Building Fast and Consistent (Geo-)Replicated Systems: from Principles to Practice

Thesis for obtaining the title of Doctor of Engineering of the Faculty of Natural Science and Technology I of Saarland University

By

Cheng Li

Saarbrücken

2016

Date of Colloquium:	May 30, 2016
Dean of Faculty:	Prof. Dr. Frank-Olaf Schreyer
Chair of the Committee:	Prof. Dr. Holger Hermanns
Reporters	
First Reviewer:	Prof. Dr. Rodrigo Rodrigues
Second Reviewer:	Prof. Dr. Peter Druschel
Third Reviewer:	Prof. Dr. Robbert van Renesse
Academic Assistant:	Dr. Rijurekha Sen

Abstract

Distributing data across replicas within a data center or across multiple data centers plays an important role in building Internet-scale services that provide a good user experience, namely low latency access and high throughput. This approach often compromises on strong consistency semantics, which helps maintain application-specific desired properties, namely, state convergence and invariant preservation. To relieve such inherent tension, in the past few years, many proposals have been designed to allow programmers to selectively weaken consistency levels of certain operations to avoid costly immediate coordination for concurrent user requests. However, these fail to provide principles to guide programmers to make a correct decision of assigning consistency levels to various operations so that good performance is extracted while the system behavior still complies with its specification.

The primary goal of this thesis work is to provide programmers with principles and tools for building fast and consistent (geo-)replicated systems by allowing programmers to think about various consistency levels in the same framework. The first step we took was to propose RedBlue consistency, which presents sufficient conditions that allow programmers to safely separate weakly consistent operations from strongly consistent ones in a coarse-grained manner. Second, to improve the practicality of RedBlue consistency, we built SIEVE - a tool that explores both Commutative Replicated Data Types and program analysis techniques to assign proper consistency levels to different operations and to maximize the weakly consistent operation space. Finally, we generalized the tradeoff between consistency and performance and proposed Partial Order-Restrictions consistency (or short, PoR consistency) - a generic consistency definition that captures various consistency levels in terms of visibility restrictions among pairs of operations and allows programmers to tune the restrictions to obtain a fine-grained control of their targeted consistency semantics.

Kurzdarstellung

Daten auf mehrere Repliken in einem Datenzentrum oder über mehrere Datenzentren zu verteilen, nimmt einen hohen Stellenwert ein, um Internet-weite Services mit guter Nutzererfahrung, insbesondere mit niedrigen Zugriffszeiten und hohem Datendurchsatz, zu implementieren. Diese Methode beeinträchtigt in der Regel die starke Konsitenzsemantik, die hilft gewünschte anwendungsspezifische Eigenschaften, die Zustandskonvergenz und Erhaltung von Invarianten, aufrechtzuerhalten. Um diesen Kompromiss zu mildern, wurde in den letzten Jahren mehrere Vorschläge entworfen, die es dem Programmierer ermöglichen für einzelne Operationen ein schwächeres Konsitenzlevel auszuwählen, um der aufwendigen Koordination paralleler Benutzeranfragen zu entgehen. Allerdings liefern diese Leitsätze für die Programmierer keine Lösungsansätze, wann welches Konsistenzlevel für eine Operation anzuwenden ist, so dass die höchstmögliche Leistung erreicht wird und gleichzeitig die Handlung des Systems die Spezifikation erfüllen.

Das Hauptziel dieser Doktorarbeit ist es Leitsätzen und Werkzeuge für Programmierer bereitzustellen, die die Entwicklung von leistungsstarken, konsistenten und (weltweit) replizierten Sytemen ermöglichen, in dem dem Programmierer mit Hilfe eines Frameworks gleichzeitig zwischen verschiedenen Konsistenzlevel wählen kann. Als ersten Schritt entwickelten wir RedBlue Konsistenz, welches die hinreichende Bedingungen erläutert, die es einem Programmierer erlauben zwischen schwacher Konsistenz und starker Konsistenz zu wählen. Um die Praktikabilität von RedBlue Konsistenz im zweiten Schritt weiter zu erhöhen, entwickelten wir SIEVE - ein Werkzeug, das sowohl kommutative, replizierte Datentypen und Programmanalyseverfahren verwendet, um den richtigen Konsistenzlevel zu verschiedenen Operationen zuzuordnen und dabei die schwach konsistenten Operationen zu maximieren. Abschliessend verallgemeinern wir den Kompromiss zwischen Konsistenz und Leistungsstärke und stellen die partiell, eingeschränkt geordnete Konsistenz vor (PoR Konsistenz) - eine generische Konsistenzdefinition, die

verschiedene Konsistenz level, hinsichtlich der Einschränkung der Sichtbarkeit zwischen paaren von Operationen, umfasst und dem Programmierer erlaubt, die Einschränkungen zu justieren, um die gewünschte Konsistenzsemantik zu erzielen.

Parts of the thesis have appeared in the following publications.

- *Geo-Replication: Fast If Possible, Consistent If Necessary.*

  Valter Balegas, Cheng Li, Mahsa Najafzadeh, Daniel Porto, Allen Clement, Sergio Duarte, Carla Ferreira, Johannes Gehrke, João Leitão, Nuno Preguiça, Rodrigo Rodrigues Marc Shapiro and Viktor Vafeiadis.

  Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 2016

- *Minimizing Coordination in Replicated Systems.*

  Cheng Li, João Leitão, Allen Clement, Nuno Preguiça, and Rodrigo Rodrigues.

  PaPoC'15, Apr. 2015, Bordeaux, France

- *Automating the Choice of Consistency Levels in Replicated Systems.*

  Cheng Li, João Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues and Viktor Vafeiadis.

  USENIX ATC'14, Jun. 2014, Philadelphia, PA, USA

- *Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary.*

  Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues.

  OSDI'12, Oct. 2012, Hollywood, CA, USA

- *Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary.*

  Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues.

  Technical Report, MPI-SWS, 2012

Additional publications published while at MPI-SWS.

- *Visigoth Fault tolerance.*

  Daniel Porto, João Leitão, Cheng Li, Allen Clement, Aniket Kate, Flavio Junqueira, and Rodrigo Rodrigues.

  EuroSys'15, Apr. 2015, Bordeaux, France

- *Lower Bound and Correctness Proofs for Consensus in the Visigoth Model.*

  Daniel Porto, João Leitão, Cheng Li, Allen Clement, Aniket Kate, Flavio Junqueira, and Rodrigo Rodrigues.

  Technical Report, Nova University of Lisbon, 2015

- *Finding Complex Concurrency Bugs in Large Multi-Threaded Applications.*

  Pedro Fonseca, Cheng Li, and Rodrigo Rodrigues.

  EuroSys'11, Apr. 2011, Salzburg, Austria

- *A study of the Internal and External Effects of Concurrency Bugs.*

  Pedro Fonseca, Cheng Li, Vishal Singhal and Rodrigo Rodrigues.

  DSN'10, Jun. 2010, Chicago, USA

Dedicated to my mummy and papa.

Contents

List of Figures

List of Tables

1 Introduction

1.1 The unprecedented popularity of Internet services

In the last decade, Internet services, such as web search, email, collaborative editing, e-commerce and social networking, have become increasingly popular at an unprecedented speed. Nowadays, at any point of time, a massive number of subscribers are interacting with these services. For instance, an annual report [FB213] shows that Facebook had on average 757 million daily active users worldwide in December 2013, which is 22% larger than the number measured a year ago. In December 2012, Google [Gooa] received 114.7 billion monthly searches, while Bing [bin] got 4.5 billion [Sul13]. Likewise, Amazon [Amab], a leading worldwide e-commerce retailer company, reported that their number of active customers per year has been remarkably increasing since 1997 and reached 270 million in 2014 [sta].

The direct implication of this trend is that the scalability of these services must withstand the tremendous scale of requests issued by users all over the world. In 2010, over one million images per second are served by Facebook when the peak load arrives [BKL+10]. In May 2013, Google reported that the High Replication Datastore integrated with Google App Engine was able to process over 4.5 trillion transactions per month [Goob]. Another important property in addition to high throughput that these services must offer is low latency access, since the amount of time a user spent waiting for responses has a strong negative impact on their subsequent behavior [Web, Lin06, SB09]. In a recent study [SB09] conducted at Microsoft, engineers measured the impact of delaying

the reply to user requests, and discovered an inverse correlation between response times and user satisfaction. For instance, 4.4% fewer users performed clicks and the revenue per user decreased by 4.3% when the delay reaches 2 seconds. Incidentally, Google made a very similar experiment and obtained consistent results [SB09]. For example, the average number of daily searches per user dropped with an increasing delay, namely by 0.59% when a 400-ms delay was artificially added.

1.2 The case for (geo-)replication

In order to achieve high service throughput and to offer fast responses to users, the providers of Internet services, such as Google [Gooa], Microsoft [Mic], Amazon [Amab] and Facebook [Fac], replicate their user state across replicas within a single data center or across multiple data centers that are either at the same location or geographically scattered over continents. User requests then are forwarded to a nearby or the least loaded replica. A few representative systems that provide geo-replication are as follows: (1) Engineers at Facebook designed TAO, a geographically distributed system, to store user data across data centers and geographic regions [BAC+13]. They claimed that read latency in TAO is independent of inter-region latency, which is often a few orders of magnitude higher than the intra-region or intra-data center latency. (2) Most Google applications satisfy their interactivity demand through adopting geo-distributed data stores like Spanner [CDE+12], Megastore [BBC+11] and Mesa [GYG+14]. (3) Pileus [TPK+13] at Microsoft is a replicated key-value store that allows applications to express different latency requirements by specifying which set of servers to contact for executing operations.

1.3 Fundamental tradeoff: consistency v. performance

While replicating user data, some form of synchronization is required to bring all copies up-to-date. The timing of this synchronization reflects the inherent tension between performance and the desired consistency semantics. On the one hand, to avoid paying the performance cost of coordinating concurrent user requests across replicas, some systems, such as Amazon's Dynamo [DHJ+07], resort to weaker consistency semantics like eventual consistency [BGY13, SS05, Vog09], under which only a small number of replicas will be contacted to produce a user response event, and later, in the background, the corresponding side effects are lazily replicated across all other replicas. This technique is favored by latency-sensitive services, such as instant messenger, social networking and online shopping, since it offers low latency access by eliminating immediate coordination. The downside, however, is that it introduces difficulties for programming applications, as it offers semantics differing from the natural semantics specified by Linearizability [HW90], where a replicated service involving multiple machines behaves as a single centralized server. In particular, weak consistency semantics require programmers to make an effort to reason about the correctness of their implementation and to handle unexpected behaviors, such as invariant violations or state divergence.

On the other hand, to avoid the above difficulties, some systems like Spanner [CDE+12] choose strong consistency [HW90], where coordination among replicas is required for them to agree on the order in which user requests are executed. This coordination, however, incurs in high latency, and the penalty will be amplified in geo-replication scenarios as the communication cost across world regions is two or three orders of magnitude larger than the one measured within a data center [LPC+12, TPK+13].

1.4 Challenges for being fast

To scale out the Internet services to meet their ever-growing user base, many recent storage systems have been proposed to replicate operations following a hybrid (or, multi-level) consistency model, where some operations can be executed optimistically at a replica without being coordinated with concurrent actions at other replicas, while others require a stronger consistency level and thus require cross-replica coordination [LLSG92, SFK+09, LPC+12, TPK+13]. Although these proposals have validated that associating operations with different consistency levels is a promising solution for building highly scalable Internet services, there are still a few challenges impeding their adoption in practice. The target of my thesis is to explore reasonable solutions to mitigate these fundamental issues.

First, **how to find conditions that guide the use of weak consistency in multi-level consistency schemes?** Weakly consistent operations are (dramatically) faster than strongly consistent ones, since fewer (geo-distributed) replicas need to be contacted. However, we cannot arbitrarily label operations weakly consistent, since an over-optimistic labeling plan might break the desired application-specific properties, e.g., two concurrent withdrawals without coordination potentially drive a shared bank balance below zero. Therefore, to safely use weak consistency, we must extract a set of sufficient conditions to guide the classification. In addition, the degree of performance improvement highly relies on the ratio of weakly consistent operations to the strongly consistent ones in a replicated service. In some applications, we observed that not many operations could accept weak consistency. To address this limitation, we need to explore a way for transforming operations so that the space of weakly consistent operations significantly increases.

Second, **can we provide tools that automate the above decision process?** The problem with multi-level consistency solutions [LLSG92, SFK+09, LPC+12, TPK+13] is two-fold: a) they impose on the application programmer the non-trivial burden of un-

derstanding the semantics of operations and the influence between operations associated with different consistency levels even if guided by sufficient conditions from addressing the previous point [LPC+12]; and b) in order to handle conflicts, they require programmers manually to adopt a new programming model to write their services from scratch, or to patch their services with newly implemented merge procedures. In summary, automation is required to make multi-level consistency models easy-of-use.

Third, **can we maximize performance by having a generic way to express various consistency requirements and leveraging this to minimize the amount of required coordination?** We observe that in some systems, in order to avoid an undesirable system behavior (e.g., state divergence or invariant violation), the adoption of multi-level consistency models (e.g., RedBlue consistency) introduces unnecessary coordination. This is because an operation not accepting weak consistency semantics only has to be coordinated w.r.t a particular group of operations, instead of all operations requiring stronger consistency semantics. Unfortunately, however, multi-level consistency models often do not allow us to have such a fine-grained tuning in consistency requirements, and hence they do not always guarantee that the amount of coordination imposed for a replicated service atop of them is minimal. To address this limitation, we need to find a generic consistency definition, which provides programmers with flexibility to express various fine-grained consistency semantics in a single framework.

1.5 Thesis contributions

In this thesis, we aimed to provide right answers to the above three questions. At a high level, we made the following contributions:

- We proposed a novel consistency definition called RedBlue consistency [LPC+12], which allows operations to run under either weak or strong consistency. We also extracted a set of principles to guide programmers to make the decision of associat-

ing operations with different consistency levels. In essence, we only label operations as strongly consistent if they are not commuting with all other operations or they might potentially break invariants, and the remaining operations are weakly consistent. In addition to the classification methodology, we realize that the performance benefits only become visible if weakly consistent operations dominate the operation space. However, given the fact that the vast majority operations in the examples we studied do not commute, we will end up labeling more operations strongly consistent. To address this limitation, we further proposed a concept called shadow operation, which helps replicate operations in a commutative manner regardless of when and where the replication takes place.

- We designed an automatic tool, SIEVE [LLaC+14], to free programmers from the time-consuming and possibly error-prone tasks of manually choosing appropriate consistency levels for various operations, while requiring a minimal amount of programmer input. To achieve this, SIEVE first leverages Commutative Replicated Data Types (CRDTs) [LPS09] to automatically create commutative shadow operations, by only requiring a small amount of annotations from programmers to specify which CRDT to be used. Using CRDTs, the remaining challenge to assign consistency levels is how to efficiently check whether a generated shadow operation will potentially break programmer-specified invariants under weakly consistent replication. To overcome this, SIEVE uses both static analysis and runtime verification, and accomplishes most of work the offline, to offer a low-cost dynamic consistency level assignment.

- To enable a flexible way to express fine-grained consistency requirements, we proposed a generic consistency definition, Partial Order-Restrictions consistency (or short, PoR consistency), which is not only able to express several existing consistency levels in a uniform fashion, but is also able to express more levels [LLaC+15]

not covered in multi-level consistency models. Basically, PoR consistency captures consistency levels in terms of visibility restrictions over pairs of operations in a partial order. Weakening or strengthening consistency semantics in the context of PoR consistency is to remove or add restrictions over pairs of relevant operations. To demonstrate the benefits of adopting PoR consistency, we designed and implemented an efficient coordination service, which executes operations complying with pre-defined restrictions among operations of a replicated service and is able to further reduce the cost of coordination by taking into account the runtime operation frequencies.

1.6 Thesis organization

In summary, the primary goal of my thesis is to help programmers make their replicated services as fast as possible and pay the coordination cost only when needed. The rest of the thesis is organized as follows:

- Chapter 2 presents the system model and properties.
- Chapter 3 presents the formalization of RedBlue consistency, and the design, implementation, and evaluation of Gemini, a geo-distributed data store supporting RedBlue consistency.
- Chapter 4 presents the design, implementation and evaluation of SIEVE, a tool for adapting applications to RedBlue consistency, i.e., automatically assigning appropriate consistency levels to various operations.
- Chapter 5 presents the formalization of PoR consistency, and the design, implementation and evaluation of Olisipo, an efficient coordination service replicating operations by following rules defined in PoR consistency.
- Finally, Chapter 6 concludes my thesis.

2 System model

In this chapter, we present the system model our work is built atop and a set of desirable end-to-end system properties (marked in bold) we intend to provide.

We align our notations with those defined in the well-known state machine replication literature [Sch90]. We assume a distributed system with state fully replicated across k sites denoted by $site_0 \dots site_{k-1}$. Each site hosts a replica, and each replica behaves following a deterministic state machine model. It is worth mentioning that a site is a logical unit that hosts a full copy of system state, and hence, it is possible to have multiple sites across geographically dispersed data centers or even within a single data center. In the rest of the document, the terms "site" and "replica" are interchangeable.

The system defines a set of operations $\mathcal{U}$ manipulating a set of reachable states $\mathcal{S}$. We do not restrict the type of operations that can be executed within that system, a property we call **general operations**. If operation u is applied against a system state S, it produces another system state S'; we will also denote this by $S' = S + u$. Given a total order $T(U, <)$ over a set of operations U, where $U \subset \mathcal{U}$, if we sequentially apply all operations U against a system state S according to $<$, then we denote the final state by $S(T)$. $S(T) = S + u_0 + u_1 + ... + u_i + ... + u_{|U|-1}$, where $0 \leq i < |U|$. We say that a pair of operations u and v *commute* if $\forall S \in \mathcal{S}, S + u + v = S + v + u$. An operation u is *globally commutative*, if it commutes with all operations in $\mathcal{U}$ (including itself).

Each operation u is initially submitted by a client at one site which we call u's *primary site* and denote $site(u)$; the system then later replicates u to all remaining sites. Upon

receiving an operation, replicas at the recipient sites apply it against their local state. It is important that all replicas that have executed the same set of operations are in the same state, i.e., that the underlying system offers a **state convergence** property; otherwise, a quiescent system would return different views of the state depending on which replicas the users connected to.

As clients are always expecting fast responses to their requests, we aim to provide **low latency** access to the service [SB09]. Another important property that consists of ensuring a good user experience is to preserve **causality**, both in terms of the monotonicity of user requests within a session and preserving causality across clients, which is key to enabling natural semantics [PST+97]. Additionally, operations invoked by client requests should return a **single value**, precluding solutions that return a set of values corresponding to the outcome of multiple concurrent updates.

The system also maintains a set of application-specific **invariants**. For instance, in a banking application, bank balance values are never negative; stock values must be non-negative as well in a shopping cart application; and, in a bidding website, the winner of an auction must issue the highest accepted bid. To capture this, we define the primitive $valid(S)$ to be *true* if state S satisfies all these invariants and *false* otherwise. We denote a valid initial state of every service by S_0. We say an operation u is *correct* if for all valid states S, $S+u$ is also valid. In the previous banking example, a `deposit` operation, adding a positive delta to a user's account balance, is *correct*, as its application against any valid state always ensures that the corresponding balance value is above zero. Unlike `deposit`, a `withdraw` operation can be *correct* if it includes an `if` statement to check whether there is enough balance, but would be incorrect if the programmer did not include this check.

3 Coexistence of strong and weak consistency

In this chapter, we present RedBlue consistency, a novel consistency definition, which allows us to strike a balance between performance and targeted consistency semantics when building (geo-)replicated services, and the design, implementation, and evaluation of Gemini, a geo-distributed storage system enabling RedBlue consistent replication.

This chapter is organized as follows. We first motivate the need for defining RedBlue consistency and briefly describe the major contributions of this work in Section 3.1. Then, we position our work in comparison to existing proposals in Section 3.2. We define RedBlue consistency and sketch the proofs of ensuring end-to-end properties in Section 3.3. In Section 3.4, we introduce the concept of shadow operations along with a set of principles for how to use this concept under RedBlue consistency. We describe our prototype system Gemini in Section 3.5, and report on the experience transitioning three application benchmarks to be RedBlue consistent in Section 3.6. We analyze experimental results in Section 3.7. Limitations are discussed in Section 3.8 and we conclude the work in Section 3.9.

3.1 Motivation and contributions

As we mentioned in Chapter 1, scaling services over the Internet to meet the needs of an ever-growing user base is challenging. In particular, in order to improve user-perceived

latency, which directly affects the quality of the user experience [Lin06, SB09], services replicate system state across geographically diverse sites and direct users to the closest or least loaded site.

To avoid paying the performance penalty of synchronizing concurrent actions across data centers, some systems, such as Amazon's Dynamo [DHJ+07], resort to weaker consistency semantics like eventual consistency where the state can temporarily diverge. Others, such as Yahoo!'s PNUTS [CRS+08], avoid state divergence due to the undesirable sets of behaviors it allows, by requiring all operations that update the service state to be funneled through a primary site and thus incurring increased latency.

In order to address the inherent tension between improving performance and maintaining meaningful consistency semantics, several approaches have been recently proposed for allowing multiple levels of consistency to coexist [LLSG92, SPAL11, SFK+09]: some operations can be executed optimistically, without synchronizing with concurrent actions at other sites, while others require a stronger consistency level and thus require cross-site synchronization. However, this places a high burden on the developer of the service, who must decide which operations to assign which consistency levels. It is challenging to make such decisions since it requires reasoning about the consistency semantics of the overall system to ensure that the behaviors that are allowed by the different consistency levels satisfy the specification of the system.

In this chapter we present a comprehensive and principled approach to this problem, aiming at enabling geo-replicated systems to be as fast as possible while ensuring that they are consistent when necessary. We make the following three contributions:

1. We propose a novel consistency definition called RedBlue consistency. The intuition behind RedBlue consistency is that blue operations execute locally and are lazily replicated in an eventually consistent manner [DHJ+07, LFKA11, TTP+95, MSL+11, FZFF10, SPBZ11b, SFK+09]. Red operations, in contrast, are serialized with respect to each other and require immediate cross-site coordination. In

addition, RedBlue consistency preserves causality by ensuring that dependencies established when an operation is invoked at its primary site are preserved as the operation is incorporated at other sites.

2. We identify the sufficient conditions under which operations must be colored red and may be colored blue in order to ensure that application invariants are never violated and that all replicas converge on the same final state. Intuitively, operations that commute with all other operations and do not impact invariants may be blue; the remaining ones must be red.

3. We observe that the commutativity requirement limits the space of potentially blue operations, provided that many operations in real world applications do not commute w.r.t each other. To address this limitation, we decompose operations into two components: (1) a generator operation that identifies the changes the original operation should make, but has no side effects itself, and (2) a shadow operation that performs the identified changes and is replicated to all sites. With this decomposition, only shadow operations are colored red or blue. This allows for a dynamic runtime classification of operations and hence broadens the space of potentially blue operations.

We built a system called Gemini that coordinates RedBlue consistent replication, and use it to extend three applications to be RedBlue consistent: the TPC-W and RUBiS benchmarks and the Quoddy social network. Our evaluation using microbenchmarks and the three applications shows that RedBlue consistency provides substantial latency and throughput benefits. Furthermore, our experience with modifying these applications indicates that shadow operations can be created with modest effort.

