baquero. Scalable eventually consistent counters over unreliable networks. Technical Report arXiv:1307.3207, July 2013. URL `http://arxiv.org/abs/1307.3207`.

[6] Paulo Sérgio Almeida, Carlos Baquero, Ricardo Gonçalves, Nuno Preguiça, and Victor Fonte. Scalable and accurate causality tracking for eventually consistent stores. In *Int. Conf. on Distr. Apps. and Interop. Sys. (DAIS)*, Berlin, Germany, June 2014.

[7] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Efficient state-based CRDTs by delta-mutation. Technical Report arXiv:1410.2803, 2014. URL `http://arxiv.org/abs/1410.2803`.

[8] Sérgio Almeida, João Leitão, and Luís Rodrigues. ChainReaction: a causal+ consistent datastore based on Chain Replication. In *Euro. Conf. on Comp. Sys. (EuroSys)*, April 2013.

[9] Peter Alvaro, Peter Bailis, Neil Conway, and Joseph M. Hellerstein. Consistency without borders. In *Symp. on Cloud Computing (SoCC)*, Santa Clara, CA, USA, October 2013.

[10] Khaled Aslan, Pascal Molli, Hala Skaf-Molli, and Stéphane Weiss. C-Set: a Commutative Replicated Data Type for Semantic Stores. In *RED: Fourth International Workshop on REsource Discovery*, Heraklion, Greece, May 2011.

[11] Peter Bailis and Kyle Kingsbury. The network is reliable: An informal survey of real-world communications failures. *ACM Queue*, 2014.

[12] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *Int. Conf. on the Mgt. of Data (SIGMOD)*, pages 761–772, New York, NY, USA, 2013. doi: 10.1145/2463676.2465279. URL `http://doi.acm.org/10.1145/2463676.2465279`.

[13] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly Available Transactions: Virtues and limitations. In *Int. Conf. on Very Large Data Bases (VLDB)*, Riva del Garda, Trento, Italy, 2014.

[14] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Scalable atomic visibility with RAMP transactions. In *Int. Conf. on the Mgt. of Data (SIGMOD)*, 2014.

[15] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination avoidance in database systems. In *Int. Conf. on Very Large Data Bases (VLDB)*, Kohala Coast, Hawaii, 2015. To appear.

[16] Carlos Baquero and Francisco Moura. Using structural characteristics for autonomous operation. *Operating Systems Review*, 33(4):90–96, 1999. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/334598.334614.

[17] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Making operation-based CRDTs operation-based. In *Int. Conf. on Distr. Apps. and Interop. Sys. (DAIS)*, Berlin, Germany, June 2014.

[18] N Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *Networked Sys. Design and Implem. (NSDI)*, pages 59–72, San Jose, CA, USA, May 2006. Usenix, Usenix. URL `https://www.usenix.org/legacy/event/nsdi06/tech/belaramani.html`.

[19] Fabrício Benevenuto, Tiago Rodrigues, Meeyoung Cha, and Virgílio Almeida. Characterizing user behavior in online social networks. In *Internet Measurement Conference (IMC)*, 2009.

[20] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. URL `http://research.microsoft.com/pubs/ccontrol/`.

[21] Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. An optimized conflict-free replicated set. Technical Report RR-8083, Institut National de la Recherche en Informatique et Automatique (Inria), Rocquencourt, France, October 2012. URL `http://hal.inria.fr/hal-00738680`.

[22] Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. An optimized conflict-free replicated set: Additional material. Unpublished extension of the technical report [21], October 2012.

[23] Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. Brief announcement: Semantics of eventually consistent replicated sets. In *Int. Symp. on Dist. Comp. (DISC)*, volume 7611 of *Lecture Notes in Comp. Sc.*, pages 441–442, Salvador, Bahia, Brazil, October 2012. Springer-Verlag. doi: 10.1007/978-3-642-33651-5_48.