Consistency level	Example systems	Immediate response	State convergence	Single value	General operations	Stable histories	Classification strategy
Strong	RSM [Lam78, Sch90]	no	yes	yes	yes	yes	N/A
Timeline/ snapshot	PNUTS [CRS+08], Megastore [BBC+11]	reads only	yes	yes	yes	yes	N/A
Fork	SUNDR [LKMS04]	all ops	no	yes	yes	yes	N/A
Eventual	Bayou [TTP+95], Depot [MSL+11]	all ops	yes	no	yes	yes	N/A
	Sporc [FZFF10], CRDT [SPBZ11b]	all ops	yes	yes	no	yes	N/A
	Zeno [SFK+09], COPS [LFKA11]	weak/all ops	yes	yes	yes	no	no / N/A
Multi	PSI [SPAL11]	cset	yes	yes	partial	yes	no
	lazy repl. [LLSG92], Horus [vRBM96]	immediate/causal ops	yes	yes	yes	yes	no
RedBlue	Gemini	Red ops	yes	yes	yes	yes	yes

Table 3.1: Tradeoffs in geo-replicated systems and various consistency levels.

3.2 Related work

In this section, we compare several proposals of consistency definitions against our work by analyzing which set of end-to-end properties described in Chapter 2 they offer. Table 3.1 shows that different proposals strike different balances between these target properties. While other consistency definitions exist, we focus on the ones most closely related to the problem of offering fast and consistent responses in geo-replicated systems.

Strong vs. weak consistency. On the strong consistency side of the spectrum, there are definitions like linearizability [HW90], where the replicated system behaves like a single server that serializes all operations. This, however, requires coordination among replicas to agree on the order in which operations are executed, with the corresponding overheads that are amplified in geo-replication scenarios. Somewhat more efficient are timeline consistency in PNUTS [CRS+08] and snapshot consistency in Megastore [BBC+11]. These systems ensure that there is a total order for updates to the service state, but give the option of reading a consistent but dated view of the service. Similarly, Facebook has a primary site that handles updates and a secondary site that acts as a read-only copy [Li, Sob08]. This allows for fast reads executed at the closest site but writes still pay a penalty for serialization. Fork consistency [LKMS04, MS02] addresses the performance limitations of strong consistency by allowing users to observe distinct causal histories. The primary drawback of fork consistency is that once replicas have forked, they can never be reconciled. Such approach is useful when building secure systems but is not appropriate in the context of geo-replication.

Eventual consistency [TTP+95] is on the other end of the spectrum. Eventual consistency is a catch-all phrase that covers any system where replicas may diverge in the short term as long as the divergence is eventually repaired and may or may not include causality. (See Saito and Shapiro [SS05] for a survey.) In practice, as shown in Table 3.1, systems that embrace weak consistency (e.g., eventual or causal consistency) have limi-

tations. Some systems waive the stable history property, either by rolling back operations and re-executing them in a different order at some of the replicas [SFK+09], or by re-sorting to a last-writer-wins strategy, which often results in loss of one of the concurrent updates [LFKA11]. Other systems expose multiple values from divergent branches in operations replies either directly to the client [MSL+11, DHJ+07] or to an application-specific conflict resolution procedure [TTP+95]. Finally, some systems restrict operations by assuming that all operations in the system commute [FZFF10, SPBZ11b], which might require the programmer to rewrite or avoid using some operations.

Coexistence of multiple consistency levels. The solution we propose for addressing the tension between low latency and strongly consistent responses is to allow different operations to run with different consistency levels. Existing systems that used a similar approach include Horus [vRBM96], lazy replication [LLSG92], Zeno [SFK+09], and PSI [SPAL11]. However, none of these proposals guide the service developer in choosing between the available consistency levels. In particular, developers must reason about whether their choice leads to the desired service behavior, namely by ensuring that invariants are preserved and that replica state does not diverge. This can be challenging due to difficulties in identifying behaviors allowed by a specific consistency level and understanding the interplay between operations running at different levels. Our research addresses this challenge, namely by defining a set of conditions that precisely determine the appropriate consistency level for each operation.

Other related work. Consistency rationing [KHAK09] allows consistency guarantees to be associated with data instead of operations, and the consistency level to be automatically switched at runtime between weak consistency and serializability based on specified policies. TACT [YV00] consistency bounds the amount of inconsistency of data items in an application-specific manner, using the following metrics: numerical error, order error and staleness. In contrast to these models, the focus of RedBlue consistency

is not on adapting the consistency levels of particular data items at runtime, but instead on systematically partitioning the space of operations according to their actions and the desired system semantics.

One of the central aspects of our work is the notion of shadow operations, which increase operation commutativity by decoupling the decision of the side effects from their application to the state. Some prior work also aims at increasing operation commutativity: Weihl exploited commutativity-based concurrency control for abstract data types [Wei88]; operational transformation [EG89, FZFF10] extends non-commutative operations with a transformation that makes them commute; Conflict-free Replicated Data Types (CRDTs) [SPBZ11b] design operations that commute by construction; Gray [Gra81] proposed an open nested transaction model that uses commutative compensating transactions to revert the effects of aborted transactions without rolling back the transactions that have seen their results and already committed; delta transactions [Sto10] divide a transaction into smaller pieces that commute with each other to reduce the serializability requirements. Our proposal of shadow operations can be seen as an extension to these concepts, providing a different way of broadening the scope of potentially commutative operations. There exist other proposals that also decouple the execution into two parts, namely two-tier replication [GHOS96] and CRDT downstreams [SPBZ11b]. In contrast to these proposals, for each operation, we may generate different shadow operations based on the specifics of the execution, which can run under different consistency levels. As a result, the decomposition enables a dynamic runtime classification of consistency levels, and allows applications to make more use of fast operations.

3.3 RedBlue consistency

In this section we introduce RedBlue consistency, a novel consistency model that allows replicated systems to be fast as possible and consistent when necessary. “Fast” is an easy

concept to understand—it equates to providing low latency responses to user requests. "Consistent" is more nuanced—consistency models technically restrict the state that operations can observe, which can be translated to an order that operations can be applied to a system. As we saw, causal consistency [LFKA11, TTP+95, MSL+11, FZFF10], for example, permits operations to be partially ordered and enables fast systems—sites can process requests locally without coordinating with each other—but sacrifices the intuitive semantics of serializing updates. In contrast, linearizability [HW90] or serializability [BHG87] provide strong consistency and allow for systems with intuitive semantics—in effect, all sites process operations in the same order —but require significant coordination between sites, precluding fast operation.

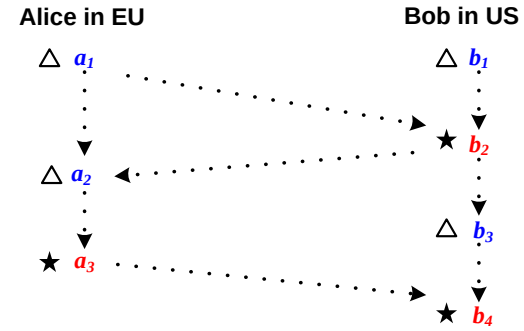
RedBlue consistency is designed to allow systems to support fast causally consistent execution when possible and (slower) strongly consistent execution when necessary. It is based on an explicit division of operations into blue operations whose order of execution can vary from site to site, and red operations that must be executed in the same order at all sites.

3.3.1 Defining RedBlue consistency

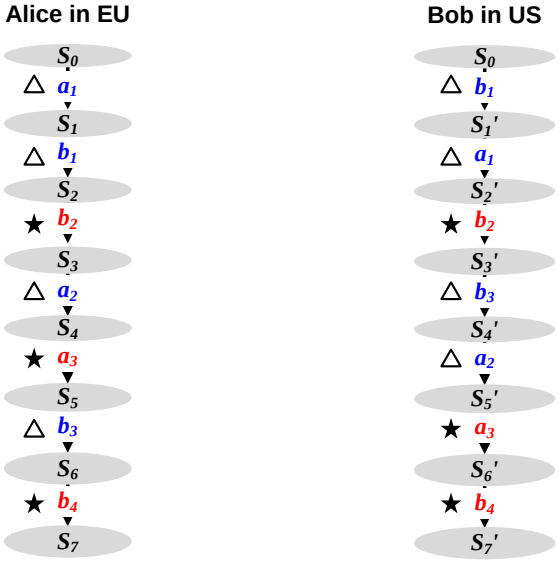
The definition of RedBlue consistency has two components: (1) A RedBlue order, which defines a (global) partial order of operations, and (2) a set of local causal serializations, which define site-specific total orders in which the operations are locally applied.

Definition 1 (RedBlue order) *Given a set of operations $U = R \cup B$, where R and B denote the red and blue operation set, respectively, and $R \cap B = \emptyset$, a* RedBlue order *is a partial order $O = (U, \prec)$ with the restriction that $\forall u, v \in R$ such that $u \neq v$, $u \prec v$ or $v \prec u$ (i.e., red operations are totally ordered).*

Recall that each site is modeled as a deterministic state machine capable of processing a totally ordered sequence of operations. We define which serializations are allowed for a given RedBlue order as follows:



(a) RedBlue order O of operations



(b) Causal serializations of O

Figure 3.1: RedBlue order and causal serializations for a system spanning two sites. Operations marked with ⋆ are red; operations marked with △ are blue. Dotted arrows in (a) indicate dependencies between operations.

Definition 2 (Legal serialization) $O' = (U, <)$ *is a* legal serialization *of RedBlue order* $O = (U, \prec)$ *if*

- O' *is a linear extension of* O*; i.e.,* $<$ *is a total order compatible with the partial order defined by* $\prec$*.*

This definition forces the serial order by which replicas execute operations to be compatible with the RedBlue order. However, it fails to enforce causality, meaning that if

an operation v sees the effects of operation u at its primary site, then any operation w that sees the effects of v must also see the effect of u at all sites in the system. In order to preserve causality, we extend the above definition by saying that if operation v sees the effects of u at its primary site, $site(v)$, then u must be serialized before v at all sites.

Definition 3 (Causal legal serialization) *Given a site i, $O_i = (U, <)$ is an* i-causal legal serialization *(or short, a* causal serialization*) of RedBlue order $O = (U, \prec)$ if*

- *O_i is a legal serialization of O, and*
- *for any two operations $u, v \in U$, if $site(v) = i$ and $u < v$ in O_i, then $u \prec v$.*

A replicated system with k sites is then RedBlue consistent if every site applies a causal serialization of the same global RedBlue order O.

Definition 4 (RedBlue consistency) *A replicated system is* O-RedBlue consistent *(or short,* RedBlue consistent*) if each site i applies operations according to an i-causal serialization of RedBlue order O.*

Figure 3.1 shows a RedBlue order and a pair of causal serializations of that RedBlue order. In systems where every operation is labeled red, RedBlue consistency is equivalent to serializability [BHG87]; in systems where every operation is labeled blue, RedBlue consistency allows the same set of behaviors as causal consistency [TTP+95, LFKA11, MSL+11]. It is important to note that while RedBlue consistency constrains possible orderings of operations at each site and thus the states the system can reach, it does not ensure *a priori* that the system achieves all the end-to-end properties identified in Chapter 2, namely, state convergence and invariant preservation, as discussed next.

3.3.2 State convergence and a RedBlue bank

In order to understand RedBlue consistency it is instructive to look at a concrete example. For this example, consider a simple bank with two users: Alice in the EU and

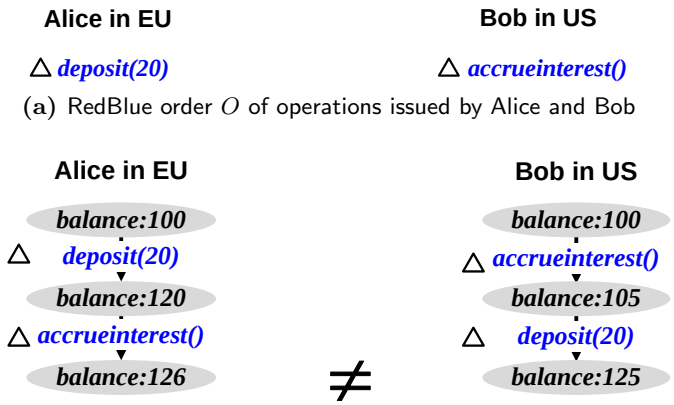
```
1  Variables:
2  float balance, interest = 0.05;

4  Operations:
5  func deposit( float money ){
6     balance = balance + money;
7  }

9  func withdraw( float money ){
10    if ( balance - money >= 0 ){
11       balance = balance - money;
12    }else{
13       print "failure";
14    }
15 }

17 func accrueinterest(){
18    float delta = balance × interest;
19    balance = balance + delta;
20 }
```

Figure 3.2: Pseudocode for the bank example.

Bob in the US. Alice and Bob share a single bank account where they can deposit or withdraw funds and where a local bank branch can accrue interest on the account (pseudocode for the operations can be found in Figure 3.2). To make the bank example fast, let the `deposit` and `accrueinterest` operations be blue. Figure 3.3 shows a RedBlue order of deposits and interest accruals made by Alice and Bob and two possible causal serializations applied at both branches of the bank.

State convergence is important for replicated systems. Intuitively a pair of replicas is state convergent if, after processing the same set of operations, they are in the same state. In the context of RedBlue consistency we formalize state convergence as follows:

Definition 5 (State convergence) *A RedBlue consistent system is state convergent if all causal serializations of the underlying RedBlue order O reach the same state S w.r.t any initial state S_0.*



(a) RedBlue order O of operations issued by Alice and Bob

(b) Causal serializations of O leading to diverged state

Figure 3.3: A RedBlue consistent account with initial balance of $100 and final diverged state.

The bank example as described is not state convergent. The root cause is not surprising: RedBlue consistency allows sites to execute blue operations in different orders but two blue operations in the example correspond to non-commutative operations—addition (`deposit`) and multiplication (`accrueinterest`). A sufficient condition to guarantee state convergence in a RedBlue consistent system is that every blue operation is *globally commutative*, i.e., it commutes with all other operations, blue or red. We formally define this condition in the following theorem.

Theorem 1 *Given a RedBlue order O, if all blue operations are globally commutative, then any O-RedBlue consistent system is state convergent.*

In order to prove the above theorem, we introduce the following three lemmas along with their proofs.

The first lemma asserts that, given a legal serialization, swapping two adjacent operations in the legal serialization that are not ordered by the underlying RedBlue order results in another legal serialization.

Lemma 1 *Given a legal serialization $O_i = (U, <_i)$ of RedBlue order $O = (U, \prec)$ with operations $u, v \in U$ such that $u <_i v$ and $u \not\prec v$ and there exists no s such that $u <_i s <_i v$,*

and let $P = \{p | p \in U \wedge p <_i u\}$ *and* $Q = \{q | q \in U \wedge v <_i q\}$. *The serialization* $O_k = (U, <_k)$ *where*

- $\forall p, q \in P \cup Q : p <_k q \iff p <_i q$,
- $\forall p \in P : p <_k v$,
- $v <_k u$,
- $\forall q \in Q : u <_k q$

is a legal serialization.

Proof: It suffices to show that $\forall r, s \in U : r <_k s$ is compatible with $\prec$. To do so, we consider the following six cases:

- **Case 1:** $r, s \in P \cup Q$. Since O_i is a legal serialization, each $r <_i s$ is compatible with $\prec$ by definition. By construction $\forall p, q \in P \cup Q : r <_k s \iff r <_i s$, so each $r <_k s$ is also compatible with $\prec$.
- **Case 2:** $r \in P$, $s = v$. $r <_k s$ is compatible with $\prec$ by similar logic as above.
- **Case 3:** $r = u$, $s \in Q$. $r <_k s$ is compatible with $\prec$ by similar logic as above.
- **Case 4:** $v <_k u$. Since $u \not\prec v$, $v <_k u$ is compatible with $\prec$.
- **Case 5:** $r \in P$, $s = u$. Since $v <_k u \wedge \forall p \in P : p <_k v \implies p <_k u$. By the construction of P, $\forall p \in P : p <_k u \iff p <_i u$. So each $r <_k s$ is also compatible with $\prec$.
- **Case 6:** $r = v$, $s \in Q$. Since $v <_k u \wedge \forall q \in Q : v <_k q \implies v <_k q$. $r <_k s$ is compatible with $\prec$ by similar logic as above.

As $U = P \cup Q \cup \{u, v\}$, by all above cases, $\forall r, s \in U : r <_k s$ is compatible with $\prec$. ∎

The following lemma asserts that given a RedBlue order and its legal serialization, if there exists a pair of elements u and v that are not ordered by the RedBlue order, then there exists an adjacent pair of elements between u and v in the legal serialization that are not ordered by the RedBlue order.

Lemma 2 *Given a legal serialization* $O_i = (U, <_i)$ *of RedBlue order* $O = (U, \prec)$*, if* $\exists u, v \in U$ *such that* $u <_i v$ *and* $u \not\prec v$*, let* $U' = \{u, v\} \cup \{q | u <_i q \wedge q <_i v\}$*, then* $\exists r, s \in U'$ *such that* $r <_i s \wedge\ r \not\prec s \wedge \nexists p \in U' : r <_i p \wedge p <_i s$.

Proof: We prove this by performing the following exhaustive analysis. The analysis terminates when the required pair of elements is found.

Let's start with u, v. Consider Q to be the sequence of elements strictly between u and v, i.e., $Q = \{q \in U | u <_i q \wedge q <_i v\}$. There are two cases we have to analyze:

- **Case 1:** Q is empty. This implies that u and v are adjacent, so the analysis terminates.
- **Case 2:** Q is not empty. This implies that u and v are not adjacent. Consider p to be the first element in Q according to $<_i$, i.e., $p \in Q : \forall q \in Q \setminus \{p\}, p <_i q$. There are two cases to consider:
  - **Case 2a:** $u \not\prec p$. It follows that p is the successor of u in O_i, then u, p is the adjacent pair that is not ordered by O. The analysis terminates.
  - **Case 2b:** $u \prec p$. It follows from the assertion that $u \not\prec v$ and the transitivity of $\prec$ that $p \not\prec v$. Then we run the analysis from the beginning with p, v. Since we are removing the first element of the sequence Q, the analysis will either eventually terminate with an empty sequence, or before that.

∎

The third lemma asserts that two legal serializations that differ in the order of exactly one pair of adjacent operations (one of which is blue) are state convergent, if all their blue operations are globally commutative.

Lemma 3 *Assume $O_i = (U, <_i)$ and $O_j = (U, <_j)$ are both legal serializations of RedBlue order $O = (U, \prec)$ that are identical except for two adjacent operations u and v such that $u <_i v$ and $v <_j u$ and that all blue operations $r \in U$ are globally commutative. Then $S_0(O_i) = S_0(O_j)$.*

Proof: Let P and Q be the greatest common prefix and suffix respectively of O_i and O_j. Further, let $S_P = S_0(P)$, $S_{uv} = S_P + u + v$, and $S_{vu} = S_P + v + u$.

It follows from the definition of legal serialization (Definition 2) that u and v are not partially ordered in $\prec$. It then follows from the definition of a RedBlue order (Definition 1) that either u or v is blue, i.e., $u \in B$ or $v \in B$. Without loss of generality, assume $u \in B$. By assumption u commutes with all operations in U, therefore $S_{uv} = S_{vu}$. It then follows from the definition of a deterministic state machine that $S_{uv}(Q) = S_{vu}(Q)$. By a similar argument, the final state reached by sequentially executing operations in O_i against S_0 according to $<_i$ is equal to the final state obtained by sequentially applying operations in Q against S_{uv} according to $<_i$, namely $S_0(O_i) = S_{uv}(Q)$. By a similar argument, we know $S_0(O_j) = S_{vu}(Q)$. Finally, we have $S_0(O_i) = S_0(O_j)$. ∎

With the above lemmas, we could prove the state convergence theorem (Theorem 1) as follows:

Proof: To prove a RedBlue consistent system is state convergent, it is sufficient to show that for a RedBlue order O of that system, any pair of its causal legal serializations reaches the same final state w.r.t any initial state S_0. To achieve this, we take a slightly more conservative approach, which is to prove that any pair of legal serializations of their underlying RedBlue order O is state convergent. Let O_i and O_j be two legal serializations of O. There are two cases to consider:

- **Case 1:** $O_i = O_j$. The underlying deterministic state machine ensures that $S_0(O_i) = S_0(O_j)$.

- **Case 2:** $O_i \neq O_j$, in which case $\exists u, v \in U$ such that $u <_i v$ and $v <_j u$. Since both O_i and O_j are legal serializations of O, it follows that $u \not\prec v$ and $v \not\prec u$. It then follows from Lemma 2 that we can find an adjacent pair of operations r, s such that $r <_i s \wedge s <_j r \wedge r \not\prec s \wedge s \not\prec r$. We construct a new serialization O_{i+1} by first duplicating O_i and then swapping the order of r and s in O_{i+1}, i.e., Q_i and Q_{i+1} are identical, except that $r <_i s \wedge s <_{i+1} r$. By Lemma 1, O_{i+1} is also a legal serialization of O.

  If $O_{i+1} \neq O_j$, we continue the construction by finding an adjacent pair of elements whose order is different in O_{i+1}, O_j. By swapping the two operations, we obtain another legal serialization O_{i+2}. We can then continue to swap all such adjacent pairs until the last constructed serialization is equal to O_j. This is achievable since for any two legal serializations generated from two consecutive steps, O' and O'', the number of pairs in O'' whose orders are different in O_j becomes smaller than the number observed in O'. At the end, the construction process results in a chain of legal serializations where the first one is O_i and the last is O_j, and any consecutive pair of legal serializations is identical except for the order of an adjacent pair of operations. It then follows from Lemma 3 and the assumption that all blue operations are globally commutative that every consecutive pair of serializations in the chain is state convergent. Thus, $S_0(O_i) = S_0(O_j)$. ∎

Theorem 1 highlights an important tension inherent to RedBlue consistency. On the one hand, low latency requires an abundance of blue operations that can be locally executed and lazily replicated. On the other hand, state convergence requires that blue operations commute with all other operations, blue or red. In order to make the banking example shown in Figure 3.2 and 3.3 converge, one has to label all three operations red, namely `deposit`, `withdraw` and `accrueinterest`. Obviously, this labeling will lead to a significant performance penalty, due all operations must be serialized w.r.t each other. The poor result implies that there exists an obstacle to making systems fast

under RedBlue consistency, which is that the number of commuting operations in the real world is quite limited. As a result, in the following section we introduce a method for addressing this tension by significantly increasing the amount of commutativity in application operations.

3.4 Replicating side effects

In this section, we observe that while operations themselves may not be commutative, *we can often make the changes they induce on the system state commute.* Let us illustrate this issue within the context of the RedBlue bank from Section 3.3.2. We can make the `deposit` and `accrueinterest` operations commute by first computing the amount of interested accrued and then treating that value as a deposit.

3.4.1 Defining shadow operations

The key idea is to split each original application operation u into two components: a *generator operation* g_u with no side-effects, which is executed only at the primary site against some system state S and produces a *shadow operation* $h_u(S)$, which is executed at every site (including the primary site). The generator operation decides which state transitions should be made while the shadow operation applies the transitions in a state-independent manner.

The simplest way of making such a decomposition is generating a no-op shadow operation for every original operation. Although this strategy makes every shadow operation globally commutative and potentially blue, it delivers a completely unmeaningful service. In order to follow the intended application semantics, one cannot split original operations in an arbitrary manner: the implementation of generator and shadow operations must obey some basic correctness requirements. First, generator operations, as mentioned, must not have any side effects. Furthermore, shadow operations must produce the same

effects as the corresponding original operation when executed against the original state S used as an argument in the creation of the shadow operation. More formally:

Definition 6 (Correct generator / shadow operations) *The decomposition of operation u into generator and shadow operations is correct if for all states S, the generator operation g_u has no effect and the generated shadow operation $h_u(S)$ has the same effect as u w.r.t S, i.e., for any state S: $S + g_u = S$ and $S + h_u(S) = S + u$.*

Note that a trivial decomposition of an original operation u into generator and shadow operations is to let g_u be a no-op and let $h_u(S) = u$ for all S. This is correct but it does not increase the space of commutativity. Later in this chapter, we will present a few examples, in which we made an effort to produce commutative shadow operations.