[24] Nikolaj Bjørner. Models and software model checking of a distributed file replication system. In *Formal Methods and Hybrid Real-Time Systems*, pages 1–23, 2007. URL `http://dx.doi.org/10.1007/978-3-540-75221-9_1`.

[25] Jonas Bonér. Server-managed CRDTs based on Akka. `https://github.com/jboner/akka-crdt`, 2014.

[26] Iwan Briquemont, Manuel Bravo, Zhongmiao Li, , and Peter Van Roy. Optimising client-side geo-replication with partially replicated data structures. In submission, 2014.

[27] Russell Brown, Sean Cribbs, Christopher Meiklejohn, and Sam Elliott. Riak DT map: A composable, convergent replicated dictionary. In *W. on the Principles and Practice of Eventual Consistency (PaPEC)*, Amsterdam, the Netherlands, 2014. Assoc. for Computing Machinery.

[28] Jerzy Brzeziński, Cezary Sobaniec, and Dariusz Wawrzyniak. From session causality to causal consistency. In *Euromicro Conference on Parallel, Distributed and Network based Processing*, 2004.

[29] Jerzy Brzeziński, Dariusz Dwornikowski, Łukasz Piątkowski, and Grzegorz Sobański. K-resilient session guarantees synchronization protocol for mobile ad-hoc networks. *Parallel Processing and Applied Mathematics*, 7203:30–39, 2012.

[30] Sebastian Burckhardt. Bringing TouchDevelop to the cloud. Inside Microsoft Research Blog, October 2013. URL `http://blogs.technet.com/b/inside_microsoft_research/archive/2013/10/28/bringing-touchdevelop-to-the-cloud.aspx`.

[31] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Mooly Sagiv. Eventually consistent transactions. In *Euro. Symp. on Programming (ESOP)*. Springer-Verlag, April 2012.

[32] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. Cloud types for eventual consistency. In *Euro. Conf. on Object-Oriented Pging. (ECOOP)*, pages

283–307, Berlin, Heidelberg, 2012. Springer-Verlag. doi: 10.1007/978-3-642-31057-7_14. URL `http://dx.doi.org/10.1007/978-3-642-31057-7_14`.

[33] Sebastian Burckhardt, Alexey Gotsman, and Hongseok Yang. Understanding eventual consistency. Technical Report MSR-TR-2013-39, Microsoft Research, Redmond, WA, USA, March 2013. URL `http://research.microsoft.com/apps/pubs/?id=189249`.

[34] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: Specification, verification, optimality. In *Symp. on Principles of Prog. Lang. (POPL)*, pages 271–284, San Diego, CA, USA, January 2014. doi: 10.1145/2535838.2535848. URL `http://doi.acm.org/10.1145/2535838.2535848`.

[35] Sebastian Burckhardt, Daan Leijen, and Manuel Fahndrich. Cloud types: Robust abstractions for replicated shared state. Technical Report MSR-TR-2014-43, Microsoft Research, Redmond, WA, USA, March 2014. URL `http://research.microsoft.com/apps/pubs/default.aspx?id=211340`.

[36] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer Publishing Company, Incorporated, 2nd edition, 2011. ISBN 3642152597, 9783642152597.

[37] Brian Cairns. Build collaborative apps with Google Drive Realtime API. Google Apps Developers Blog, March 2013. URL `http://googleappsdeveloper.blogspot.com/2013/03/build-collaborative-apps-with-google.html`.

[38] Baquero Carlos. Scaling up reconciliation in eventual consistency, November 2012. URL `http://haslab.wordpress.com/2012/11/28/scaling-up-reconciliation-in-eventual-consistency/`.

[39] Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39(1), 1991.

[40] Byung-Gon Chun, Carlo Curino, Russell Sears, Alexander Shraer, Samuel Madden, and Raghu Ramakrishnan. Mobius: Unified messaging and data serving for mobile apps. In *Int. Conf. on Mobile Sys., Apps. and Services (MobiSys)*, pages 141–154, New York, NY, USA, 2012.