In practice, as exemplified in Section 3.6, separating the decision of which transition to make from the act of applying the transition allows many objects and their associated usage in shadow operations to form an abelian group and thus dramatically increase the number of commutative (i.e., blue) operations in the system. Furthermore, unlike previous approaches [GHOS96, SPBZ11b], for a given original operation, our solution allows its generator operation to generate state-specific shadow operations with different properties, which can then be assigned different colors in the RedBlue consistency model.

3.4.2 Revisiting RedBlue consistency

The key insight that underlies shadow operations is breaking the execution of an operation down into the decide (generator) and apply (shadow) phases. This decomposition, however, requires us to revisit the foundations of RedBlue consistency. In particular, only shadow operations are included in a RedBlue order while the causal serialization for site i additionally includes the generator operations initially executed at site i. The causal serialization must ensure that generator operations see the same state that is associated with the generated shadow operation and that shadow operations appropriately inherit all dependencies from their generator operation.

```
1  func deposit'( float money ){
2     balance = balance + money;
3  }

5  func withdrawAck'( float money ){
6     balance = balance - money;
7  }

9  func withdrawFail'(){
10    /* no-op */
11 }

13 func accrueinterest'( float delta ){
14    balance = balance + delta;
15 }
```

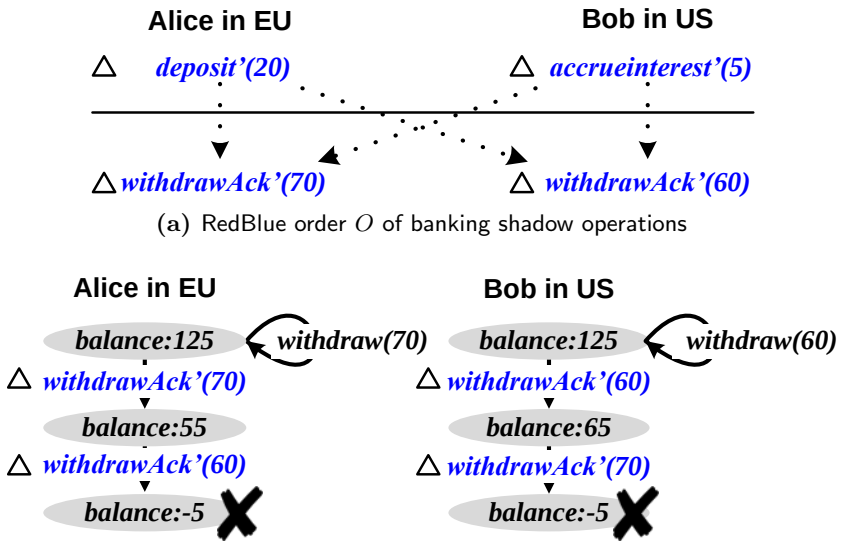
Figure 3.4: Pseudocode for shadow bank operations.

We capture these subtleties in the following revised definition of causal serializations. Let U be the set of shadow operations executed by the system and V_i be the generator operations executed at site i.

Definition 7 (Causal serialization–revised) *Given a site i, $O_i = (U \cup V_i, <)$ is an i-causal serialization of RedBlue order $O = (U, \prec)$ if*

- *O_i is a total order;*
- *$(U, <)$ is a linear extension of O;*
- *For any $h_v(S) \in U$ generated by $g_v \in V_i$, S is the state obtained after applying the sequence of shadow operations preceding g_v in O_i;*
- *For any $g_v \in V_i$ and $h_u(S), h_v(S') \in U$, $h_u(S) < g_v$ in O_i iff $h_u(S) \prec h_v(S')$ in O.*

Note that shadow operations appear in every causal serialization, while generator operations appear only in the causal serialization of the initially executing site. Unlike the causal serialization definition, the definitions of legal serialization and RedBlue order re-



(a) RedBlue order O of banking shadow operations

(b) Invalid but convergent causal serializations of O

Figure 3.5: A RedBlue consistent bank with only blue operations reaches an invalid state. The starting balance of \$125 is the result of applying shadow operations above the solid line to an initial balance of \$100. Loops indicate generator operations.

main fundamentally unchanged, with the only exception on that "operation" is replaced with "shadow operation".

3.4.3 Shadow banking and invariants

Figure 3.4 shows the shadow operations for the banking example. In this example, the `withdraw` operation maps to two distinct shadow operations that may be labeled as blue or red independently—`withdrawAck'` and `withdrawFail'`. `withdrawAck'` refers to successful withdrawal, while `withdrawFail'` corresponds to failure due the balance value is not enough.

Figure 3.5 illustrates that shadow operations make it possible for all operations to commute, provided that we can identify the underlying abelian group. This does not mean, however, that it is safe to label all commutative shadow operations blue. In this example (Figure 3.5(b)), such a labeling would allow Alice and Bob to successfully

withdraw \$70 and \$60 at their local branches, thus ending up with a final balance of \$-5. This violates the fundamental invariant that a bank balance should never be negative.

To determine which shadow operations can be safely labeled blue, we begin by defining that a shadow operation is invariant safe if, when applied to a valid state, it always transitions the system into another valid state.

Definition 8 (Invariant safe) *Shadow operation $h_u(S)$ is invariant safe if for all valid states S and S', the state $S' + h_u(S)$ is also valid.*

We also assume that the original applications without being RedBlue consistent replicated are correct, i.e., all their original operations always transition from a valid system state to another valid state. This is captured by the following trivial definition:

Definition 9 (Correct original operation) *Original operation t is correct if for all valid states S, $S + t$ is also valid.*

The following theorem states that in a RedBlue consistent system with appropriate labeling, each replica transitions only through valid states.

Theorem 2 *Given a RedBlue consistent system, if every original operation and any pair of generator and shadow operations is correct and all its blue shadow operations are invariant safe and globally commutative, then for any execution of that system that starts from a valid state, no site is ever in an invalid state.*

It is worth noting that this theorem highlights a non-obvious result: even red shadow operations that may break invariants are allowed to be applied against completely different state, provided that those operations are serialized w.r.t each other in the same order at all sites (but not w.r.t the remaining ones).

Proof by contradiction. Let $O = (U, \prec)$ be a RedBlue order. For every shadow operation u in U, u's original operation is correct and the corresponding decomposition

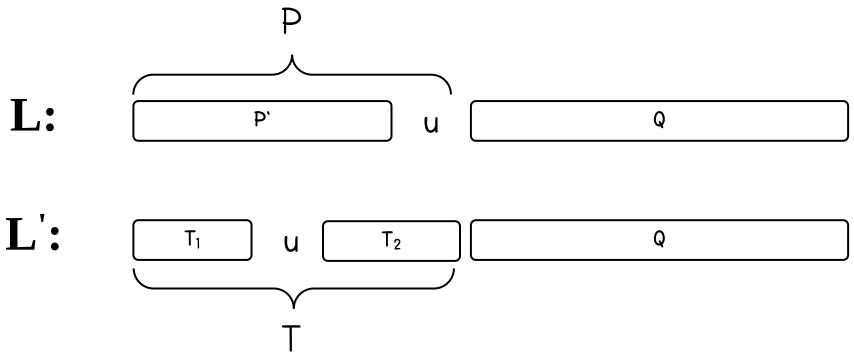


Figure 3.6: Two legal serializations L and L'. L' is constructed by swapping every shadow operation v in P' and u if u and v are not partially ordered.

is correct. Every blue shadow operation v, i.e., $v \in B$, is invariant safe and globally commutative. The initial state S_0 is valid.

Let L be a causal serialization of O, which is shown in Figure 3.6. Assume that L is in an invalid state. We prove this theorem by performing the following exhaustive analysis and showing the contradictions found.

Analysis: Let $P(U_P, <_P)$ be the shortest prefix of L that produces an invalid state. If P is empty, then $S_0(P) = S_0$, and L is in a valid state. This violates the assumption that L is in an invalid state. The theorem is proved.

If P is non-empty, then consider u to be the last shadow operation in P such that $P = P' + u$, where P' is a prefix of P. Let t be the original operation of u. By the definition of shadow operation, we know $u = h_t(S)$, where S is the state in which u was generated. There are two cases we need to consider:

- **Case 1:** u is blue. As every blue shadow operation is invariant safe, the state reached before applying u, $S_0(P')$, must be invalid. This contradicts the assumption that P is the shortest prefix that introduces an invalid state. The theorem is proved.

- **Case 2:** u is red. S has two possible values.

– **Case 2a:** $S = S_0(P')$, i.e., the state that u was applied against is the same as the state that u was created from. It follows from the correct generator/shadow operation definition (Definition 6) that $S + u = S + t$ and $S + t$ is invalid. It then follows the correct original operation definition (Definition 9) that S must be invalid as well. By the same logic in Case 1, we found a shorter prefix P' other than P that produces an invalid state. By contradiction, the theorem is proved.

– **Case 2b:** $S \neq S_0(P')$. It follows the definition of RedBlue order (Definition 1) and causal serialization (Definition 7) that there exists some blue shadow operations v that precede u in L but are not partially ordered with u in O, i.e., $v <_L u$ and $u \not<_O v \wedge v \not<_O u$. It then follows from Lemmas 1 and 2 that we can construct a new causal serialization L' of O by duplicating L and swapping the order between u and every v in L', so that u is bubbled up over every such v. The result is shown in Figure 3.6. The only difference between L and L' is as follows: $\forall i \in U_p : i <_L u \wedge i \not<_O u \implies u <_{L'} i$. T_1 represents a sequence of shadow operations that precede u in O, while T_2 represents a sequence of blue shadow operations that are not partially ordered with u. By the state convergence theorem (Theorem 1) and the assumption that every blue shadow operation is globally commutative, so the prefix T and P must be state convergent, i.e., $S_0(P) = S_0(T)$. As $S_0(P)$ is invalid, $S_0(T)$ is also invalid.

  By the deterministic state machine model, we know $S_0(T) = S_0(T_1) + u + T_2$, where as shown in Figure 3.6 T_1 and T_2 are the aforementioned prefix and suffix of T, respectively. As all blue shadow operations are invariant safe, $S_0(T_1) + u$ must be invalid. By the causal serialization definition (Definition 7) and the construction of L', the state $S_0(T_1)$ is the state in which u was generated. It then follows the correct generator/shadow operation definition

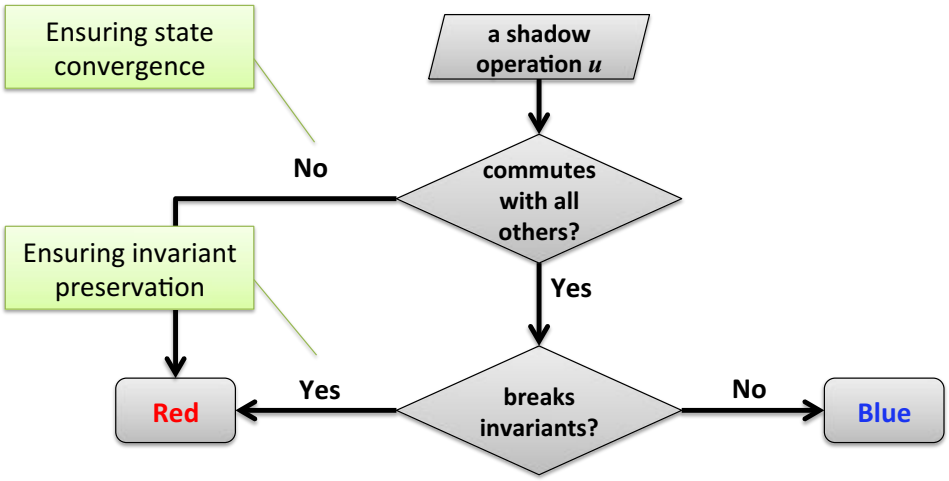


Figure 3.7: Labeling methodology diagram.

(Definition 6) that $S_0(T_1) + u = S_0(T_1) + t$. It then follows from the correct original operation definition (Definition 9) that $S_0(T_1)$ must be invalid as well. We proceed by starting again the analysis using as the input a new causal serialization of O and a new shortest prefix that produces an invalid state, i.e., $P = T_1 \wedge L = L'$. This analysis is guaranteed to terminate since the size of P at every subsequent analysis step decreases.

■

3.4.4 What can be blue? What must be red?

As illustrated by Theorem 1, the sufficient condition of ensuring the state convergence property is that a shadow operation must be labeled as red if it is not globally commutative. The second theorem (Theorem 2) states that invariants are maintained if all non-invariant safe shadow operations are serialized. In summary, the combination of these two theorems leads to the following procedure (shown in Figure 3.7) for deciding which shadow operations can be blue or must be red if a RedBlue consistent system is to provide both state convergence and invariant preservation:

1. For any pair of non-commutative shadow operations u and v, label both u and v red.

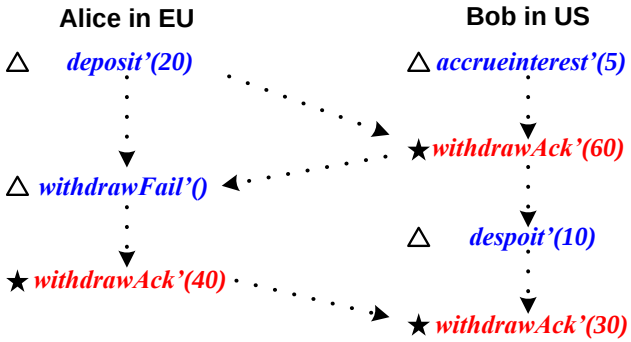
2. For any shadow operation u that may result in an invariant being violated, label u red.

3. Label all non-red shadow operations blue.

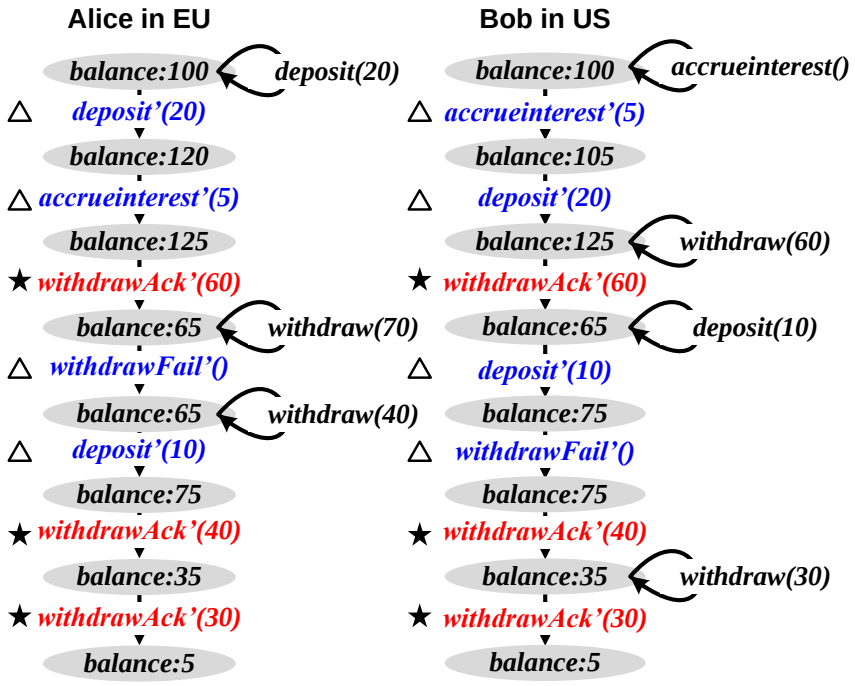
Applying this decision process to the bank example leads to a labeling where `withdrawAck'` is red and the remaining shadow operations are blue. The only restriction we placed is to make any pair of successful withdraw shadow operations be partially ordered, i.e., one must see the effect introduced by another. Figure 3.8 shows a RedBlue order with appropriately labeled shadow operations and causal serializations for the two sites that converge to the same valid final state. In this example, the first `withdraw` operation issued by Alice in the EU site cannot proceed even provided that her local balance is enough to complete this withdrawal. Instead, the execution must wait until the changes carried by the shadow operation `withdrawAck'(60)` from US have been made visible at the EU replica. Upon this, the generator of Alice's `withdraw(70)` reads the current balance value and produces a failure withdrawal (blue). In the end, the balance value remains non-negative.

3.4.5 Discussion

Shadow operations and RedBlue consistency introduce some surprising anomalies to a user experience. Notably, while the effect of every user action is applied at every site, the final system state is not guaranteed to match the state resulting from a serial ordering of the original operations. The important thing to keep in mind is that the decisions made always make sense in the context of the *local* view of the system: when Alice accrues interest in the EU, the amount of interest accrued is based on the balance that Alice observes at that moment. If Bob concurrently makes a deposit in the US



(a) RedBlue order O of banking shadow operations



(b) Convergent and invariant preserving causal serializations of O

Figure 3.8: A RedBlue consistent bank with correctly labeled shadow operations and initial balance of $100.

and subsequently observes that interest has been accrued, the amount of interest *will not* match the amount that Bob would accrue based on the balance as he currently observes it. As such, shadow operations always provide for a coherent sequence of state transitions that reflects the effects demanded by user activity; while this sequence of state transitions is coherent (and convergent), the state transitions are chosen based on

the locally observable state when/where the user activity initiated and not the system state when they are applied.

3.5 Gemini design & implementation

In this section we describe the design and implementation of Gemini, a prototype architecture that enables applications to run under RedBlue consistency.

3.5.1 Design rationale

As we saw in Section 3.4, some original operations like `withdraw` may produce either blue or red shadow operations depending on the current system state and the user input they are observing. A naïve solution would be to coordinate all generator operations that may produce a red shadow. This solution imposes more restrictions than what RedBlue consistency exactly needs, i.e., all relevant shadow operations even including those blue ones would be serialized w.r.t each other. As a result, it offsets the goal of RedBlue consistency of only paying a performance penalty when strong consistency is needed. To avoid this, we instead optimistically run the generator operation at its primary site in the first place, and then speculatively generate a tentative shadow operation based on the local state and user input. If the corresponding shadow operation is blue, then a reply will be produced locally without contacting remote replicas; Otherwise, replicas have to speak to each other for establishing a total order among all red shadow operations that are received, and making sure that this shadow operation is generated from a state reflecting all side effects introduced by all its preceding shadow operations. Therefore, a red tentative shadow operation might rollback, when the local state is different from the global state due to conflicts, and we need to restart the process of generating another shadow operation.

In addition, there are two requirements for executing generator operations: (1) they should not interfere with other concurrent operations; and (2) there is no need for them

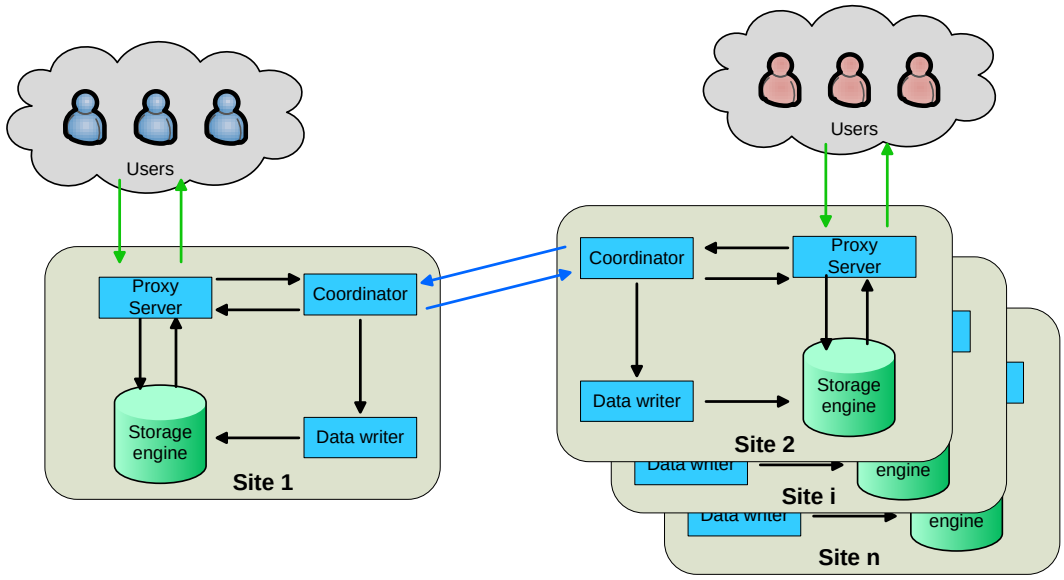


Figure 3.9: Gemini system architecture. Blue arrows represent communication between sites, black arrows indicate communication between system components within a site, and green arrows correspond to communication between users and the replicated service.

to make their identified side effects persistent. Given these observations, using a lock-based concurrency control solution would be very conservative, since granting locks to an operation may prevent other operations from making progress. In summary, we resort to a form of optimistic concurrency control (OCC) [BHG87] in Gemini, as we describe next. It is worth mentioning that the Gemini OCC slightly deviates from the traditional textbook algorithm, since our algorithm recognizes the fact that concurrent blue shadow operations are never conflicting with all other shadow operations.

3.5.2 System overview

We implemented the Gemini storage system to provide RedBlue consistency. As shown in Figure 3.9, each Gemini site consists of four components: a storage engine, a proxy server, a concurrency coordinator, and a data writer. A multi-site deployment is constructed by replicating the single data center components across multiple sites.

The basic flow of user requests through the system is straightforward. A user issues requests to a *proxy server* located at the closest site. The proxy server processes a request by executing the generator operation of an appropriate application transaction, which is implemented as a single Gemini original operation, comprising multiple data accesses; individual data accesses within a generator operation execute in a temporary private scratchpad, providing a virtual private copy of the service state. The original data lies in a *storage engine*, which provides a standard storage interface. In our implementation, the storage engine is a relational database, and scratchpad operations are executed against a set of non-shared in-memory tables. Upon completion of the generator operation, the proxy server sends the produced shadow operation on to the *concurrency coordinator* to admit or reject this operation according to RedBlue consistency. The concurrency coordinator notifies the proxy server if the shadow operation is accepted or rejected. Additionally, accepted shadow operations are appended to the end of the local legal causal serialization and propagated to remote sites and to the local *data writer* for execution against the storage engine. When a shadow operation is rejected, the proxy server re-executes the generator operation and restarts the process.

3.5.3 Ordering and replicating transactions

The most sophisticated part of Gemini is how to establish a RedBlue order of shadow operations generated by different replicas and to replicate all these shadow operations in site-dependent causal legal serializations at every replica. First, Gemini uses timestamps to determine if shadow operations can complete successfully, i.e., shadow operations can be admitted to appear in the corresponding global RedBlue order. Timestamps are logical clocks [Lam78] of the form $\langle\langle b_0, b_1, \ldots, b_{k-1}\rangle, r\rangle$, where b_i is the local count of shadow operations initially executed by site i and r is the global count of red shadow operations. To ensure that different sites do not choose the same red sequence number (i.e., all red operations are totally-ordered) we use a simple token passing scheme: only

the coordinator in possession of a unique red token is allowed to increase the counter r and approve red operations. In the current prototype, a coordinator holds onto the red token for up to 1 second before passing it along.

When a generator operation completes, the corresponding shadow operation is produced and colored according to the classification results obtained by applying the labeling methodology in Section 3.4 against the target application. Then, the colored shadow operation is passed to the coordinator for determining if this operation can be accepted to the global RedBlue order. If it is blue, the coordinator only performs a read coherence check, i.e., the logical timestamps of the data items in its read set are less than or equal to the begin timestamp assigned when the corresponding transaction started. If the pending shadow operation is red, then the coordinator has to verify if the state where the operation was generated from reflects the effects of the set of accepted red shadow operations that precede it according to some total order established by the token assignment scheme. To do this, the coordinator has to wait until the red token has reached its site, i.e., red shadow operations initially executed at the previous red token holder site have been applied locally. Then, the coordinator performs a read-write conflict check consisting of two steps: (a) acquiring locks for data items in the pending shadow operation's write set, in order to prevent local concurrent pending red shadow operations from proceeding; and (b) checking if the data items in the pending shadow operation's read set are not locked and have not been modified by any other accepted shadow operations between the time when the transaction generating the pending shadow operation started and the check was triggered.

Upon successful completion of the above checks, the coordinator assigns the corresponding shadow operation a timestamp that is component-wise equal to the latest operation that was incorporated at its site, and increments its blue and, if this shadow operation is red, the red component of the logical timestamp. This timestamp determines the position of the shadow operation in the RedBlue order, with the normal rules that

determine that two operations are partially ordered if one is equal to or dominates the other in all components. It also allows sites to know when it is safe to incorporate remote shadow operations: they must wait until all shadow operations with smaller timestamps have already been incorporated in the local state of the site. When a remote shadow operation is applied at a site, the most recent local logical clock maintained by this site will be replaced with the entry-wise max of its current value and the timestamp shipped with that shadow operation. This captures dependencies that span local and remote operations.

Read-only shadow operations. As a performance optimization, blue shadow operations can be marked as read-only. Read-only shadow operations receive special treatment from the coordinator: once the generator operation passes the coherence check, the proxy is notified that the shadow operation has been accepted but the shadow operation is *not* incorporated into the local serialization or global RedBlue order. Thus, read-only operations are never sent across sites.

3.5.4 Failure handling

The current Gemini prototype is designed to demonstrate the performance potential of RedBlue consistency in geo-replicated environments and as such is not implemented to tolerate faults of either a local (i.e., within a site) or catastrophic (i.e., of an entire site) nature. Addressing these concerns is orthogonal to the primary contributions of this work, nonetheless we briefly sketch mechanisms that could be employed to handle faults.

Isolated component failure. The Gemini architecture consists of four main components at each site, each representing a single point of failure. Standard state machine replication techniques [Lam78, Sch90] can be employed to make each component robust to failures.

Site failure. Our Gemini prototype relies on a simple ring-exchange for serializing all red shadow operations. Thus, the failure of a single site is enough to stop the token exchange and prevent future red transactions from completing. To avoid halting the system upon a site failure, a fault tolerant consensus protocol like Paxos [Lam98] can regulate red tokens.

Operation propagation. Gemini relies on each site to propagate its own local operations to all remote sites. A pair-wise network outage or failure of a site following the replication of a operation to some but not all of the sites could prevent sites from exchanging operations that depend on the partially replicated operation. This can be addressed using standard techniques for exchanging causal logs [MSL+11, ANB+94, TTP+95, PST+97] or reliable multicast [FJL+97].

Cross-session monotonicity. The proxy that each user connects to enforces the monotonicity of user requests within a session [TDP+94]. However, a failure of that proxy, or the user connecting to a different site may result in a subset of that user's operations not carrying over. This can be addressed by allowing the user to specify a "last-read" version when starting a new session or requiring the user to cache all relevant requests [MSL+11] in order to replay them when connecting to a new site.

3.5.5 Implementation

The Gemini system consists of 10k lines of Java code[1], and uses MySQL [MyS] as its storage backend, and the Netty asynchronous i/o library[Net] for communication. We extended a JDBC driver [Jdb] so that it is able to facilitate the integration of Gemini into the MySQL based applications that will be discussed in Section 3.6. The source code of Gemini is available at [Gem].

[1] The lines of code is measured by `cloc` [cod].

App	Original					RedBlue consistent extension				
	user requests	transactions			LOC	shadow operations				LOC changed
		total	read -only	update		blue		red	LOC	
						no-op	update			
TPC-W	14	20	13	7	9k	13	14	2	2.8k	429
RUBiS	26	16	11	5	9.4k	11	7	2	1k	180
Quoddy	13	15	11	4	15.5k	11	4	0	495	251

Table 3.2: Original applications and the changes needed to make them RedBlue consistent. LOC stands for "Lines of code".

3.6 Case studies

In this section we report on our experience in modifying three existing applications—the TPC-W shopping cart benchmark [con02, TPC11], the RUBiS auction benchmark [EJ09], and the Quoddy social networking application [Fog12]—to work with RedBlue consistency. The two main tasks to fulfill this goal are (1) decomposing the application original operations into generator and shadow operations and (2) labeling the shadow operations blue or red appropriately.

Writing generator and shadow operations. Each of the three case study applications executes MySQL database transactions as part of processing user requests, generally one transaction per request. We map these application level transactions to the original operations and they also serve as a starting point for the generator operations. For shadow operations, we turn each execution path in the original operation into a distinct shadow operation; an execution path that does not modify system state is explicitly encoded as a no-op shadow operation. When the shadow operations are in place, the generator operation is augmented to invoke the appropriate shadow operation at each path.

Labeling shadow operations. Table 3.2 reports the number of transactions in the TPC-W, RUBiS, and Quoddy, the number of blue and red shadow operations we identified using the labeling rules in Section 3.4.3, and the application changes measured in lines

```
1  doBuyConfirm(cartId){
2   beginTxn();
3   cart = exec(SELECT * FROM cartTb WHERE cId=cartId);
4   cost = computeCost(cart);
5   orderId = getUniqueId();
6   exec(INSERT INTO orderTb VALUES(orderId,cart.item.id,
        cart.item.qty,cost));
7   item =exec(SELECT * FROM itemTb WHERE id=cart.item.id);
8   if item.stock − cart.item.qty < 10 then:
9      delta = item.stock − cart.item.qty + 21;
10     if delta > 0 then:
11        exec(UPDATE itemTb SET item.stock+ = delta);
12     else rollback();
13  else exec(UPDATE itemTb SET item.stock− = cart.item.qty);
14  exec(DELETE FROM cartContentTb WHERE cId=cartId AND id=
        cart.item.id);
15  commit();
16 }
```

Figure 3.10: Pseudocode for the product purchase transaction `doBuyConfirm` in TPC-W. For simplicity the pseudocode assumes that the corresponding shopping cart only contains a single item.

of code. Note that read-only transactions always map to blue no-op shadow operations. In the rest of this section we expand on the lessons learned from making applications RedBlue consistent.

3.6.1 TPC-W

TPC-W [con02] models an online bookstore. The application server handles 14 different user requests such as browsing, searching, adding products to a shopping cart, or placing an order. Each user request generates between one and four transactions that access state stored across eight different tables. We extend an open source implementation of the benchmark [Rit12] to allow a shopping cart to be shared by multiple users across multiple sessions.

Writing TPC-W generator and shadow operations. Of the twenty TPC-W transactions, thirteen are read-only and admit no-op shadow operations. The remaining seven

```
1  doBuyConfirmGenerator(cartId){
2   sp = getScratchpad();
3   sp.beginTxn();
4   cart = sp.exec(SELECT * FROM cartTb
        WHERE cId=cartId);
5   cost = computeCost(cart);
6   orderId = getUniqueId();
7   sp.exec(INSERT INTO orderTb VALUES (
        orderId,cart.item.id,cart.item.qty,
        cost));
8   item = sp.exec(SELECT * FROM itemTb
        WHERE id=cart.item.id);
9   if item.stock − cart.item.qty < 10 then:
10     delta = item.stock − cart.item.qty + 21;
11     if delta > 0 then:
12       sp.exec(UPDATE itemTb SET
             item.stock+ = delta);
13     else sp.discard(); return;
14  else sp.exec(UPDATE itemTb SET
        item.stock− = cart.item.qty);
15  sp.exec(DELETE FROM cartTb WHERE cId=
        cartId AND id=cart.item.id);
16  L_TS = getCommitOrder();
17  sp.discard();
18  if replenished return (
        doBuyConfirmIncre'(orderId,cartId,
        cart.item.id,cart.item.qty,cost,delta,
        L_TS));
19  else return (doBuyConfirmDecre'(
        orderId,cartId,cart.item.Id,
        cart.item.qty,cost,L_TS));}
```

(a) Generator operation that manipulates data via a private *scratchpad*.

```
1  doBuyConfirmIncre'(orderId,cartId,
       itId,qty,cost,delta,L_TS){
2   exec(INSERT INTO orderTb VALUES(
        orderId,itId,qty,cost,L_TS));
3   exec(UPDATE itemTb SET
        item.stock+ = delta);
4   exec(UPDATE itemTb SET
        item.l_ts = L_TS WHERE
        item.l_ts < L_TS);
5   exec(UPDATE cartContentTb SET
        flag = TRUE WHERE id = itId AND
         cid = cartId AND l_ts <= L_TS);}
```

(b) Shadow doBuyConfirmIncre (Blue) that replenishes the stock value.

```
1  doBuyConfirmDecre'(orderId,cartId,
       itId,qty,cost,L_TS){
2   exec(INSERT INTO orderTb VALUES(
        orderId,itId,qty,cost,L_TS));
3   exec(UPDATE itemTb SET
        item.stock− = qty);
4   exec(UPDATE itemTb SET
        item.l_ts = L_TS WHERE
        item.l_ts < L_TS);
5   exec(UPDATE cartContentTb SET
        flag = TRUE WHERE id = itId AND
         cid = cartId AND l_ts <= L_TS);}
```

(c) Shadow doBuyConfirmDecre (Red) that decrements the stock value.

Figure 3.11: Pseudocode for the generator and shadow operations of the original TPC-W transaction `doBuyConfirm` shown in Figure 3.10.

update transactions translate to one or more shadow operations according to the number of distinct execution paths in the original operation.

We now give an example transaction, `doBuyConfirm`, which completes a user purchase. The pseudocode for the original transaction is shown in Figure 3.10. The `doBuyConfirm` transaction removes all items from a shopping cart, computes the total cost of the purchase, and updates the stock value for the purchased items. If the stock would drop below

a minimum threshold, then the transaction also replenishes the stock. The key challenge in implementing shadow operations for `doBuyConfirm` is that the original transaction does not commute with itself or any transaction that modifies the contents of a shopping cart. Naively treating the original transaction as a shadow operation would force every shadow operation to be red.

Figure 3.11(a) shows the generator operation of `doBuyConfirm`, and Figures 3.11(b) and Figures 3.11(c) depict the corresponding pair of shadow operations: `doBuyConfirmIncre'` and `doBuyConfirmDecre'`. The former shadow operation is generated when the stock falls below the minimum threshold and must be replenished; the latter is generated when the purchase does not drive the stock below the minimum threshold and consequently does not trigger the replenishment path. In both cases, the generator operation is used to determine the quantity of the item purchased and total cost as well the shadow operation that corresponds to the initial execution. At the end of the execution of the generator operation these parameters and the chosen shadow operation are then propagated to other replicas.

Labeling TPC-W shadow operations. For the 29 shadow operations in TPC-W, we found that 27 can be blue and only two must be red. To label shadow operations, we identified two key invariants that the system must maintain. First, the number of in-stock items can never fall below zero. Second, the identifiers generated by the system (e.g., for items or shopping carts) must be unique.

The first invariant is easy to maintain by labeling `doBuyConfirmDecre'` (Figure 3.11(c)) and its close variant `doBuyConfirmAddrDecre'` red. We observe that they are the only shadow operations in the system that decrease the stock value, and as such are the only shadow operations that can possibly invalidate the first invariant. Note that the companion shadow operation `doBuyConfirmIncre'` (Figure 3.11(b)) *increases* the stock level, and can never drive the stock count below zero, so it can be blue.

The second invariant is more subtle. TPC-W generates IDs for objects (e.g., shopping carts, items, etc.) as they are created by the system. These IDs are used as keys for item lookups and consequently must themselves be unique. To preserve this invariant, we have to label many shadow operations red. This problem is well-known in database replication [CCA08] and was circumvented by modifying the ID generation code, so that IDs become a pair $\langle appproxy\_id,\ seqnumber \rangle$, where $appproxy\_id$ denotes a globally unique proxy id across sites and $seqnumber$ denotes a counter managed by each proxy. This change makes these operations trivially blue, while not modifying application-specific semantics.

3.6.2 RUBiS

RUBiS [EJ09] emulates an online auction website modeled after eBay [eba12]. RUBiS defines a set of 26 requests that users can issue ranging from selling, browsing for, bidding on, or buying items directly, to consulting a personal profile that lists outstanding auctions and bids. These 26 user requests are backed by a set of 16 transactions that access the storage backend.

Of these 16 transactions, 11 are read-only, and therefore trivially commutative. For the remaining 5 update transactions, we construct shadow operations to make them commute, similarly to TPC-W. Each of these transactions leads to between 1 and 3 shadow operations. The effort to write the shadow operations was nominal and mechanically very similar to our efforts with TPC-W.

Through an analysis of the application logic, we determined three invariants. First, that identifiers assigned by the system are unique. Second, that nicknames chosen by users are unique. Third, that item stock cannot fall below zero. Again, we preserve the first invariant using the global id generation strategy described in Section 3.6.1. The second and third invariants require both `RegisterUser'`, checking if a name submitted

by a user was already chosen, and `storeBuyNow'`, which decreases stock, to be labeled as red.

We also found that the available version of RUBiS is not complete since it lacks a real close auction operation, which declares the winner of each auction when its trading period ends. If such an operation existed, then there would be another invariant: the selected winners must be the users issuing highest accepted bids. It is very challenging to maintain this invariant while improving performance under the context of RedBlue consistency. This is because the shadow operation `storeBid'`— putting a bid on an open auction and updating the total number of bids and the max bid value for the corresponding item — would be labeled red and is considered to be a common request in all RUBiS-like bidding systems. We will illustrate this challenge and the approach to overcome it in Chapter 5.

3.6.3 Quoddy

Quoddy [Fog12] is an open source Facebook-like social networking site. Despite being under development, Quoddy already implements the most important features of a social networking site, such as searching for a user, browsing user profiles, adding friends, posting a message, etc. These main features define 13 user requests corresponding to 15 different transactions. Of these 15 transactions, 11 are read-only transactions, thus requiring trivial no-op shadow operations.

Writing and labeling shadow operations for the 4 remaining transactions in Quoddy was straightforward. Besides reusing the recipe for unique identifiers, we only had to handle an automatic conversion of dates to the local timezone (performed by default by the database) by storing dates in UTC in all sites. In the social network we did not find system invariants to speak of; we found that all shadow operations could be labeled blue.

3.6.4 Experience and discussion

Our experience showed that writing shadow operations is easy; it took us about one week to understand the code, and implement and label shadow operations for all applications. We also found that the strategy of generating a different shadow operation for each distinct execution path is beneficial for two reasons. First, it leads to a simple logic for shadow operations that can be based on operations that are intrinsically commutative, e.g., *increment/decrement*, *insertion/removal*. Second, it leads to a fine-grained classification of operations, with more execution paths leading to blue shadow operations. Finally, we found that it was useful in more than one application to make use of a standard last-writer-wins strategy to make operations that overwrite part of the state commute.

3.7 Evaluation

We evaluate Gemini and RedBlue consistency using microbenchmarks and our three case study applications. The primary goal of our evaluation is to determine if RedBlue consistency can improve latency and throughput in geo-replicated systems. More precisely, we focus on the following main questions:

- What is the impact of colors of shadow operations on user observed latency?
- How does throughput change when varying the ratio of red (strongly consistent) shadow operations?
- What is the prevalence of blue or red shadow operations in the three applications introduced in the previous section?
- How does throughput change when increasing the replication factor (i.e., the number of sites)?
- What is the overhead of Gemini?

	UE	UW	IE	BR	SG
UE	0.4 ms 994 Mbps	85 ms 164 Mbps	92 ms 242 Mbps	150 ms 53 Mbps	252 ms 86 Mbps
UW		0.3 ms 975 Mbps	155 ms 84 Mbps	207 ms 35 Mbps	181 ms 126 Mbps
IE			0.4 ms 996 Mbps	235 ms 54 Mbps	350 ms 52 Mbps
BR				0.3 ms 993 Mbps	380 ms 65 Mbps
SG					0.3 ms 993 Mbps

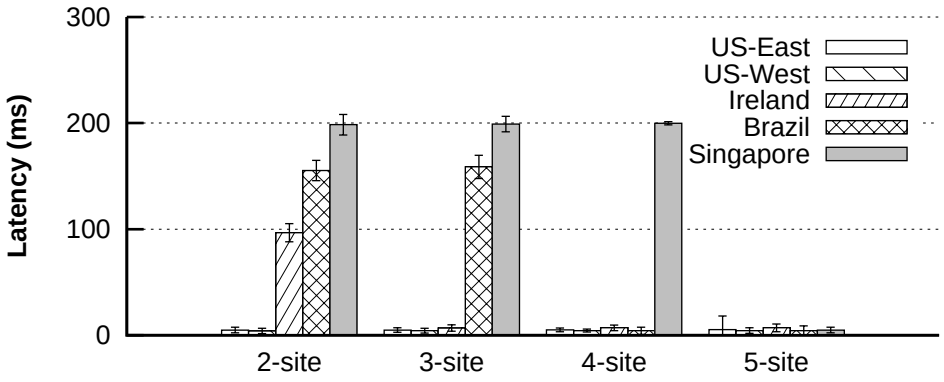
Table 3.3: Average round trip latency and bandwidth between Amazon datacenters (obtained in 2012).

3.7.1 Experimental setup

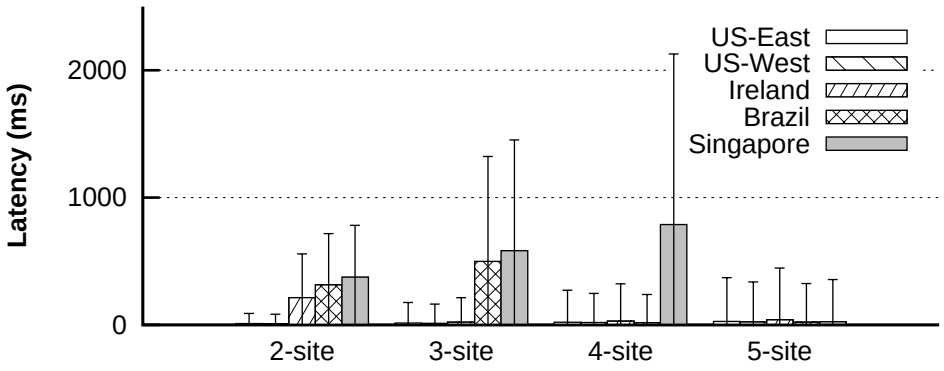
We run experiments on Amazon EC2 [Amaa] using extra large virtual machine instances located in five sites: US east (UE), US west (UW), Ireland (IE), Brazil (BR), and Singapore (SG). Table 3.3 shows the average round trip latency and observed bandwidth between every pair of sites. For experiments with fewer than 5 sites, new sites are added in the following order: UE, UW, IE, BR, SG. Unless otherwise noted, users are evenly distributed across all sites. Each VM has 8 virtual cores and 15GB of RAM. VMs run Debian 6 (Squeeze) 64 bit, MySQL 5.5.18, Tomcat 6.0.35, and Sun Java SDK 1.6. Each experimental run lasts for 10 minutes.

3.7.2 Microbenchmark

We begin the evaluation with a simple microbenchmark designed to stress the costs and benefits of partitioning operations into red and blue sets. Each user issues requests accessing a random record from a MySQL database. Each request maps to a single shadow operation; we say a request is blue if it maps to a blue shadow operation and red otherwise. The offered workload is varied by adjusting the number of outstanding requests per user and the ratio of red and blue requests.



(a) Blue request latency for all users as number of sites increases



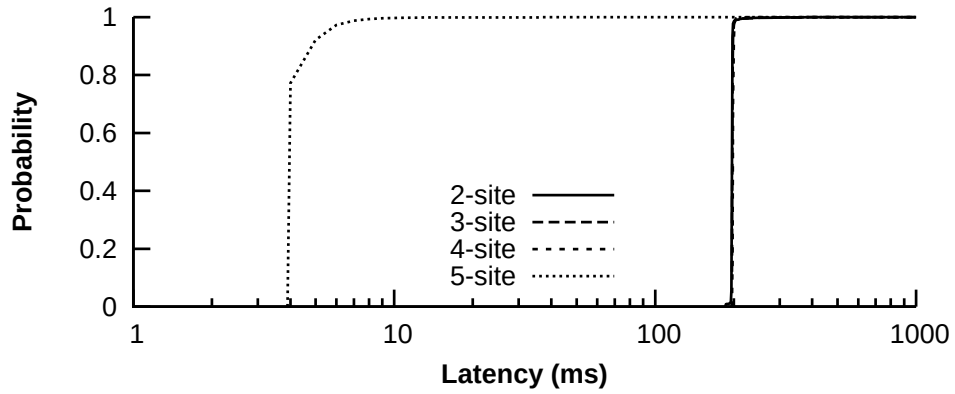
(b) Red request latency for all users as number of sites increases

Figure 3.12: (a) and (b) show the average latency and standard deviation for blue and red requests issued by users in different locales as the number of sites is increased, respectively.

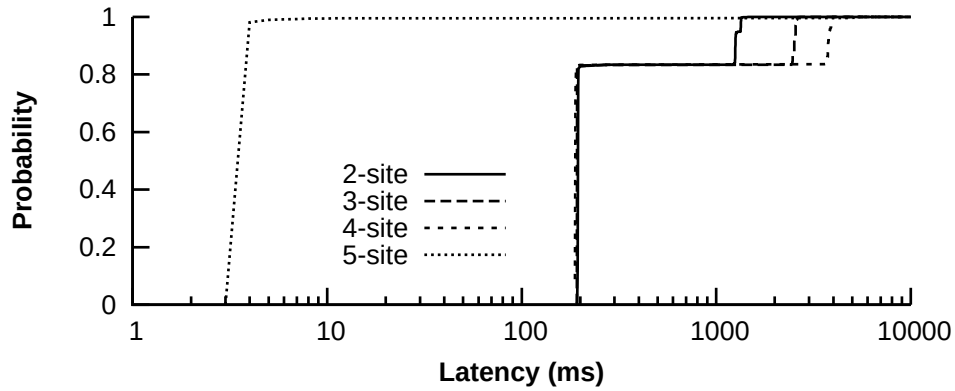
We run the microbenchmark experiments with a dataset consisting of 10 tables each initialized with 1,000,000 records; each record has 1 text and 4 integer attributes. The total size of the dataset is 1.0 GB.

User observed latency

The primary benefit of using Gemini to replicate a service across multiple sites is the decrease in latency from avoiding the intercontinental round-trips as much as possible. As a result, we first explore the impact of RedBlue consistency on user experienced



(a) Blue latency CDF for Singapore users as number of sites increases



(b) Red latency CDF for Singapore users as number of sites increases

Figure 3.13: (a) and (b) show the CDF of latencies for blue and red requests issued by users in Singapore as the number of sites is increased, respectively.

latency. In the following experiments each user issues a single outstanding request at one time.