[41] Neil Conway, Peter Alvaro, Emily Andrews, and Joseph M Hellerstein. Edelweiss: Automatic storage reclamation for distributed programming. *Proc. VLDB Endow.*, 7(6), 2014.

[42] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Symp. on Cloud Computing (SoCC)*, pages 143–154, Indianapolis, IN, USA, 2010.

[43] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, pages 251–264, Hollywood, CA, USA, October 2012. Usenix. URL `https://www.usenix.org/system/files/conference/osdi12/osdi12-final-16.pdf`.

[44] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Symp. on Op. Sys. Principles (SOSP)*, volume 41 of *Operating Systems Review*, pages 205–220, Stevenson, Washington, USA, October 2007. Assoc. for Computing Machinery. doi: http://doi.acm.org/10.1145/1294261.1294281.

[45] Andrei Deftu and Jan Griebsch. A scalable conflict-free replicated set data type. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*, pages 186–195, Washington, DC, USA, 2013.

[46] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Symp. on Cloud Computing (SoCC)*, pages 11:1–11:14, Santa Clara, CA, USA, October 2013. Assoc. for Computing Machinery. doi: 10.1145/2523616.2523628. URL `http://doi.acm.org/10.1145/2523616.2523628`.

[47] Jiaqing Du, Calin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. Closing the performance gap between causal consistency and eventual consistency,. In *W. on the Principles and Practice of Eventual Consistency (PaPEC)*, Amsterdam, the Netherlands, 2014. URL `http://eventos.fct.unl.pt/papec/pages/program`.

[48] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Int. Conf. on the Mgt. of Data (SIGMOD)*, pages 399–407, Portland, OR, USA, 1989. Assoc. for Computing Machinery. doi: http://doi.acm.org/10.1145/67544.66963.

[49] Faith Fich and Eric Ruppert. Hundreds of impossibility results for distributed computing. *Distrib. Comput.*, 16(2-3):121–163, September 2003. doi: 10.1007/s00446-003-0091-y. URL `http://dx.doi.org/10.1007/s00446-003-0091-y`.

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

[51] Richard A. Golding. *Weak-consistency group communication and membership*. PhD thesis, University of California Santa Cruz, Santa Cruz, CA, USA, December 1992. URL `ftp://ftp.cse.ucsc.edu/pub/tr/ucsc-crl-92-52.ps.Z`. Tech. Report UCSC-CRL-92-52.

[52] Alexey Gotsman and Hongseok Yang. Composite replicated data types. In submission, 2014. URL `http://software.imdea.org/~gotsman/papers/compos.pdf`.

[53] Maurice Herlihy and Jeannette Wing. Linearizability: a correcteness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990. URL `http://doi.acm.org/10.1145/78969.78972`.

[54] Martin Hofmann, Benjamin Pierce, and Daniel Wagner. Symmetric lenses. In *Symp. on Principles of Prog. Lang. (POPL)*, pages 371–384, New York, NY, USA, 2011. Assoc. for Computing Machinery.

[55] Caroline Jay, Mashhuda Glencross, and Roger Hubbold. Modeling the effects of delayed haptic and visual feedback in a collaborative virtual environment. *ACM Trans. Comput.-Hum. Interact.*, 14(2), August 2007. doi: 10.1145/1275511.1275514. URL `http://doi.acm.org/10.1145/1275511.1275514`.

[56] Paul R. Johnson and Robert H. Thomas. The maintenance of duplicate databases. Internet Request for Comments RFC 677, Information Sciences Institute, January 1976. URL `http://www.rfc-editor.org/rfc.html`.

[57] Aman Kansal, Bhuvan Urgaonkar, and Sriram Govindan. Using dark fiber to displace diesel generators. In *Hot Topics in Operating Systems*, Santa Ana Pueblo, NM, USA, 2013.

[58] Kyle Kingsbury. Call me maybe: Cassandra. `http://aphyr.com/posts/294-call-me-maybe-cassandra/`, September 2013.

[59] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Trans. on Comp. Sys. (TOCS)*, 10(5):3–25, February 1992. URL `http://www.acm.org/pubs/contents/journals/tocs/1992-10`.

[60] Robert Kroeger. Gmail for mobile HTML5 series: Cache pattern for offline HTML5 web applications. Google Code Blog, June 2009. URL `http://googlecode.blogspot.com/2009/06/gmail-for-mobile-html5-series-cache.html`.

[61] Rivka Ladin, Barbara Liskov, and Liuba Shrira. Lazy replication: Exploiting the semantics of distributed services. *Operating Systems Review*, 25(1):49–55, January 1991.

[62] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *Trans. on Computer Systems*, 10(4):360–391, November 1992. URL `http://dx.doi.org/10.1145/138873.138877`.

[63] Avinash Lakshman and Prashant Malik. Cassandra, a decentralized structured storage system. In *W. on Large-Scale Dist. Sys. and Middleware (LADIS)*, volume 44 of *Operating Systems Review*, pages 35–40, Big Sky, MT, USA, October 2009. ACM SIG on Op. Sys. (SIGOPS), Assoc. for Computing Machinery. doi: http://dx.doi.org/10.1145/1773912.1773922.

[64] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978. URL `http://doi.acm.org/10.1145/359545.359563`.

[65] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, pages 265–278, Hollywood, CA, USA, October 2012.

[66] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Symp. on Op. Sys. Principles (SOSP)*, pages 401–416, Cascais, Portugal, October 2011. Assoc. for Computing Machinery. doi: http://doi.acm.org/10.1145/2043556.2043593.

[67] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Networked Sys. Design and Implem. (NSDI)*, pages 313–328, Lombard, IL, USA, April 2013. Usenix. URL `https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final149.pdf`.

[68] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. Consistency, availability, and convergence. Technical Report UTCS TR-11-22, Dept. of Comp. Sc., The U. of Texas at Austin, Austin, TX, USA, 2011.

[69] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. *Trans. on Computer Systems*, 29(4):12:1–12:38, December 2011. doi: 10.1145/2063509.2063512. URL `http://doi.acm.org/10.1145/2063509.2063512`.

[70] Dahlia Malkhi and Douglas B. Terry. Concise version vectors in WinFS. *Distributed Computing*, 20(3):209–219, October 2007. doi: http://dx.doi.org/10.1007/s00446-007-0044-y.

[71] Friedmann Mattern. Virtual time and global states of distributed systems. In *Int. W. on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B.V. (North-Holland), 1989.

[72] Madhavan Mukund, Gautham Shenoy, and S. P. Suresh. Optimized OR-sets without ordering constraints. In *Int. Conf. on Distributed Comp. and Net. (ICDCN)*, pages 227–241. Springer, 2014.

[73] Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. LSEQ: An adaptive structure for sequences in distributed collaborative editing. In *Proceedings of the ACM Symposium on Document Engineering*, DocEng '13, pages 37–46, New York, NY, USA, 2013. Assoc. for Computing Machinery. URL `http://doi.acm.org/10.1145/2494266.2494278`.

[74] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Proving correctness of transformation functions in collaborative editing systems. Technical Report RR-5795, LORIA – INRIA Lorraine, December 2005. URL `http://hal.inria.fr/inria-00071213/`.

[75] Gérald Oster, Pascal Molli, Pascal Urso., and Abdessamad Imine. Tombstone Transformation Functions for ensuring consistency in collaborative editing systems. In *Int. Conf. on Coll. Computing: Networking, Apps. and Worksharing (CollaborateCom)*, November 2006.