Figure 3.12(a) shows that the average latency for blue requests is dominated by the latency between the user and the closest site; as expected, average latency decreases as additional sites appear close to the user. For example, with replicas in two sites in US, users at US-East get responses in less than 10 ms, whereas users at Ireland get responses of 100 ms on average, slightly above the round-trip latency of 92 ms presented in Table 3.3. Figure 3.12(b) shows that this trend also holds for red requests. The average latency and standard deviation, however, are higher for red requests than

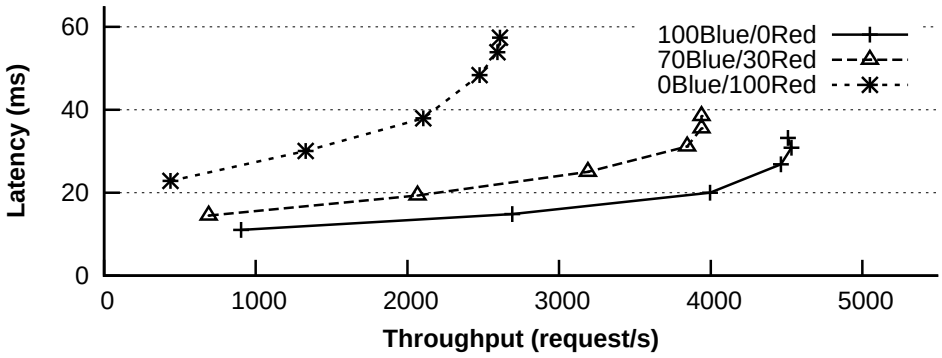


Figure 3.14: Throughput versus latency graph for a 2 site configuration with varying red-blue workload mixes.

for blue requests. This is because red shadow operations can be as fast as blue ones if their primary site holds the unique red token, but will be much slower if the site does not have that privilege.

To understand this effect, we plot in Figures 3.13(a) and 3.13(b) the CDFs of observed latencies for blue and red requests, respectively, from the perspective of users located in Singapore. The observed latency for blue requests tracks closely with the round-trip latency to the closest site. In the $k = 2$ through $k = 4$ site configurations, four red requests from a user in Singapore are processed at the closest site during the one second in which the closest site holds the red token; every fifth request must wait $k - 1$ seconds for the token to return. In the 5 site configuration, the local site also becomes a replica of the service and, therefore, a much larger number of requests (more than 300) can be processed while the local site holds the red token. This changes the format of the curve, even though the request issued immediately after the red token is released also needs to wait four seconds for the token to return.

Peak throughput

We now shift our attention to the throughput implications of RedBlue consistency. Figure 3.14 shows a throughput-latency graph for a 2 site configuration and three workloads:

100% blue, 100% red, and a 70% blue/30% red mix. The different points in each curve are obtained by increasing the offered workload, which is achieved by increasing the number of outstanding requests per user. For the mixed workload, users are partitioned into blue and red sets responsible for issuing requests of the specified color and the ratio is a result of this configuration.

The results in Figure 3.14 show that increasing the ratio of red requests degrades both latency and throughput. In particular, the two-fold increase in throughput for the all blue workload in comparison to the all red workload is a direct consequence of the coordination (not) required to process red (blue) requests: while red requests can only be executed by the site holding the red token to process, every site may independently process blue requests. The peak throughput of the mixed workload is proportionally situated between the two pure workloads.

3.7.3 Case studies: TPC-W and RUBiS

Our microbenchmark experiments indicate that RedBlue consistency instantiated with Gemini offers latency and throughput benefits in geo-replicated systems with sufficient blue shadow operations. Next, we evaluate Gemini using TPC-W and RUBiS.

Configuration and workloads

In all case study experiments a single site configuration corresponds to the original unmodified code with users distributed amongst all five sites. Two through five site configurations correspond to the modified RedBlue consistent systems running on top of Gemini. When necessary, we modified the provided user emulators so that each user maintains k outstanding requests and issues the next request as soon as a response is received.

	Blue	Red	read-only	update
TPC-W shop	99.2	0.8	85	15
TPC-W browse	99.5	0.5	96	4
TPC-W order	93.6	6.4	63	37
RUBiS bid	97.4	2.6	85	15

Table 3.4: Proportion of blue and red shadow operations and read-only and update requests in TPC-W and RUBiS workloads at runtime.

TPC-W. TPC-W [con02] defines three workload mixes differentiated by the percentage of client requests related to making purchases: browsing (5%), shopping (20%), ordering (50%). The dataset is generated with the following TPC-W parameters: 50 EBS and 10,000 items.

RUBiS. RUBiS defines two workload mixes: browsing, exclusively comprised of read-only interactions, and bidding, where 15% of user interactions are updates. We evaluate only the bidding mix. The RUBiS database contains 33,000 items for sale, 1 million users, 500,000 old items and is 2.1 GB in total.

Prevalence of blue and red shadow operations

Table 3.4 shows the distribution of blue and red shadow operations during the execution of the TPC-W and RUBiS workloads. The results show that TPC-W and RUBiS exhibit sufficient blue shadow operations for it to be likely that we can exploit the potential of RedBlue consistency.

User observed latency

We first explore the per request latency for a set of exemplar blue and red requests from TPC-W and RUBiS. For this round of experiments, each site hosts a single user issuing one outstanding request to the closest site.

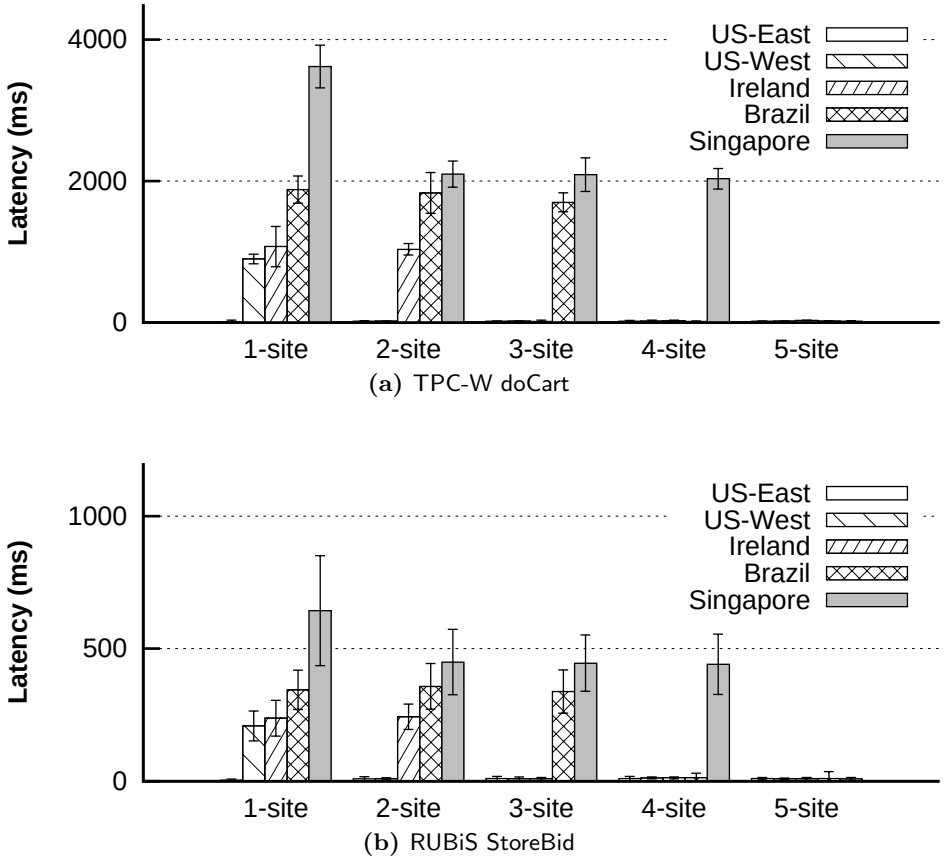


Figure 3.15: Average latency for selected TPC-W and RUBiS user interactions. Shadow operations for doCart and StoreBid are always blue.

From TPC-W we select doBuyConfirm (discussed in detail in Section 3.6.1) as an exemplar for red requests and doCart (responsible for adding/removing items to/from a shopping cart) as an exemplar for blue requests; from RUBiS we identify StoreBuyNow (responsible for purchasing an item at the buyout price) as an exemplar for red requests and StoreBid (responsible for placing a bid on an item) as an exemplar for blue requests. Note that doBuyConfirm and StoreBid can produce either red or blue shadow operations; in our experience they produce red shadow operations 98% and 99% of the time respectively.

Figures 3.15(a) and 3.15(b) show that the latency trends for blue shadow operations are consistent with the results from the microbenchmark—observed latency is directly

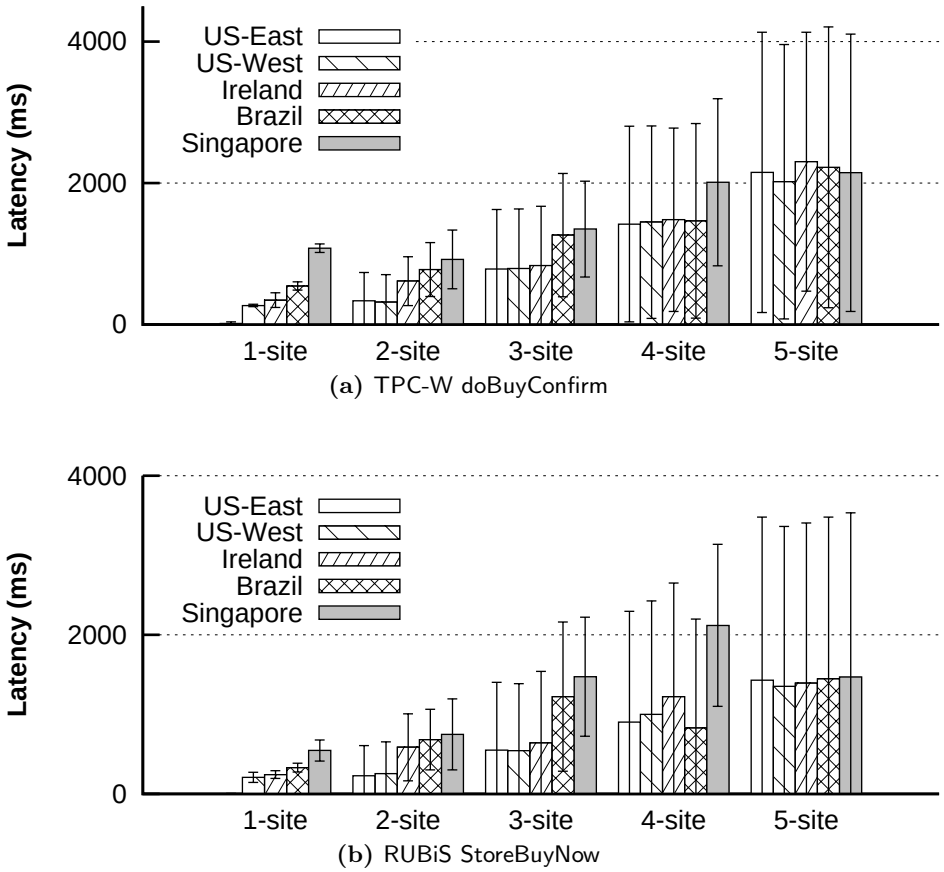
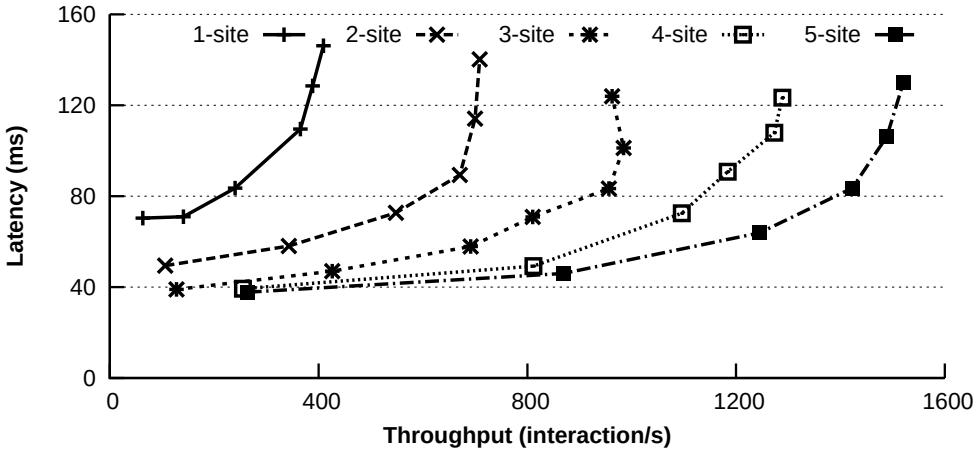


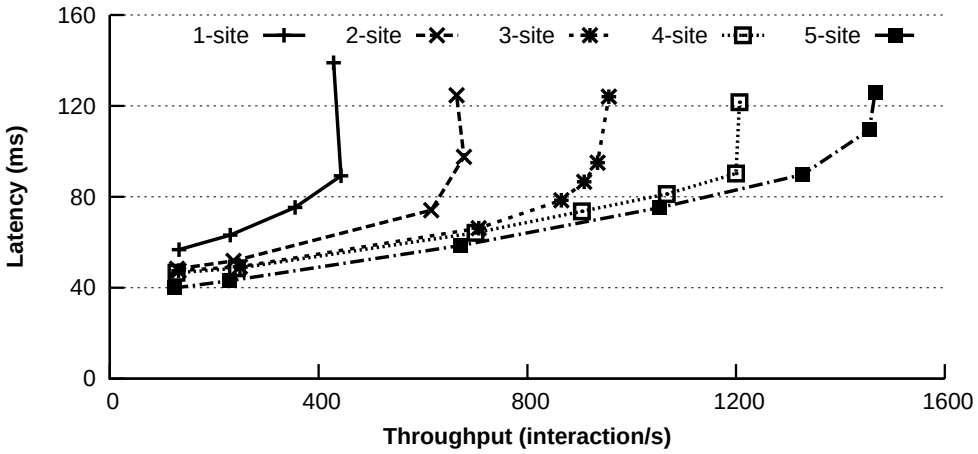
Figure 3.16: Average latency for selected TPC-W and RUBiS user interactions. Shadow operations for doBuyConfirm and StoreBuyNow are red 98% and 99% of the time respectively.

proportional to the latency to the closest site. The raw latency values are higher than the round-trip time from the user to the nearest site because processing each request involves sending one or more images to the user.

For red requests, Figures 3.16(a) and 3.16(b) show that latency and standard deviation both increase with the number of sites. The increase in standard deviation is an expected side effect of the simple scheme that Gemini uses to exchange the red token and is consistent with the microbenchmark results. Similarly, the increase in average latency is due to the fact that the time for a token rotation increases, together with the fact that red requests are not frequent enough that several cannot be slipped in during the same



(a) TPC-W shopping mix



(b) RUBiS bidding mix

Figure 3.17: Throughput versus latency for the TPC-W shopping mix and RUBiS bidding mix. The 1-site line corresponds to the original code; the 2/3/4/5-site lines correspond to the RedBlue consistent system variants.

token holding interval. We reiterate that the token passing scheme used by Gemini is simple and we leave as future work the implementation of a more sophisticated scheme like Paxos [Lam98] for regulating red shadow operations.

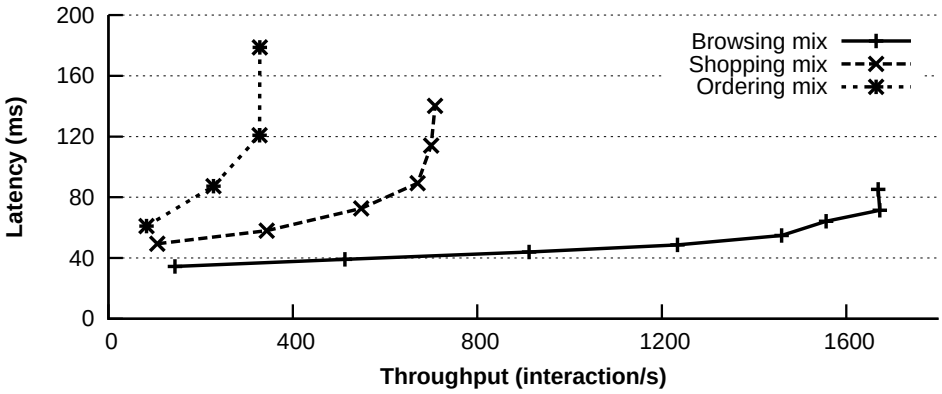


Figure 3.18: TPC-W: Throughput vs. latency graph for TPC-W with Gemini spanning two sites when running the three workload mixes.

Peak throughput

We now shift our attention to the throughput afforded by our RedBlue consistent versions of TPC-W and RUBiS, and how it scales with the number of sites. For these experiments we vary the workload by increasing the number of outstanding requests maintained by each user. Throughput is measured according to interactions per second, a metric defined by TPC-W to correspond to user requests per second.

Figure 3.17 shows throughput and latency for the TPC-W shopping mix and RUBiS bidding mix as we vary the number of sites. In both systems, increasing the number of sites increases peak throughput and decreases average latency. The decreased latency results from situating users closer to the site processing their requests. The increase in throughput is due to processing blue and read-only operations at multiple sites, given that processing their side effects is relatively inexpensive. The speedup for a 5 site Gemini deployment of TPC-W is 3.7x against the original code for the shopping mix; the 5 site Gemini deployment of RUBiS shows a speedup of 2.3x.

Figure 3.18 shows the throughput and latency graph for a two site configuration running the TPC-W browsing, shopping, and ordering mixes. As expected, the browsing mix, which has the highest percentage of blue and read-only requests, exhibits the highest

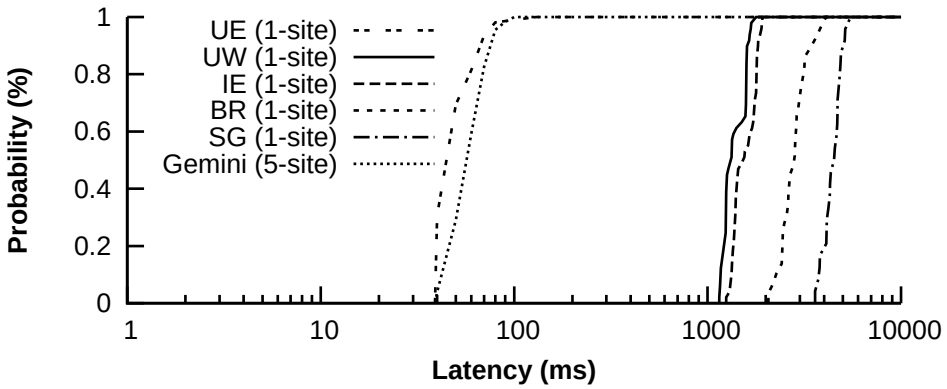


Figure 3.19: User latencies CDF for the addFriend request in single site Quoddy and 5-site Gemini deployments.

peak throughput, and the ordering mix, with the lowest percentage of blue and read-only requests, exhibits the lowest peak throughput.

3.7.4 Case study: Quoddy

Quoddy differs from TPC-W and RUBiS in one crucial way: it has no red shadow operations. We use Quoddy to show the full power of RedBlue geo-replication.

Quoddy does not define a benchmark workload for testing purposes. Thus we design a social networking workload generator based on the measurement study of Benevenuto et al. [BRCA09]. In this workload, 85% of the interactions are read-only page loads and 15% of the interactions include updates, e.g., request friendship, confirm friendship, or update status. Our test database contains 200,000 users and is 2.6 GB in total size.

In a departure from previous experiments, we run only two configurations. The first is the original Quoddy code in a single site. The second is our Gemini based RedBlue consistent version replicated across 5 sites. In both configurations, users are distributed in all 5 regions.

Figure 3.19 shows the CDF of user experienced latencies for the addFriend operation. All Gemini users experience latency comparable to the local users in the original Quoddy deployment; a dramatic improvement for users not based in the US East region.

	TPC-W shopping mix		RUBiS biding mix	
	Original	Gemini	Original	Gemini
Throughput (interaction/s)	409	386	450	370
Average latency (ms)	14	15	6	7

Table 3.5: Performance comparison between the original code and the Gemini version for both TPC-W and RUBiS within a single site.

The significantly higher latencies for remote regions are associated with the images and Javascript files that Quoddy distributes as part of processing the addFriend request.

3.7.5 Gemini overheads

Gemini is a middleware layer that interposes between the applications that leverage RedBlue consistency and a set of database systems where data is stored. We evaluate the performance overhead imposed by our prototype by comparing the performance of a single site Gemini deployment with the unmodified TPC-W and RUBiS systems directly accessing a database. For this experiment we locate all users in the same site as the service.

Table 3.5 presents the peak throughput and average latency for the TPC-W shopping and RUBiS bidding mixes. The peak throughput of a single site Gemini deployment is between 82% and 94% of the original and Gemini increases latency by 1ms per request.

3.8 Limitations and future work

Although RedBlue consistency significantly succeeded in making our example applications fast, i.e., uniformly low user observed latency and high system throughput, without sacrificing their targeted behavior, there are still several points, which we either address in subsequent chapters or leave as future work.

First, RedBlue consistency offers a coarse-grained classification scheme, which can lead to a conservative labeling result for applications that require more consistency

levels other than weak and strong consistency. We address this limitation by introducing a generic consistency model providing us with more flexibility to express consistency requirements in Chapter 5.

Second, the adoption of RedBlue consistency requires programmers to make effort to write shadow operations, to apply changes to the original code, and to reason about operation commutativity and invariant violation in the presence of parallelism. Without the support of automatic tools, the manual work can be error-prone and does not scale, as the code base increases. We address this limitation by building SIEVE in Chapter 4, which combines operational transformation and programming language techniques to provide an automatic and provably correct solution.

Third, the simple token passing scheme for offering strong consistency is not efficient and fault tolerant. At each point of time, only one site can admit its red shadow operations to the global RedBlue order when this site is possessing the red token, while the remaining sites are waiting. This leads to a high latency for user requests, and would cause the whole system to stop executing this type of operations if the site where the red token stays crashes. To address this limitation, we leave as future work the implementation of Paxos for serializing all red shadow operations across sites.

Fourth, using logical clocks might introduce false causal dependencies among operations. As every site increases its own entry when assigning monotonic timestamps to all its receiving operations, these operations become totally ordered, which, provided that some of them are blue, is not necessary. This might limit the amount of concurrency within a site, so we leave to future work an analysis of the impact of the usage of logical clock on scalability.

3.9 Summary

In this chapter, we presented a principled approach to building geo-replicated systems that are fast as possible and consistent when necessary. Our approach to addressing the

tension between running operations locally as often as possible but without sacrificing important application properties, namely state convergence and invariant preservation, hinges on three major technical contributions: (1) a novel notion of RedBlue consistency allowing both strongly consistent (red) operations and causally consistent (blue) operations to coexist, (2) a concept of shadow operation increasing the coverage of blue operations, and (3) a labeling methodology for precisely determining which operations to be assigned which consistency level. We implemented a distributed storage system called Gemini that executes and replicates red and blue operations, and used it along with our labeling conditions to run three existing web applications, namely TPC-W, RUBiS and Quoddy, under RedBlue consistency. Experimental results show that RedBlue consistency significantly improves the performance of geo-replicated systems.

4 Automatic consistency level assignment

In this chapter, we describe the design, implementation, and evaluation of SIEVE, the first tool to automate the choice of consistency levels in a replicated system. SIEVE performs a combination of static and dynamic analysis, offline and at runtime, to determine when it is necessary to use strong consistency to preserve application-specific invariants and when it is safe to use causally consistent commutative replicated data types (CRDTs).

This chapter is organized as follows. We first outline the motivation and contributions of SIEVE in Section 4.1. Then we discuss the most relevant related work in Section 4.2. We present the design rationale of SIEVE, and detail its implementation in Sections 4.3, 4.4, 4.5. Section 4.6 describes the case study applications, the experience on applying SIEVE to these applications, and the corresponding experimental results. Finally, we discuss SIEVE's limitations in Section 4.7 and conclude this chapter in Section 4.8.

4.1 Motivation and contributions

As mentioned in Chapter 1, the providers of planetary-scale services—such as Google [Gooa], Amazon [Amab], or Facebook [Fac] face an inherent tension between improving performance and maintaining targeted consistency semantics. In order to resolve this tension, in Chapter 3, we presented the RedBlue consistency framework, which offers the choice between executing an operation under a strong or a weak consistency

model, and the methodology for increasing the safe usage of weak consistency. As shown in Section 3.6 in Chapter 3, adapting existing applications to RedBlue consistency consists of the following two manual tasks. First, one must transform every application operation into a generator and a set of commutative shadow operations, each of which corresponds to a distinct side effect. Second, one must correctly identify which shadow operations may break application invariants, and label them appropriately so that they execute under strong consistency. Although our experience shows that modifying benchmark applications to be RedBlue consistent is not difficult, in practice, as the code base increases, this manual work can become very challenging and error-prone. This is because it imposes on the application programmer the non-trivial burden of (a) figuring out side effects of every code path in the original operations; (b) implementing shadow operations and verifying whether any pair of them commutes; and (c) understanding the semantics of each shadow operation to determine if it meets the properties for safe execution under weak consistency.

In this chapter, to ease this burden on the programmer, we present SIEVE, the first tool (to the best of our knowledge) that automates this adaptation to multi-level consistency such as RedBlue consistency. This tool focuses on an important and widely deployed class of applications, namely Java-based applications with a database backend. Overall, we make the following contributions:

1. **Commutativity transformation.** One of the obstacles for labeling a large number of operations as blue is the fact that not many operations are naturally commuting with all others, as shown in Section 3.3 and 3.6 in Chapter 3. To ensure good performance, SIEVE automatically transforms the side effects of every application operation into their commutative form. To this end, we build on previous work on commutative replicated data types (CRDTs) [SPBZ11b, PMSL09], i.e., data types whose concurrent operations commute, and apply this concept to relational databases. This allows programmers to only specify which particular CRDT

semantics they intend to use by adding a small annotation in the database schema, and SIEVE automatically generates the shadow operation code implementing the chosen semantics.

2. **Efficient labeling.** SIEVE uses program analysis to identify commutative shadow operations that might violate application-specific invariants when executed under weak consistency semantics, and runs them under strong consistency. To make the analysis accurate and lightweight, we divide it into a potentially expensive static part and an efficient check at runtime. The static analysis generates a set of abstract forms (*templates*) that represent the space of possible shadow operations produced at runtime, and identifies for each template a logical condition (*weakest precondition*) under which invariants are guaranteed to be preserved. This information is then stored in a dictionary, which is looked up and evaluated at runtime, to determine whether each shadow operation can run under weak consistency.

3. **Minimal manual intervention.** Unlike previous work, in which either the adoption of new programming models or a significant number of changes to the original source code is needed, using SIEVE, the programmer has to only specify the application invariants that must be preserved and to annotate a small amount of semantic information about how to merge concurrent updates, while keeping the application code base unchanged.

We evaluate SIEVE using TPC-W and RUBiS. Our results show that it is possible to achieve the performance benefits of weakly consistent replication when it does not lead to breaking application invariants without imposing the burden of choosing the appropriate consistency level on the programmer, and with a low runtime overhead.

4.2 Related work

We summarize and compare previous work with SIEVE according to the following categories:

Weak consistency and commutativity. As we saw in Chapter 3, in order to provide users with low latency access to web services, a wide range of their underlying replicated systems have relied on weak consistency levels such as causal consistency[LFKA11]. They produce a reply to the user as soon as the corresponding operation executes in a single replica with respect to physical proximity. The usage of these systems requires a special care, i.e., they must be equipped with procedures for handling conflicts that may arise from concurrent operations. In some systems, such as Bayou [TTP+95], Depot [MSL+11], and Dynamo [DHJ+07], the programmer has to provide application-specific code for merging concurrent versions. Other systems, such as Cassandra [LM10], COPS [LFKA11], Eiger [LFKA13] and ChainReaction [ALaR13], use a simple last-writer-wins strategy for merging concurrent versions. This simple strategy may, however, lead to lost updates.

Some systems have explored using operation commutativity to guarantee that all replicas converge to the same state, regardless of operation execution order. For example, Preguiça et al. and Shapiro et al. propose CRDTs (commutative or conflict-free replicated data types), a set of abstract data types whose operations commute in presence of concurrency[PMSL09, SPBZ11b]. More recently, Walter [SPAL11] includes a single pre-defined commutative data type, *cset*, which could be seen as an appreciation of the previous CRDT work. Commutative operations that implement variants of CRDTs can also be used in different frameworks such as Lazy replication [LLSG92], RedBlue consistency [LPC+12], Generalized-Paxos [Lam05], and Generic-Broadcast [PS99], for supporting unordered execution of these operations and hence making the corresponding systems or protocols more scalable.

The major drawback of these above systems is that operation commutativity is achieved at a cost, i.e., by either modifying existing application code or adopting a new programming model. Unlike these systems, SIEVE instead offers the programmer a CRDT library and automatically generates commutative shadow operations that encode side effects of every application operation at runtime, requiring only a small amount of CRDT annotations specifying the merging semantics. This automation eases the burden on the programmer and eliminates errors of implementing this semantics, from application to application.

Classification for multi-level consistency. SIEVE is built on top of the RedBlue consistency model, in which operations execute under either strong or weak consistency. The primary goal of SIEVE is to automatically assign appropriate consistency levels to various operations so that state convergence and invariant preservation are ensured despite having weakly consistent replication. The consistency level assignment problem has been studied in many recent multi-level consistency proposals. For example, relying on a probabilistic model, consistency rationing [KHAK09] associates different consistency levels with different states, instead of operations, and allows states to switch from one level to another at runtime. Unlike this approach, we partition operations into strong and causal consistency groups. Pileus is a replicated key-value store, which trades off between consistency and latency requirements of read-only operations via consistency-based service level agreements (SLAs) defined by the user [TPK+13]. Different than Pileus, SIEVE does not restrict operation types. In addition, both RedBlue consistency [LPC+12] and I-confluence [BFF+14] define conditions that operations must meet in order to run under weak consistency, i.e., without coordination. We build on this line of work and extend it so that an automatic tool, and not the programmer, is responsible for determining whether the operations meet these conditions.

Automation. To free programmers from manually making choices of consistency levels, some researchers have attempted to apply program analysis techniques to reason about the consistency requirements of real applications. Alvaro et al. [ACHM11, ACHM14] identify code locations that need to inject coordination to ensure target consistency semantics, while Zhang et al. [ZPZ+13] inspect read/write conflicts across all operations. However, they merely focus on commutativity and ignore application invariants, which are very important and taken into account by our solution. Instead of a fully static solution, we offer a dynamic and optimistic classification by combining a static analysis of computing weakest preconditions for shadow operations and a runtime evaluation to determine operations to be strongly consistent if the corresponding conditions evaluate to `FALSE`.

Very recently, the concept of warranties imposes a set of time-limited invariant-related assertions over shared objects in a replicated system, and allows transactions to commit without coordination if the relevant assertions are still valid [LMA+14]. Compared with warranties, preconditions in SIEVE are logical formulas defined over parameters of shadow operations rather than system state. As a result, SIEVE is able to always perform condition checks locally, while warranties have to invalidate assertions when updates are replicated or the expiration time reaches, and to delay updates for making read-only transactions fast. The work from Roy et al. [RKF+14] resembles the concept of warranties and presents an algorithm to analyze transaction code for producing warranties. That work is complementary to the goal of SIEVE since we rely on a verification tool, Jahob [Kun07], to determine certain properties (encoded in weakest preconditions) of shadow operations.

Other related work. Commutativity has been explored in other settings to improve performance and scalability – e.g., in databases [Wei88] and in OS design for multi-core systems [CKZ+13]. Program analysis techniques have also been used to identify

commuting code blocks. Aleen et al. [AC09] propose a new approach to find commutative functions automatically at compile time for allowing legacy software to extract performance from many-core architectures. Kim et al. [KR11] used the Jahob verification system to determine commuting conditions under which two operations can execute in different orders. Unlike these two prior solutions that only focus on identifying commutative code blocks, our tool automatically transforms operations by decoupling side effect generation and application, which makes more operations commute [LPC+12], and we also focus on determining invariant safety.

4.3 Overview

This section presents the two main challenges that SIEVE aims to address and its design rationale and architecture.

4.3.1 Design rationale

As described in Chapter 3, adapting applications to RedBlue consistency requires the programmer to generate commutative shadow operations and identify which shadow operations can be blue (weakly consistent) and which must be red (strongly consistent). Thus, to make this model easy-of-use, the goal of SIEVE is to automate these two tasks, to the extent possible.

With regard to the first task, we leverage the rich commutative replicated data type (CRDT) literature [SPBZ11b, PMSL09], which defines a list of data types whose operations commute. CRDTs can be employed to produce commutative shadow operations that converge to identical final states, independent of the order in which they are applied. Shadow operations are thus constructed as a sequence of updates to CRDT data types that commute by construction.

The challenge in developing shadow operations based on CRDTs is that the programmer must explicitly transform the applications to replace all the application state

mutations by calls to the appropriate CRDT object. This involves not only identifying the parts of the programs that encode these actions, but also understanding the catalogue of CRDT structures and choosing the appropriate one. To minimize this programmer intervention, we focus on two-tier architectures that store all of the state that must persist across operations in a database. This gives us two main advantages: (1) We can automatically identify the actions that mutate the state, namely the operations that access the database. (2) We can reduce the user intervention to small annotations in the database data organization regarding how to reconcile concurrent updates to different data items.

The second challenge SIEVE addresses is automatically labeling commutative shadow operations. To this end, for each shadow operation that is generated, we only need to decide whether it is invariant safe, according to the definition in Section 3.4. (Commutativity does not need to be checked since the previous step ensures that shadow operations commute by design.) To automate the classification process, there are two design alternatives that represent two ends of a spectrum: (1) a dynamic solution, which determines at runtime, when the shadow operation is produced, whether that shadow operation meets the invariant safety property, and (2) a fully static solution that determines which combinations of initial operation types, parameters, and initial states they are applied against lead to generating a shadow operation that is invariant safe. The problem with the former solution is that it introduces runtime overheads, and the problem with the latter solution, as we will detail in Section 4.5, is that the static analysis could be expensive and end up conservatively flagging too many operations as strongly consistent.

To strike a balance between the two approaches, we split the labeling into a potentially expensive static part and a lightweight dynamic part. Statically, we generate a set of templates corresponding to different possible combinations of CRDT operations that comprise shadow operations, along with weakest preconditions for each template to be

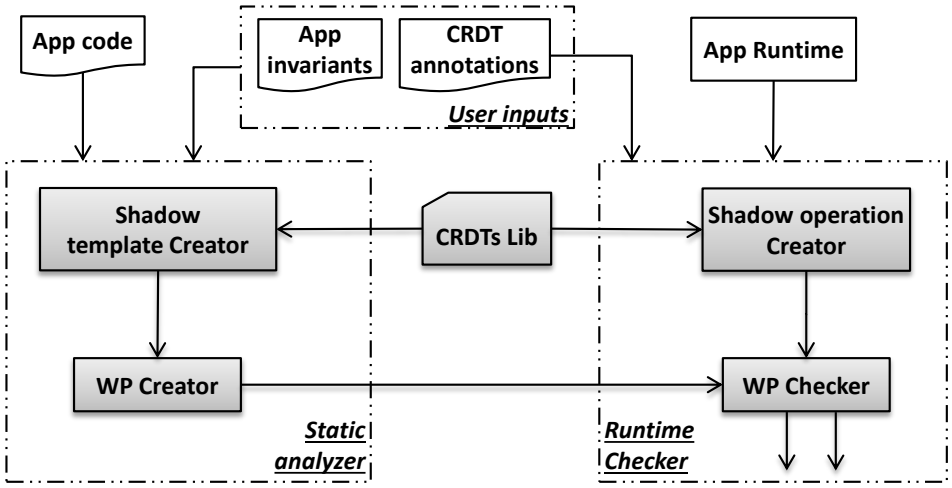


Figure 4.1: Overview of SIEVE. Shaded boxes are system components comprising SIEVE. (WP stands for weakest precondition.)

invariant safe. Then, at runtime, we perform a simple dictionary lookup to determine which template the shadow operation falls into, so that we can retrieve the corresponding weakest precondition and determine whether it is met.

4.3.2 SIEVE architecture

These two main solutions above lead to the high level system architecture depicted in Figure 4.1. The application programmer writes the *application code* as a series of transactions written in Java, which access a database for storing persistent state. Beyond the application code, the only additional inputs that the programmer needs to provide are *CRDT annotations* specifying the semantics for merging concurrent updates and a set of application-specific *invariants*. The static analyzer then creates *shadow operation templates* from the code of each transaction, where these templates represent different sequences of invocations of functions in a *CRDT library*. The analyzer also computes the *weakest preconditions* required for each template to be invariant safe.

At runtime, application servers run both the Java logic and the *runtime checker*, and interact with a database server (not shown in the figure) and the replication tier (not

shown in the figure). While executing a transaction, the application server runs the generator operation and accumulates its side effects in a *shadow operation creator*. When commit is called by the generator, instead of directly committing side effects to the database, the *creator* generates a shadow operation consisting of a sequence of invocations from the CRDT library. This shadow operation is then fed to the *weakest precondition checker* to decide which static template it falls into, and what is the precondition required for the operation to be invariant safe, which allows the runtime to determine how to label the operation. The labeled shadow operation is then fed to the replication system implementing multi-level consistency. In the following sections we further detail the design and implementation of the main components of this architecture.

4.4 Generating shadow operations

This section covers how we automate the conversion of application code into commutative *shadow operations*.

4.4.1 Leveraging CRDTs

We leverage several observations and technologies to achieve a sweet spot between the need to capture the semantics of the original operation when encoding its side effects and the desire to minimize the amount of programmer intervention. First, we observe that many applications are built under a two-tier model, where all the persistent state of the service is stored in a relational database accessed through SQL commands. Second, we leverage CRDTs [PMSL09], which construct operations that commute by design by encapsulating all side effects into a library of commutative operations.

These two concepts allow us to achieve commutativity while overcoming the disadvantage of CRDTs, namely the need to adapt applications. This is because the state of two-tier applications is accessed through the narrow SQL interface, and therefore we can focus exclusively on adapting the implementation of SQL commands to access a CRDT.

SQL type	CRDT	Description
FIELD*	LWW	Use last-writer-wins to solve concurrent updates
	NUMDELTA	Add a delta to the numeric value
TABLE	AOSET, UOSET, AUSET, ARSET	Sets with restricted operations (add, update, and/or remove). Conflicting operations are logically executed by timestamp order.

Table 4.1: Commutative replicated data types (CRDTs) supported by our type system. * FIELD covers primitive types such as integer, float, double, datetime and string.

In particular, database tables can be seen as a set of tuples, and therefore all the calls in the original operation to add or remove tuples in a table can be replaced in the shadow operation with a CRDT set add or remove, which, in turn, is implemented on top of the database.

However, it is impossible to completely remove the programmer from the loop, due to the choice of which CRDT to use for encoding appropriate merging semantics. For instance, when an integer field of a tuple is written to in a SQL update command, the programmer could have two different intentions in terms of what the update means and how concurrent updates should be handled: (1) the update can represent a delta to be added or subtracted from the current value (e.g., when updating the stock of a certain item), in which case all concurrent updates should be applied possibly in a different order at all replicas to ensure that no stock changes are lost, or (2) it can be overwriting an old value with a new value (e.g., when updating the year of birth in a user profile), in which case an order for these updates should be arbitrated, and the last written value should prevail. Even though both strategies ensure state convergence, their semantics differ significantly. For example, the second strategy leads to a final state that does not reflect the effects of all update operations.

```
@AUSET CREATE TABLE exampleTable (
    objId INT(11) NOT NULL,
    @NUMDELTA objCount INT(11) default 0,
    @LWW objName char(60) default NULL,
    PRIMARY KEY (id)
) ENGINE=InnoDB
```

Figure 4.2: Annotated table definition schema.

Since the appropriate merging strategy is application-specific, the programmer has to convey this decision. To minimize this input, we only require the programmer to select the appropriate merging strategy (i.e., the adequate CRDT type) to encode these operations rather than programming these CRDT transformations or changing the code of each operation. In more detail, we provide programmers a number of CRDT types (shown in Table 4.1), and they should declare which types to use on a per-table and per-attribute basis. These types form two categories: field, which is the smallest component of a record and defines its commuting update operation in the presence of concurrency, and set, which is a collection of such records plus the support for commutative appending or removing. Programmers only need to annotate the data schema with the desired CRDT type using the following annotation syntax:

$$@[CRDTName][TableName|DataFieldName] \tag{4.1}$$

Figure 4.2 presents a sample annotated SQL table creation statement. We assign `exampleTable` the type `AUSET` (Append-Update Set), a CRDT set that only allows append and update operations, thus precluding the concurrent insertion and deletion of the same item (less restrictive CRDT sets also exist). The field `objCount` associated with `NUMDELTA` always expects a delta value to be added or subtracted to its current value. By default, if no annotations are provided, we conservatively mark the corresponding table or field to be read-only.

4.4.2 Runtime creation of shadow operations

With these schema annotations in place, it is easy to generate commutative shadow operations at runtime. The idea is to invoke the original operation upon the arrival of a new user request (as would happen in a system that does not make use of shadow operations) but with the difference that all the calls to execute commands in the database are intercepted by a modified JDBC driver that builds the sequence of CRDT operations that comprise the shadow operation as the original operation progresses. Furthermore, using the schema annotations, SIEVE maps each database update to an appropriate merge semantics and replaces the operations on a certain table or field with the appropriate operations over the corresponding CRDT type.

For instance, to create a shadow operation for a transaction that updates `objCount` in Figure 4.2, when an update is invoked, we first query the old value s, and then, given the new value s', we compute a `delta` by subtracting s from s'. Finally, we use `delta` and the primary key pk of the corresponding object to parameterize a CRDT operation that reads the tuple identified by pk and then adds `delta` to it.

Finally, when the original operation issues a commit to the database, the tool outputs a shadow operation containing the accumulated sequence of CRDT operations.

4.4.3 Miscellaneous

In this part, we discuss a few interesting aspects related to the commutativity conversion.

Treatment of non-deterministic SQL statements. In addition to the previously described logic to construct commutative shadow operations, we also eliminate all sources of non-determinism that might exist in the operation code, which could lead to state divergence when executing the shadow operations. This is achieved by transparently encoding deterministic values into CRDTs whenever sources of non-determinism are used. For example, if some transaction relies on the current time, we simply replace the call

that provides this value, with a static value obtained at the creation of the shadow operation. Some queries update or delete records when these records match a certain condition rather than specifying the primary keys. With regard to this case, we first perform a select to fetch a list of primary keys from the set of records matching the condition, and then encode the primary keys along the updating semantics.

Annotation suggestion. We found that it would be possible to effectively recommend a few commutative types to programmers, by statically analyzing the application code, or the SQL queries, or both. For example, if a data field is always modified by an assignment statement, then a LWW solution may be suitable for it. If a data field is manipulated via either addition or subtraction, then the $NUMDELTA$ is a good CRDT candidate. Furthermore, if a table is never modified by a delete query, then we would suggest it to be tagged as an AUSET (Append-Update set). We leave this optimization to our future work.

4.5 Classification of shadow operations

In this section, we first discuss the main challenges and design choices of making SIEVE correctly label commutative shadow operations either strongly or weakly consistent in Section 4.5.1. Then, we expand in Section 4.5.2 on how to leverage a static analysis to enable an efficient and less pessimistic runtime labeling. Finally, we explain how the runtime component takes advantage of the information generated in the static phase in Section 4.5.3.

4.5.1 Overview

As mentioned in Section 4.3, a possible solution would be to statically compute the combinations of operation types, parameters, and initial states that generate invariant safe shadow operations. This can be done by performing a weakest precondition

computation—a common technique from Programming Languages and Verification research for which some tool support already exists—which enables us to statically compute, given the code of each operation and the application-specific invariants (which are inserted as postconditions), a precondition over the initial state and operation parameters that ensures the invariant safety property. The advantage of using the weakest precondition concept is to allow us to have a dynamic and optimistic labeling result at runtime, i.e., shadow operations need to execute with coordination only if the respective precondition evaluates to `FALSE`.

Despite the above benefit, however, the weakest precondition computation raises the following two important problems. First, there is a scalability problem, which is exemplified by the following hypothetical code for the generator operation, assuming an invariant that the state variable `x` should be non-negative. (For simplicity, we write conventional Java code accessing variable `x` instead of SQL.)

```
void generator(string s) {
  if (SHA-1(s)==SOME_CONSTANT) {
     if (x>=10){
        x -= 10;
     }
  } else
     x +=10;
}
```

The problem with this code is that a weakest precondition analysis to determine which values of `s` lead to a negative (non-invariant safe) delta over `x` is computationally infeasible, since it amounts to inverting a hash function. As such, we would end up conservatively labeling the shadow operations generated by this code as red (i.e., the weakest precondition would be `FALSE`). Even though this is an extreme example, it

highlights the difficulty in handling complex conditions over the input, even when the side effects are simple. In particular, there are only three patterns of side effects produced by this generator, regardless of the inputs provided to the generator operation. Based on this observation, to simplify the weakest precondition computation and to minimize the space of strongly consistent shadow operations, our static analysis is conducted over the set of possible sequences of CRDT operations that can be generated, which is the same as saying that we analyze all possible shadow operations. We call each possible sequence of shadow operations that can be generated by a given generator operation a `template`. In the above example, there are only three sequences of shadow operations that can be generated: the empty sequence, adding a delta of 10, and adding a delta of -10. From these three possible sequences, only a delta of -10 leads to a weakest precondition of `FALSE`, i.e., is always non-invariant safe. The remaining ones have a weakest precondition of `TRUE`. (Note that in the general case, the precondition can be parameterized over the parameters of shadow operations.)

The second challenge that needs to be overcome is related to handling loops. The generator code in Figure 4.3(a) illustrates that the number of iterations in the loop can be unbounded, which in turn leads to an unbounded number of sequences of CRDT operations in the shadow operation. To abstract this, we could produce a template that preserves the loop structure, such as the one in Figure 4.3(b). However, when computing a weakest precondition over this piece of code, verification tools face a scalability problem, which is overcome by requiring the programmer to specify loop invariants that guide the computation of this weakest precondition [Kun07]. Again, this would represent an undesirable programmer intervention.

To address this challenge, we note that in many cases (including all applications that we analyzed), loop iterations are independent, in the sense that the parts of the state modified in each iteration are disjoint. Again, this is illustrated by the example in

```
1  Begin transaction;
2  for(int i = 0; i < x.length; i++){
3    if(x[i] < 100)
4      x[i]++;
5    else
6      x[i] = -100
7  }
8  End transaction;
```

(a) Original code

```
1  func txnShadow(int[] obsX, int[] deltaA){
2      for (i = 0; i < obsX.length; i++){
3          if(obsX[i] < 100)
4              CRDT_x[i].applyDelta(deltaA[i]);
5          else:
6              CRDT_x[i].applyDelta(deltaA[i]);
7      }
8  }
```

(b) Possible corresponding shadow template

Figure 4.3: Code snippet of a transaction and a possible template for the corresponding shadow operation.

Figure 4.3, where the loop is used to iterate over a set of items, and each iteration only modifies the state of the item being iterated.