[76] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data consistency for P2P collaborative editing. In *Int. Conf. on Computer-Supported Coop. Work (CSCW)*, pages 259–268, Banff, Alberta, Canada, November 2006. ACM Press. doi: http://doi.acm.org/10.1145/1180875.1180916. URL `http://www.loria.fr/~molli/pmwiki/uploads/Main/oster06cscw.pdf`.

[77] D. Stott Parker, Jr., Gerald J. Popek, Gerald Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. on Soft. Engin.*, SE-9(3):240–247, May 1983.

[78] José Pereira, Luís Rodrigues, and Rui Oliveira. Semantically reliable multicast: Definition, implementation, and performance evaluation. *IEEE Trans. on Computers*, 52(2):150–165, February 2003. ISSN 0018-9340. doi: 10.1109/TC.2003.1176983. URL `http://dx.doi.org/10.1109/TC.2003.1176983`.

[79] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Symp. on Op. Sys. Principles (SOSP)*, pages 288–301, Saint Malo, October 1997. ACM SIGOPS. URL `http://doi.acm.org/10.1145/268998.266711`.

[80] Gerald J. Popek, Richard G. Guy, Thomas W. Page, Jr., and John S. Heidemann. Replication in Ficus distributed file systems. In *IEEE Computer Society Technical Committee on Operating Systems and Application Environments Newsletter*, volume 4, pages 24–29. IEEE, IEEE Computer Society, 1990.

[81] Nuno Preguiça, Joan Manuel Marquès, Marc Shapiro, and Mihai Leţia. A commutative replicated data type for cooperative editing. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*,

pages 395–403, Montréal, Canada, June 2009. doi: http://doi.ieeecomputersociety.org/10.1109/ICDCS.2009.20. URL http://lip6.fr/Marc.Shapiro/papers/icdcs09-treedoc.pdf.

[82] Nuno Preguiça, J. Legatheaux Martins, Henrique Domingos, and Sérgio Duarte. Data management support for asynchronous groupware. In *Int. Conf. on Computer-Supported Coop. Work (CSCW)*, pages 69–78, New York, NY, USA, 2000. Assoc. for Computing Machinery. ISBN 1-58113-222-0. doi: 10.1145/358916.358972. URL http://doi.acm.org/10.1145/358916.358972.

[83] Venugopalan Ramasubramanian, Thomas Rodeheffer, Douglas B. Terry, Meg Walraed-Sullivan, Ted Wobber, Cathy Marshall, and Amin Vahdat. Cimbiosys: A platform for content-based partial replication. In *Networked Sys. Design and Implem. (NSDI)*, 2009.

[84] Redis. Redis is an open source, BSD licensed, advanced key-value store. http://redis.io/, May 2014.

[85] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated Abstract Data Types: Building blocks for collaborative applications. *Journal of Parallel and Dist. Comp.*, 71(3):354–368, March 2011. doi: http://dx.doi.org/10.1016/j.jpdc.2010.12.006.

[86] Masoud Saeida Ardekani, Marek Zawirski, Pierre Sutra, and Marc Shapiro. The space complexity of transactional interactive reads. In *Int. W. on Hot Topics in Cloud Data Processing (HotCDP)*, Bern, Switzerland, April 2012. Assoc. for Computing Machinery. doi: 10.1145/2169090.2169094. URL http://doi.acm.org/10.1145/2169090.2169094.

[87] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. Non-Monotonic Snapshot Isolation: scalable and strong consistency for geo-replicated transactional systems. In *Symp. on Reliable Dist. Sys. (SRDS)*, pages 163–172, Braga, Portugal, October 2013. IEEE Comp. Society. doi: 10.1109/SRDS.2013.25. URL http://lip6.fr/Marc.Shapiro/papers/NMSI-SRDS-2013.pdf.

[88] Nicolas Schiper, Pierre Sutra, and Fernando Pedone. P-Store: Genuine partial replication in wide area networks. In *Symp. on Reliable Dist. Sys. (SRDS)*, pages 214–224, New Dehli, India, October 2010. IEEE Comp. Society. URL http://doi.ieeecomputersociety.org/10.1109/SRDS.2010.32.

[89] E. Schurman and J. Brutlag. Performance related changes and their user impact, June 2009. URL https://www.youtube.com/watch?v=bQSE51-gr2s. Velocity Web Performance and Operations Conference.

[90] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994. ISSN 0178-2770. doi: 10.1007/BF02277859. URL http://dx.doi.org/10.1007/BF02277859.

[91] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Convergent and commutative replicated data types. Technical Report 104, June 2011. URL `http://www.eatcs.org/images/bulletin/beatcs104.pdf`.

[92] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 6976 of *Lecture Notes in Comp. Sc.*, pages 386–400, Grenoble, France, October 2011. Springer-Verlag. doi: 10.1007/978-3-642-24550-3_29. URL `http://www.springerlink.com/content/3rg39l2287330370/`.

[93] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report RR-7506, Institut National de la Recherche en Informatique et Automatique (Inria), Rocquencourt, France, January 2011. URL `http://hal.archives-ouvertes.fr/inria-00555588/`.

[94] Craig Shoemaker. Build an HTML5 offline application with Application Cache, Web Storage and ASP.NET MVC. CODE Magazine, 2013. URL `http://www.codemag.com/Article/1112051`.

[95] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Symp. on Op. Sys. Principles (SOSP)*, pages 385–400, Cascais, Portugal, October 2011. Assoc. for Computing Machinery. doi: http://doi.acm.org/10.1145/2043556.2043592.

[96] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *Int. Conf. on Para. and Dist. Info. Sys. (PDIS)*, pages 140–149, Austin, Texas, USA, September 1994.

[97] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Symp. on Op. Sys. Principles (SOSP)*, pages 172–182, Copper Mountain, CO, USA, December 1995. ACM SIGOPS, ACM Press. URL `http://www.acm.org/pubs/articles/proceedings/ops/224056/p172-terry/p172-terry.pdf`.

[98] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, pages 91–104, San Francisco, CA, USA, December 2004. Usenix, Usenix. URL `http://www.usenix.org/events/osdi04/tech/renesse.html`.

[99] Kaushik Veeraraghavan, Venugopalan Ramasubramanian, Thomas L. Rodeheffer, Douglas B. Terry, and Ted Wobber. Fidelity-aware replication for mobile devices. In *Int. Conf. on Mobile Sys., Apps. and Services (MobiSys)*. Assoc. for Computing Machinery, June 2009. URL `http://research.microsoft.com/apps/pubs/default.aspx?id=80670`.

[100] Werner Vogels. Eventually consistent. *ACM Queue*, 6(6):14–19, October 2008. doi: http://doi.acm.org/10.1145/1466443.x.

[101] Stéphane Weiss, Pascal Urso, and Pascal Molli. Logoot: a scalable optimistic replication algorithm for collaborative editing on P2P networks. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*, Montréal, Canada, June 2009.

[102] Gene T. J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Symp. on Principles of Dist. Comp. (PODC)*, pages 233–242, Vancouver, BC, Canada, August 1984. URL `http://doi.acm.org/10.1145/800222.806750`.

[103] Marek Zawirski, Marc Shapiro, and Nuno Preguiça. Asynchronous rebalancing of a replicated tree. In *Conf. Française sur les Systèmes d'Exploitation (CFSE)*, Saint-Malo, France, May 2011.

[104] Marek Zawirski, Annette Bieniusa, Valter Balegas, Sérgio Duarte, Carlos Baquero, Marc Shapiro, and Nuno Preguiça. SwiftCloud: Fault-tolerant geo-replication integrated all the way to the client machine. Technical Report RR-8347, Institut National de la Recherche en Informatique et Automatique (Inria), Rocquencourt, France, August 2013. URL `http://hal.inria.fr/hal-00870225/`.