This iteration independence property enables us to significantly simplify the handling of loops. In particular, when generating the weakest precondition associated with a loop, we only have to consider the CRDT operations invoked in two sets of control flow paths, one where the code within the loop is never executed, and another with all possible control flow paths when the loop is executed and iteration repetitions are eliminated. (We will explain in detail how to handle loops using an example in the following subsection.) This condition can then be validated against each individual iteration of the loop at runtime and, given the independence property, this validation will be valid for the entire loop execution.

Sequential path	Description
$2 \cdot 3 \cdot 4 \cdot 2$	only if
$2 \cdot 3 \cdot 6 \cdot 2$	only else
$2 \cdot 3 \cdot 4 \cdot 2 \cdot 3 \cdot 6 \cdot 2$	else follows if
$2 \cdot 3 \cdot 6 \cdot 2 \cdot 3 \cdot 4 \cdot 2$	if follows else

Table 4.2: Distinct sequential paths obtained for the transaction in Figure 4.3(a).

In our current framework, the iteration independence property is validated manually. In all our case study applications, it was straightforward to see that this property was met at all times. We leave the automation of this step as future work.

4.5.2 Generating templates and weakest preconditions

Instead of reasoning about the generator code, our analysis is simplified by reasoning about the side effects of each code path taken by the generator operation. Furthermore, we can cut the number of possible code paths by eliminating code sections that are repeated due to loops.

To perform this analysis, we require an algorithm for extracting the set of sequential paths of a transaction and eliminating loop repetition. The high level idea of this algorithm is to split branch statements and replace loops with all non-repeating combinations of branches that can be taken within a loop. The algorithm works as follows. First, for every transaction, we create its path abstraction, which is a regular expression encoding all control flow information within that transaction. In the example shown in Figure 4.3(a), its path abstraction is $2 \cdot (3 \cdot (4|6) \cdot 2)*$, where numbers represent the statement identifiers shown in the figure, $\cdot$ concatenates two sequential statements, $|$ is a binary operator that indicates that the statements at its two sides are in alternative branches, and $*$ represents repetition within a loop. Second, we recursively apply the following two steps to simplify a path abstraction until it is sequential (i.e., no $*$ and $|$). For a path abstraction containing $*$, we create two duplicated abstractions, where one excludes the entire loop, and the other simplifies the loop into its body. For a path

abstraction containing the operator |, we create two duplicated path abstractions, where one excludes the right operand and the other excludes the left operand. Additionally, if such | is affected by a $*$, then we have to create another path abstraction combining both alternatives, i.e., where the if and the else sides are executed sequentially.

In the previous example, the set of sequential paths that is produced is shown in Table 4.2. By ignoring the read-only path where the loop is not executed, we only consider four cases, namely only the if or the else path, and the two sequences including both if and else. Because of the loop independence property, these cases are able to capture all relevant sequences of shadow operations. Note that we would only require considering one of the two orderings for the if and the else code within the loop, since their side effects commute, but taking both orderings into account simplifies the runtime matching of an execution to its corresponding path.

Given a set of sequential paths for a transaction, creating shadow operation templates become straightforward. For each path, we collect a sequence of statements specified by the identifiers in the abstraction from the corresponding control flow graph. Then, we translate every database function call into either a CRDT operation by following the instructions stated in Section 4.4, or a no-op operation (for read-only queries). Finally, all these CRDT operations are packed into a function, which denotes the shadow operation template. These CRDT operations are parameterized by their respective arguments, and the static analysis computes a weakest precondition over these arguments for the template to be invariant safe. We did not devise an algorithm to compute weakest preconditions, instead, we rely on a verification tool called Jahob [Kun07] to do this job. The input fed into Jahob is comprised by a set of templates along with their preconditions and postconditions, which are automatically extracted by the static analysis code.

The final output from the static analysis is a dictionary consisting of a set of $\langle key, value \rangle$ pairs, one for each previously generated shadow operation template, where key is the unique identifier of the template, and $value$ is the weakest precondition for

the template. The unique identifier of the template encodes the set of possible paths using signatures of CRDT operations in a restricted form of regular expression.

4.5.3 Runtime evaluation

Determining if a generated shadow operation is red (strongly consistent) or blue (causally consistent) consists of two steps: (a) fetching its weakest precondition by matching this operation to its corresponding template; and (b) evaluating the condition by substituting variables in the condition with values carried by this operation.

Template/shadow operation matching. At runtime, we must lookup in the dictionary created during the static analysis the template corresponding to each shadow operation as it is produced.

The challenge with performing this lookup is that it requires determining the identifier of the shadow operation corresponding to the path taken, and this must be done by taking into account *only* the operations that are controlled by the runtime, i.e., the CRDT operations. This explains why the dictionary keys consist only of CRDT operations. With the shadow operation identifier, matching the path taken at runtime with the keys present in the dictionary is done efficiently by using a search tree.

Weakest precondition check. Finally, once the weakest precondition for the template that corresponds to a particular shadow operation is retrieved, we evaluate that precondition against the CRDT parameters of the shadow operation. This is achieved by simply replacing the variables in the precondition with their instantiated values and evaluating the final expression to either true or false. If the weakest precondition is evaluated to true the shadow operation is labeled blue, otherwise the shadow operation in labeled red.

After this step, the shadow operation is delivered to the replication layer, which replicates it using different strategies according to its classification, namely red (blue) shadow

operations need (no) coordination. The replication layer we use is the Gemini system (seen in Section 3.5) we built in the RedBlue consistency framework with two following minor changes: (a) make the proxy library use SIEVE instead of manually created shadow operations; and (b) make the data writer code be able to decode and automatically execute generated shadow operations.

4.6 Evaluation

In this section, we report our experience with implementing SIEVE, adapting existing web applications to run with SIEVE, and evaluating these systems.

4.6.1 Implementation

We implemented most of our tool using Java (15k lines of code), and changed parts of the Jahob code to obtain weakest preconditions in OCaml (553 lines of code) [1]. The backend storage system we used was a MySQL database. We used an existing Java parser [jav13] to parse Java files for generating an abstract syntax tree (AST). Finally, we connected our tool to the Gemini replication and coordination system, as presented in Section 3.5, to enable both consistency classification and operation replication. The source code of SIEVE is available at [SIE].

4.6.2 Case studies

To adapt an application to use SIEVE, one has to annotate the corresponding SQL schema with the proper CRDT semantics, specify all invariants, and finally the original JDBC driver must be replaced by the driver provided by SIEVE, to enable SIEVE to intercept interactions between the application and the database.

We applied SIEVE to two web application benchmarks, namely TPC-W [con02] and RUBiS [EJ09]. Both of them simulate an online store and the interactions between users

[1] The lines of code is measured by `cloc` [cod].

App	Invariants
TPC-W	$\forall item \in item\_table.\ \ item.stock \geq 0$
RUBiS	$\forall item \in item\_table.\ \ item.quantity \geq 0$
	$\forall u, v \in user\_table.$ $u.uname = v.uname \implies u = v$

Table 4.3: Application-specific invariants

and the web application. There are two main motivations for selecting these use cases: (1) both have been widely used by the community to evaluate system performance; and (2) both have application-specific invariants that can be violated under causal consistency. We recall the invariants of these two applications in Table 4.3. (In Chapter 3, a social application is evaluated, but it made no sense to include this application because it did not contain any invariants that could be violated under weak consistency.)

For TPC-W, we use `AOSET`, `AUSET`, `UOSET` and `ARSET`, as specified in Table 4.1, to annotate the database tables, no annotations for unmodified attributes, `NUMDELTA` for `stock`, and `LWW` for the remaining attributes. For RUBiS, we annotate its tables with `AUSET` and `AOSET`. We use `NUMDELTA` as annotations for both `quantity` (`stock`) and `numOfBids`, and no annotations or `LWW` for the remaining attributes. For additional details, we refer the interested reader to the examples available in [Li214].

In terms of the time required to do this adaptation, we do not report results for TPC-W as we relied on this use case during the design and development phase of SIEVE. However for the RUBiS use case, the entire process was concluded in only a few hours. An interesting point to highlight is that SIEVE was able to detect inconsistencies between these annotations, namely tagging a table as update-only (`UOSET`) but where the original code contained insert SQL commands against that table. Thus, SIEVE enables programmers to correct mistakes such as type omissions in the SQL schema that are inconsistent with the CRDT annotations.

In both the RedBlue consistency framework (Chapter 3) and SIEVE, the effort we made analyzing application code to determine invariants and merge semantics is unavoid-

able. In the former case, however, we additionally spent a significantly larger amount of time manually implementing merge semantics, and classifying shadow operations by taking into account their properties, for every application. SIEVE eliminates all this manual work and limits human error.

4.6.3 Experimental setup

All reported experiments were obtained by deploying applications on a local cluster, where each machine has 2*6 i7 cores and 48GB RAM, and runs Linux 3.2.48.1 (64bit), MySQL 5.5.18, Tomcat 6.0.35, and Java 1.7.0. The reason we did not include geo-distributed experiments is that we wanted to extensively focus on evaluating various aspects of SIEVE, instead of performance benefits enhanced by RedBlue consistency, which are already shown in Section 3.7.

4.6.4 Experimental results

Our experimental work aims at evaluating both the static analysis component of SIEVE and also the runtime component, which includes a performance comparison between each application using our tool, its unmodified version, and its version under RedBlue consistency where the entire classification is done manually and offline.

Concerning the static analysis component we focus on the following main questions:

1. How long does the static analysis process take to complete?
2. What is the scalability of the static analysis component in relation to the size of the code base?

For the runtime component of SIEVE we focus on the following main questions:

1. Is the runtime classification of shadow operations accurate?
2. What is the (runtime) overhead for adapted applications compared to their stand-alone unmodified counterparts?

Transaction name	#paths	#templates	Transaction name	#paths	#templates
createEmptyCart	1	1	getRelated	1	0
doCart	36	36	getNewProducts	1	0
GetMostRecentOrder	1	0	createNewCustomer	2	2
adminUpdate	4	4	getBestSellers	1	0
getName	1	0	doAuthorSearch	1	0
doSubjectSearch	1	0	GetPassword	1	0
doTitleSearch	1	0	refreshSession	1	1
GetUserName	1	0	getCustomer	1	0
getCart	1	0	doBuyConfirm-A	32	32
doBuyConfirm-B	16	16	getBook	1	0

(a) TPC-W

Transaction name	#paths	#templates	Transaction name	#paths	#templates
ViewUserInfo	6	0	PutComment	10	0
PutBid	14	0	BrowseRegions	5	0
StoreComment	11	3	StoreBid	17	5
BuyNow	7	0	ViewBidHistory	11	0
AboutMe	37	0	ViewItem	10	0
StoreBuyNow	13	6	RegisterItem	59	24
SearchItemsByCategory	20	0	BrowseCategories	13	0
SearchItemsByRegion	20	0	RegisterUser	14	3

(b) RUBiS

Table 4.4: Number of reduced paths and templates generated for each transaction in TPC-W and RUBiS.

3. What are the performance gains obtained through weakly consistent replication using SIEVE?

Static analysis

As mentioned before, taking the application source code and CRDT annotations as input, SIEVE first maps each transaction into a set of distinct paths, and automatically transforms each path into a shadow operation template.

Table 4.4 summarizes the number of paths (excluding loops) and the corresponding number of shadow operation templates that were produced by SIEVE for both TPC-W and RUBiS. For TPC-W, 15 out of the total 20 transactions only exhibit a single path, as the code of these transactions is sequential. The two most complex transactions in this

App	#code	templates		#db code	#specs
		num	#code		
TPC-W	8.3k	92	1554	879	730
RUBiS	9.8k	41	251	477	371

Table 4.5: Overview of the output produced by the static analysis. "db code" refers to the Java classes representing database structures required for computing weakest preconditions.

	WP	Comments
TPC-W	`TRUE`	Not influencing invariants
	`delta` ≥ 0	Non-negative stock
RUBiS	`TRUE`	Not influencing invariants
	`FALSE`	Nickname must be unique
	`delta` ≥ 0	Non-negative quantity
	`quantity` ≥ 0	Non-negative quantity (new item)

Table 4.6: Weakest preconditions (WP)

use case are `doBuyConfirm` and `doCart`, which are associated with the user actions of shopping and purchasing. In contrast, most transactions in RUBiS have a more complex control flow, which generated a larger number of possible execution paths.

Note that the majority of transactions in both use cases do not lead SIEVE to produce any template. This happens when the transactions are read-only, and therefore do not have side effects. Additionally, in TPC-W every path in an update transaction generates a shadow operation template, since system state is always modified. However, this is not true in RUBiS, because its code verifies several conditions, some of which lead to a read-only transaction.

As shown in Table 4.5, the execution of SIEVE generated a total of 92 and 41 shadow operation templates for TPC-W and RUBiS, respectively. In addition to these templates, our tool also generates automatically a set of Java classes that represent database data structures, which are necessary for computing weakest preconditions.

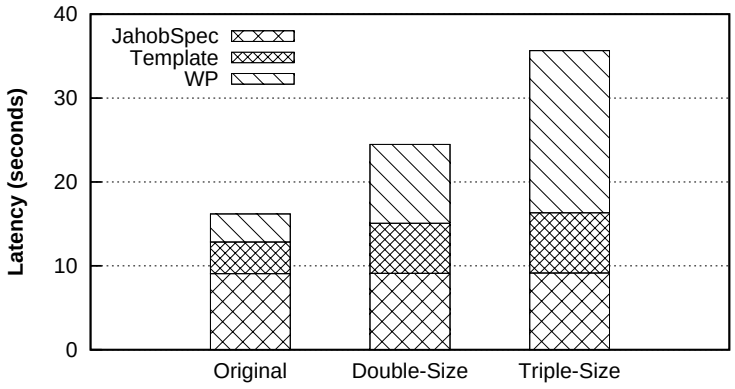
Table 4.6 shows a full list of the different weakest preconditions generated by SIEVE for both use cases. These weakest preconditions alongside their respective shadow op-

App	JahobSpec	Template	WP	Total
TPC-W	9.1 ± 0.1	3.8 ± 0.1	3.3 ± 0.1	16.2 ± 0.3
RUBiS	8.9 ± 0.0	3.3 ± 0.3	0.9 ± 0.1	13.2 ± 0.3

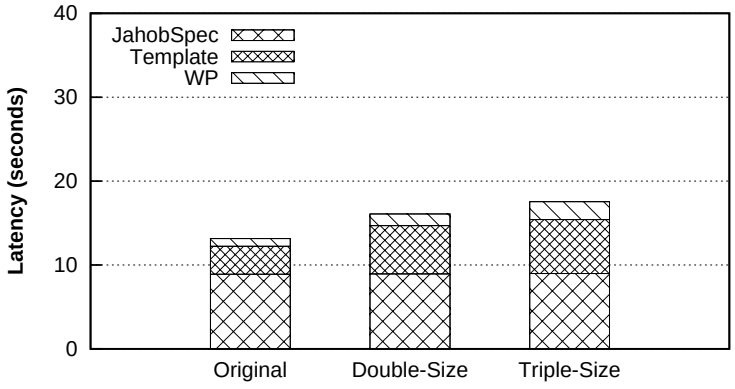
Table 4.7: Average and standard deviation of latency in seconds for static analysis tasks (5 runs).

eration template identifiers are used by the runtime logic to classify shadow operations as either blue or red. A weakest precondition denoted by `TRUE` implies that any shadow operation associated with that template is always invariant safe and therefore labeled blue. In contrast, a weakest precondition denoted by `FALSE` implies that shadow operations associated with that template must always be classified as red. The remaining non-trivial conditions must be evaluated at runtime by replacing their arguments with concrete values. For instance, when a `doBuyConfirm` transaction produces a negative delta, then the condition will be evaluated to `FALSE` and the corresponding shadow operation will be classified as red, otherwise the condition will be evaluated to `TRUE` and the shadow operation will be classified as blue.

Cost of static analysis. A relevant aspect of the static analysis component in SIEVE is the time required to execute it. To study this we have measured the time taken by the static analysis and present the obtained results in Table 4.7. We not only measured the end-to-end completion time, but also the time spent at each step, namely creating database data structures required by Jahob (JahobSpec), template creation (Template), and weakest precondition computation (WP). Overall, we can see that the execution time of the static component of SIEVE is acceptable, as less than 20 seconds are required to analyze both TPC-W and RUBiS. The code generation phase including both JahobSpec and Template dominates the overall static analysis. Compared to TPC-W, the time spent computing weakest preconditions is shorter in RUBiS, due to the smaller number of templates in Table 4.5.



(a) TPC-W



(b) RUBiS

Figure 4.4: Static analysis time vs. code base size.

Scalability. The code base size of TPC-W and RUBiS is somewhat small when compared to deployed applications. This raises a question concerning the scalability of the static analysis component of SIEVE with respect to the size of the code base. In order to analyze this aspect of SIEVE we have artificially doubled and tripled the size of each application code base and measured the time spent analyzing these larger code bases when compared with the original. The results are shown in Figure 4.4. The time spent generating the data structures required by Jahob is constant, since we did not change the database schema. However, the time spent computing the weakest preconditions for templates in TPC-W grows exponentially, and the time taken for the remaining steps

App	Workload	Manual	SIEVE
TPC-W	Browsing mix	0.49 (± 0.03)	0.48 (± 0.02)
	Shopping mix	0.79 (± 0.02)	0.81 (± 0.02)
	Ordering mix	6.31 (± 0.04)	6.30 (± 0.07)
RUBiS	Bidding mix	2.65 (± 0.09)	2.62 (± 0.07)

Table 4.8: Percentage of red shadow operations classified manually and by SIEVE (5 runs).

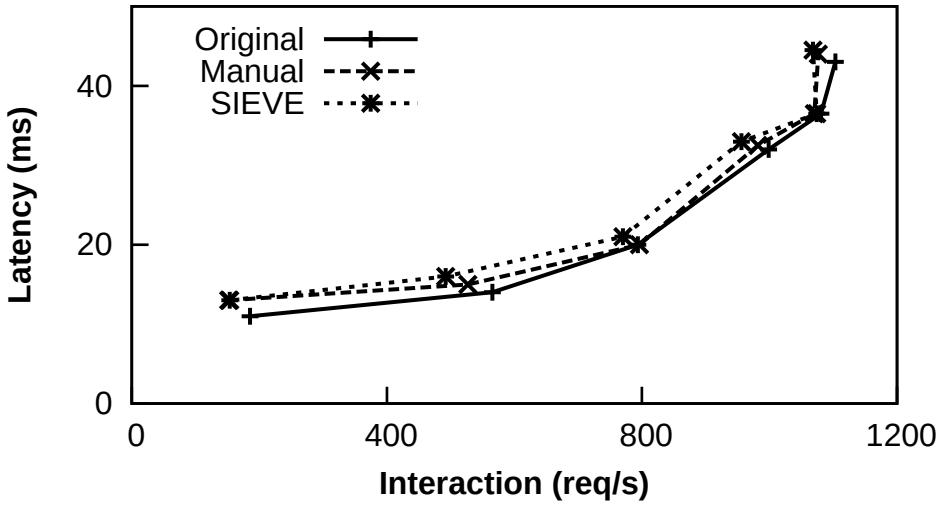
presents a sub-linear increase. These results lead us to conclude that the static analysis of SIEVE may scale to reasonable (though not very large code) sizes, especially taking into account that this process is executed a single time when adapting an application through the use of SIEVE.

Runtime logic

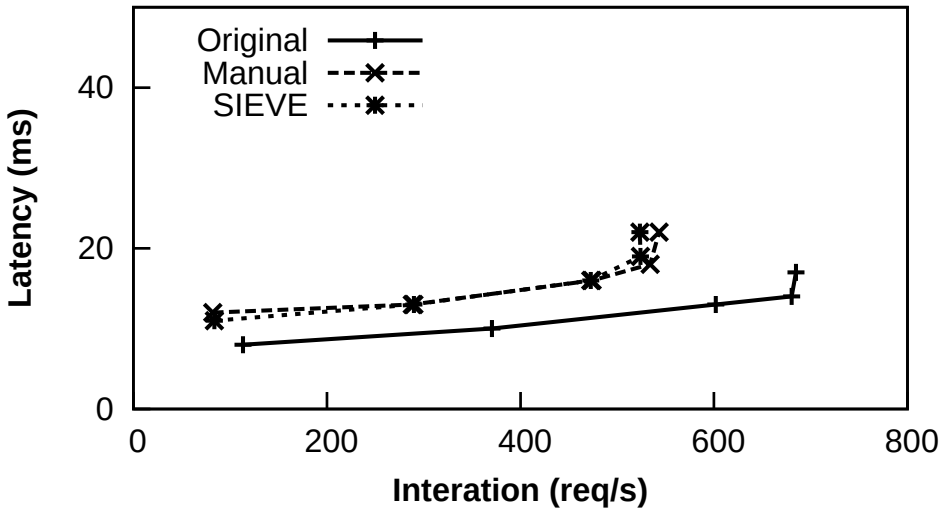
We evaluated the runtime performance of our example applications using SIEVE on top of Gemini.

Configurations. We populated the dataset for TPC-W using the following parameters: 50 EBS and 10,000 items. For RUBiS we populated the dataset with 33,000 items for sale, 1 million users, and 500,000 old items. We exercised all TPC-W workloads, namely browsing mix, shopping mix, and ordering mix, where the purchase activity varies from 5% to 50%. For RUBiS, we ran the bidding mix workload, in which 15% of all user activities generate updates to the application state.

Correctness validation. To verify that SIEVE labels operations correctly for both case studies, we compared the classification results obtained by running SIEVE with TPC-W and RUBiS against the results achieved manually in Chapter 3. Our finding in Table 4.8 shows that the percentage of shadow operations classified as red by SIEVE matches the results obtained through the manual classification. In addition, a careful inspection of the logs shows that the expected pairs of functions and parameters were



(a) TPC-W shopping mix



(b) RUBiS bidding mix

Figure 4.5: Throughput-latency graph without replication

in fact labeled as red. This implies that SIEVE is able to achieve the same labeling as a manual process while saving a significant amount of effort from programmers and avoiding human mistakes.

SIEVE runtime overhead. Next we compared the performance (throughput vs. latency) of the two applications across three single-site deployments: (1) SIEVE, (2)

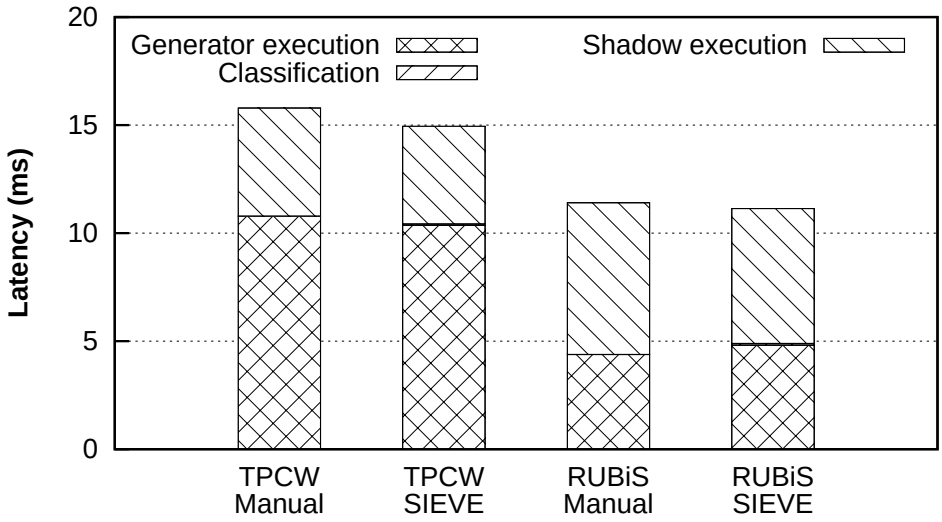


Figure 4.6: Breakdown of latency.

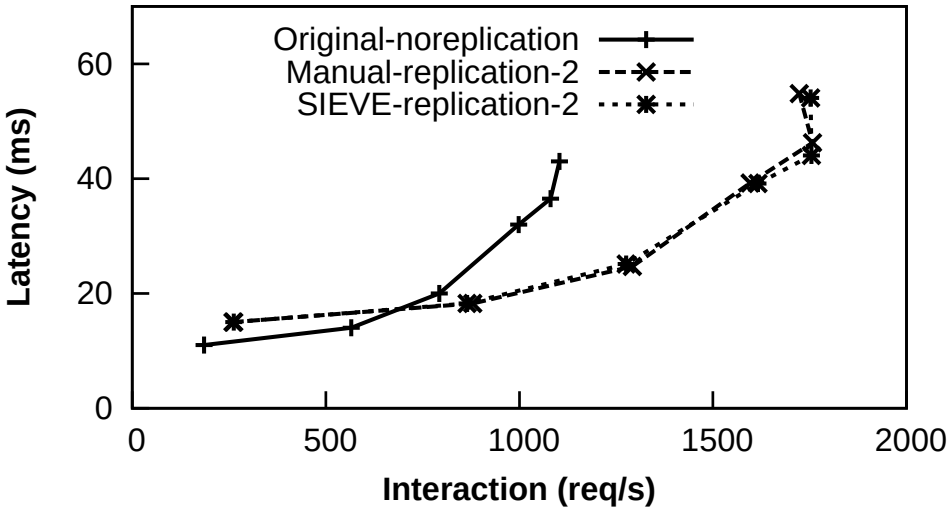
Original—the original unreplicated service without any overheads from creating and applying shadow operations, and (3) Manual—the RedBlue scheme with all labeling performed offline by the programmer. The expected sources of overhead for SIEVE are: (*i*) the dynamic creation of shadow operations; and (*ii*) the runtime classification of each shadow operation. The results in Figure 4.5 show that the performance achieved by SIEVE is similar to the one obtained with a manual classification scheme, and therefore the overheads of runtime classification are low. The comparison with the original scheme in a single site shows some runtime overhead due to creating and applying shadow operations (which is required for a replicated deployment so that all operations commute).

To better understand the sources of overhead imposed by SIEVE we measured the latency contribution of each runtime step executed by SIEVE and compared it with the latency contribution of these steps when relying on a manual adaptation. In particular, we focused on the following tasks: generator execution (producing a shadow operation), classification (determining shadow operation colors), and shadow execution (applying shadow operations).

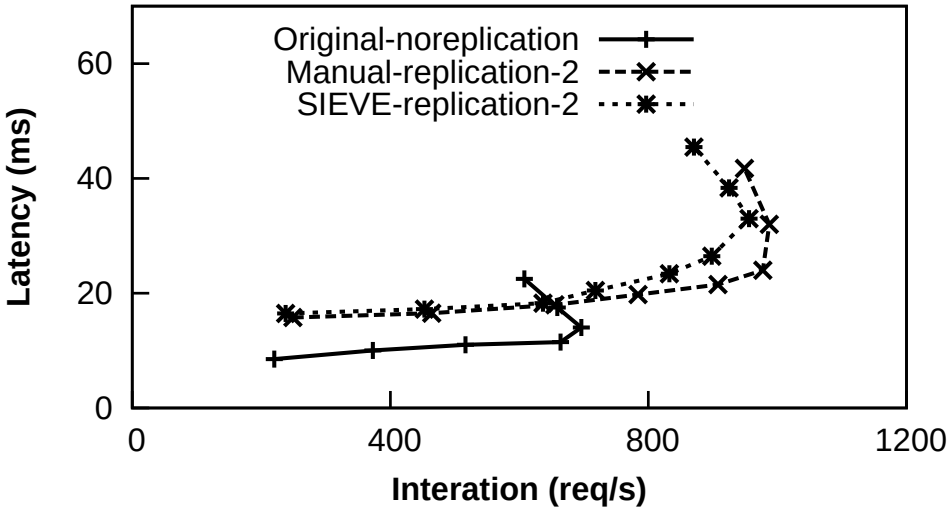
Figure 4.6 shows the average contribution to request latency of each of these steps (Only update requests are considered since read-only queries do not generate side effects.) For the manual adaptation, there is no latency associated with classifying shadow operations, since the classification of all shadow operations is pre-defined. In contrast, SIEVE performs a runtime classification, but the results show that the time consumed in this task is negligible. In particular, SIEVE takes 0.064 ± 0.002 ms and 0.072 ± 0.001 ms for looking up the dictionary and evaluating the condition for TPC-W and RUBiS, respectively. Regarding the generator execution and shadow execution, both the manual adaptation and SIEVE present the same latency overheads.

Replication benefits. The results previously discussed in this section have shown that the use of SIEVE imposes a small overhead when compared to a standalone execution of the unmodified use cases, mostly due to the runtime shadow operation generation and classification. However, SIEVE was designed to allow replication to bring performance gains through the use of weak consistency in replicated deployments. To evaluate these benefits, we conducted an experiment where we deployed the two applications (1) without replication, (2) using manual classification in Gemini, and (3) using SIEVE, with two replicas in the same site for the last two options. (The use of single site replication instead of geo-replication makes our results conservative, since the overheads of runtime classification become diluted when factoring in cross-site latency.)

The results in Figure 4.7 show that weakly consistent replication for a large fraction of the operations brings performance gains. In particular, one observes that the peak throughput with 2 replicated Gemini instances running TPC-W is improved by 59.0%, and the peak throughput for RUBiS in this setting is improved by 37.4%. The additional latency introduced in this case is originated by the necessity of coordination among replicas to totally order red shadow operations. The results also confirm that the overhead of runtime classification when compared to the manual, offline classification are low. Note



(a) TPC-W shopping mix



(b) RUBiS bidding mix

Figure 4.7: Throughput-latency graph of systems with no replication or with two replicas.

that there is a point where the throughput goes down while there is still an increase in latency in Figure 4.7(b). This happens because the database becomes saturated at this point.

4.7 Limitations and future work

Although SIEVE reduces human intervention that might be involved in making the choice of CRDTs and consistency levels for scaling out web services, there are still several points for optimization, which we leave as future work to address.

First, while the CRDT library covers a set of most representative CRDT types that suffice for all use cases exhibited in our case study applications, it does not include some more recent proposals like maps [ria], and does not have a full coverage of SQL features defined in [SQL99]. This incompleteness may limit the selection of merging semantics, which the programmer may intend to use not only for ensuring state convergence, but also for providing meaningful merged outcomes.

Second, we observed performance degradation when running unreplicated applications with SIEVE. This is because we implemented the CRDT transformation in a JDBC driver and it requires us to parse every SQL statement to figure out the side effects. One possibility is augmenting the database code with this logic so that we can take advantage of rich information from query execution plans generated by the database.

Third, our approach is based on a fundamental assumption that iterations in loops are independent w.r.t each other, so that weakest preconditions can be efficiently computed. This assumption is also a limitation of our approach, as SIEVE will conservatively generate a `FALSE` condition for operations if their precondition computation fails, in case such a loop independence property does not hold. Additionally, we would like to explore algorithms to automatically verify loop independence, instead of relying on manual processing.

4.8 Summary

In this chapter, we presented SIEVE, which is, to the best of our knowledge, the first tool to automate the choice of consistency levels in a replicated system. Our system re-

lieves the programmer from having to reason about the behaviors that weak consistency introduces. SIEVE minimizes human intervention by only requiring the programmer to write the system invariants that must be preserved and to provide annotations regarding merge semantics. Our evaluation shows that SIEVE labels operations accurately, incurring a modest runtime overhead when compared to labeling operations manually and offline.

5 Minimizing coordination in replicated systems

In this chapter, we present a novel consistency definition, Partial Order-Restrictions consistency (or short, PoR consistency), generalizing the tradeoff between performance and the amount of coordination paid to restrict the ordering of certain operations behind RedBlue consistency. We also describe the design, implementation, and evaluation of Olisipo, which is an efficient coordination service for offering PoR consistent replication.

This chapter is organized as follows. We describe the motivation and contributions of this chapter in Section 5.1. Then we discuss the most relevant work in Section 5.2. We introduce the definition of PoR consistency in Section 5.3, and a set of principles to infer restrictions in Section 5.4. We describe an efficient coordination service called Olisipo in Section 5.5. In section 5.6, we analyze the experimental results from replicating an extended version of RUBiS under PoR consistency through Olisipo. Finally, some limitations of our work are discussed in Section 5.7 and we conclude the chapter in Section 5.8.

5.1 Motivation and contributions

As presented in Chapter 3, our first attempt to relieve the tension between consistency and performance in geo-distributed scenarios is to introduce RedBlue consistency, in which some operations can be executed under strong consistency (and therefore incur

```
1  boolean placeBid(int itemId, int
       clientId, int bid){
2   boolean result = false;
3   beginTxn();
4   if(open(itemId)){
5     exec(INSERT INTO bidTable VALUES
          (bid, clientId, itemId));
6     result = true;
7   }
8   commitTxn();
9   return result;
10 }
```

(a) Original placeBid operation.

```
1  placeBid'(int itemId, int clientId,
       int bid){
2    exec(INSERT INTO bidTable VALUES
          (bid, clientId, itemId));
3  }
```

(b) Shadow placeBid' operation.

```
1  int closeAuction(int itemId){
2   int winner = -1;
3   beginTxn();
4   close(itemId);
5   winner = exec(SELECT userId FROM
        bidTable WHERE iId = itemId ORDER
         BY bid DESC limit 1);
6   exec(INSERT INTO winnerTable VALUES (
        itemId, winner));
7   commitTxn();
8   return winner;
9  }
```

(c) Original closeAuction operation.

```
1  closeAuction'(int itemId, int winner){
2   close(itemId);
3   exec(INSERT INTO winnerTable VALUES (
        itemId, winner));
4  }
```

(d) Shadow closeAuction' operation.

Figure 5.1: Pseudocode for the original and shadow operations of the `placeBid` and `closeAuction` transactions in an extended version of RUBiS.

in a high performance penalty) while other operations can be executed under weaker consistency (namely causal consistency [LFKA11]). The core of this solution is a labeling methodology for guiding the programmer to classify shadow operations (side effects of original application operations) into the strong and weak consistency categories. The labeling process works as follows: shadow operations that either do not commute w.r.t all others or potentially violate invariants must be strongly consistent, while the remaining can be weakly consistent. To make the adoption of RedBlue consistency easy, in addition, we built SIEVE (seen in Chapter 4) to automate this binary decision by requiring a small amount of programmer input.

This binary classification methodology works well for many web applications, but it can also lead to unnecessary coordination in some cases. We illustrate this with an

extended version of RUBiS[1], as shown in Figure 5.1, where an operation `placeBid` (Figure 5.1(a)) creates a new bid for an item if the corresponding auction is still open, and an operation `closeAuction` (Figure 5.1(c)) closes an auction for an item, declaring a single winner. In this example, the application-specific invariant is that the winner must be associated with the highest bid across all accepted bids. The other two subfigures (Figure 5.1(b) and Figure 5.1(d)) depict the shadow operations of the two prior operations, respectively, guaranteeing that these shadow operations apply changes in a commutative fashion regardless of execution order. We omit in the Figure the commutative shadow operation generation, since it has been covered in Sections 3.6 and 4.4.

When applying RedBlue consistency to replicate such an auction service, we note that the concurrent execution under weak consistency of a `placeBid` operation with a bid that is higher than all accepted bids and a `closeAuction` operation can lead to the violation of the application invariant. This happens because the generator of `closeAuction` will ignore the highest bid created by the concurrent shadow `placeBid'`. Unfortunately, the only way to address this issue in RedBlue consistency is to label both shadow operations as strongly consistent (red), i.e., all shadow operations of either type will be totally ordered w.r.t each other, which will incur in a high coordination overhead, while not taking advantage of the flexibility provided by RedBlue consistency. Intuitively, however, there is no need to order pairs of `placeBid'`, since a bid coming before or after another does not affect the winner selection. This highlights that our previous coarse-grained operation classification into two levels of consistency can be conservative, and some services could benefit from additional flexibility in terms of the level of coordination.

[1]The original RUBiS is not complete since it does not include a closeAuction operation that declares the winners for auctions. As a result, in this chapter, we extended the original RUBiS by adding it a closing auction functionality.

In this chapter, to address the above issue, we present a principled methodology for allowing developers to tune tradeoffs between performance and consistency requirements. In summary, we make the following three main contributions:

1. We generalize the principles behind the binary classification by breaking down the coarse-grained constraint that totally orders all strongly consistent operations into a set of fine-grained restrictions, each of which only imposes an order between a pair of operations. Following this path, we propose a novel generic consistency definition, *Partial Order-Restrictions consistency (or short, PoR consistency)*, which takes a set of restrictions as input and forces these restrictions to be met in all partial orders. This creates the opportunity for defining many consistency guarantees within a single replication framework by expressing consistency levels in terms of visibility restrictions on pairs of operations. Weakening or strengthening the consistency semantics in the context of PoR consistency is achieved by imposing fewer or more restrictions on pairs of operations.

2. We design an analysis to identify, for every application, a set of restrictions over pairs of its operations so that state convergence and invariant preservation are ensured if these restrictions are enforced throughout all executions of the system. The fundamental challenge of doing this is that missing required restrictions will lead applications to diverge state or violate invariants, while placing unnecessary restrictions will lead to a performance penalty due to the additional coordination. To overcome this, this analysis aims to find a minimal set of restrictions. (By minimal we mean that removing a single restriction no longer ensures the desired properties.)

3. We further observe that, given a set of restrictions across the visibility of operations, a key aspect to ensure good performance in a replicated service is to enforce these restrictions in an efficient way. In fact, there exist several coordina-

tion techniques/protocols that can be used for enforcing a given restriction, such as Paxos, distributed locking, or global barriers. However, depending on the frequency over time in which the system receives operations confined by a restriction, different coordination approaches lead to different performance tradeoffs. Therefore, to minimize the runtime coordination overhead, we also propose an efficient coordination service that helps replicated services use the most efficient protocol by taking into account the deployment characteristics measured at runtime.

We extended RUBiS to incorporate a closing auction functionality, determined how to best run it under PoR consistency, replicated this web application with Olisipo, and compared against the results we obtained from the RedBlue consistent version. The experiment results show that PoR consistency requires fewer restrictions than RedBlue consistency, and the usage of PoR consistency and Olisipo offers a significantly better performance than the combination of RedBlue consistency and Gemini.

5.2 Related work

We summarize most relevant related work and compare it against our PoR consistency framework in the following categories.

Consistency models. In the past decades, many consistency proposals have been focusing on the reduction in coordination among concurrent operations to improve scalability in replicated systems [LLSG92, SPAL11, LPC+12, LLaC+14, ACHM14, ACHM11, ZSS+15]. However, they only allow the programmer to choose from a limited number of consistency levels that they support, such as strong, causal or eventual consistency. Unlike these approaches, PoR consistency offers a fine-grained tunable tradeoff between performance and consistency using the visibility restrictions between pairs of operations to express consistency semantics. In addition, most previous proposals [ACHM14, LLSG92, SPAL11, ACHM11] only take into account operation commuta-

tivity to determine the need for coordination, instead of invariant preservation, which is analyzed in our solution.

In the family of consistency proposals concerning application-specific invariants, Bailis et al. [BFF+14] proposed I-confluence to avoid coordination by determining if a set of transactions are I-confluent w.r.t database integrities, i.e., integrity constraints might be violated if they were executing without coordination. Indigo [BDF+15] defines consistency as a set of invariants that must hold at any time, and presents a set of mechanisms to enforce these invariants efficiently on the top of eventual consistency. Similar to Indigo, warranties [LMA+14] map consistency requirements to a set of assertions that must hold in a given period of time, but it needs to periodically invalidate assertions when updates arrive. The work from Roy et al. additionally proposes a program analysis against transaction code for producing warranties [RKB+15]. In contrast to these approaches, PoR consistency takes an alternative approach by modeling consistency as restrictions over operations.

There also exist a few proposals which map consistency semantics to the ordering constraints defined over pairs of operations. For example, Generic Broadcast defines conflict relations between messages for fast message delivery, which are analogous to visibility restrictions used in our solution [PS99]. Most recently, a concurrent work proposed by Gotsman et al. encoded the `conflict relation` concept into a proof system, which enables to analyze if consistency choices expressed into conflict relations meet the target properties [GYF+15]. Our approach differs from all these consistency proposals in the following aspects. First, we have a different formalism that captures new situations that lead to invariant violations, such as the possibility of three different types of requests being necessary to trigger such cases. Second, we provide programmers with the ability to infer a minimal set of (fine-grained) restrictions to achieve state convergence and invariant preservation. Third, we explore the possibility of using different coordination protocols to enforcing restrictions efficiently.

Paxos and its variants. State machine replication [Sch90] is a standard technique to make a set of servers behave like a single machine. Paxos [Lam98], one of the classic algorithms that implement state machines, forces every replica to process a set of requests in the same sequential order. In order to reduce the number of message exchanges for achieving distributed consensus, several variants of Paxos have been proposed. Fast Paxos [Lam06] aims at improving latency by allowing every replica to propose values but suffers from high latency when concurrent proposals occur. To avoid the penalty introduced by collisions, some other variants of Paxos explore operation semantics to take into account a weaker guarantee that not all operations are needed to be totally ordered [Lam05, KPF+13, MAK13]. Generalized Paxos (GPaxos) allows replicas to execute a set of operations in different orders as long as operations commute w.r.t each other; however, it still has to resort to the classic Paxos algorithm [Lam98] when the leader notices two concurrent non-commuting requests [Lam05]. Kraska et al. design an optimistic commit protocol called MDCC, which embodies GPaxos and explores operation commutativity for making geo-replicated transactions fast. Egalitarian Paxos (EPaxos) takes as input a set of pre-defined constraints, each of which defines a dependency between a pair of operations, and enables each replica to order two concurrent conflicting requests according to their apriori dependency relation [MAK13].

A major difference between our work and these Paxos variants is that we develop an analysis to extract pairs of conflicting operations by considering the impact of concurrent executions on achieving state convergence and invariant preservation. Furthermore, all these protocols only reduce the number of communication steps, but still require to talk to a large quorum of replicas. In contrast, in our work, operations that are not confined by conflicting relations can be first accepted in a single replica and later asynchronously replicated to other replicas.

Efficient transaction processing. Some other work focuses on how to reduce coordination in transaction processing. For example, transaction chopping [SLSV95] and Lynx [ZPZ+13] suggest that breaking large transactions into smaller pieces can improve performance, and they design analysis algorithms for chopping transactions without sacrificing serializability. While this work has been done merely by checking conflicts in read/write sets between pairs of transaction pieces, we design a comprehensive and fine-grained analysis concerning commutativity and invariant preservation for avoiding coordination when possible. These techniques are also orthogonal to our proposal so that we can apply them to prune out the non-critical code sections prior to running our analysis.

5.3 Partial Order-Restriction Consistency

In this section we introduce Partial Order-Restrictions consistency (or short, PoR consistency), a novel consistency model that allows the developer to reason about various consistency requirements in a single system. The key intuition behind our proposal is that this model is generic and can be perceived as a set of restrictions imposed over admissible partial orders across the operations of a replicated system.

5.3.1 Defining PoR consistency

We formulate PoR consistency by following the same methodology we used for RedBlue consistency (Chapter 3). The definition of PoR consistency includes three important components: (1) a set of restrictions, which specifies the visibility relations between pair-wise operations; (2) a restricted partial order (or short, R-order), which establishes a (global) partial order of operations respecting operation visibility relations; and (3) a set of site-specific causal serializations, which corresponds to total orders in which the operations are locally applied. We define these components formally as follows:

Definition 10 (Restriction) *Given a set of operations U, a* restriction *is a symmetric binary relation on $U \times U$.*

For any two operations u and v in U, if there exists a restriction relation $r(u, v)$, then they must be ordered in any partial order $\prec$, i.e., $u \prec v \vee v \prec u$. We capture this point in the following definition.

Definition 11 (Restricted partial order) *Given a set of operations U, and a set of restrictions R over U, a* restricted partial order *(or short,* R-order*) is a partial order $O = (U, \prec)$ with the following constraint: $\forall u, v \in U,\ r(u, v) \in R \implies u \prec v \vee v \prec u$.*

We also say that the restrictions in R are met in the corresponding R-order if this order satisfies the above definition. The restriction definition is analogous to the conflict relation in generic broadcast [PS99]. Therefore, the coordination plan required by replicating the previously described banking service with generic broadcast is the same as the one associated with PoR consistency, i.e., any pair of `withdraw(x)` operations must be ordered with each other, since a `withdraw(x)` operation can only modify the relevant account balance if the current balance is not below x. However, the PoR consistency framework improves on generic broadcast by offering a precise method for identifying a set of restrictions (or, conflict relations), which comprise a minimal amount of coordination.

Under the context of PoR consistency, every site (replica) executes operations following a linear extension of the global R-order. The following definition defines what linear extensions are allowed with respect to a given R-order.

Definition 12 (Legal serialization) *$O' = (U, <)$ is a* legal serialization *of R-order $O = (U, \prec)$ if O' is a linear extension of O; i.e., $<$ is a total order compatible with the partial order defined by $\prec$.*

As introduced in Chapter 3, our proposal of splitting original application operations into pairs of generator and shadow operations changes the traditional state machine

replication definitions. In this work, we also embrace the shadow operation concept so that we can reduce the number of required restrictions for ensuring state convergence. With this change, when user requests are accepted by any site, that site executes their generator operations and creates corresponding shadow operations. In addition, every site also incorporates remote shadow operations that are shipped from all other sites into its local serialization. We denote U a set of shadow operations. For a site i, its generator operation set is denoted by V_i. The following definition captures the application of both local and remote shadow operations at a site i.

Definition 13 (Causal legal serialization) *Given a site i, an R-order $O = (U, \prec)$ and the set of generator operations V_i received at site i, we say that $O_i = (U \cup V_i, <_i)$ is an* i-causal legal serialization *(or short, a* causal serialization*) of O if*

- *O_i is a total order;*
- *$(U, <_i)$ is a legal serialization of O;*
- *For any $h_v(S) \in U$ generated by $g_v \in V_i$, S is the state obtained after applying the sequence of shadow operations preceding g_v in O_i;*
- *For any $g_v \in V_i$ and $h_u(S) \in U$, $h_u(S) <_i g_v$ in O_i iff $h_u(S) \prec h_v(S')$ in O.*

A replicated system with k sites is then PoR consistent if every site applies a causal serialization of the same global R-order O.

Definition 14 (Partial Order-Restrictions consistency) *A replicated system $\mathscr{S}$ with a set of restrictions R is* Partial Order-Restrictions consistent *(or short,* PoR consistent*) if each site i applies shadow operations according to an i-causal serialization of R-order O.*

5.3.2 Expressiveness

The intuition behind the PoR consistency model is that the model can be viewed as a parametrized function, which takes restrictions as input, and outputs a particular consistency model where the restrictions must be met in any partial order. To demonstrate the power of PoR consistency, we use it to express many different consistency requirements. For causal consistency [LFKA11] (excluding any restrictions to provide session guarantees), the restriction set is empty, since causality is already preserved in the definition of PoR consistency by having $u \prec v \wedge v \prec w \implies u < w$. Regarding RedBlue consistency, to capture the notion of strongly consistent (red) operations, we define the following restriction set: for any pair of operations u, v, if u and v are strongly consistent, we have $r(u, v)$. Serializability [BHG87] totally orders all operations, so its restriction set is as follows: for any pair of operations u, v, we have $r(u, v)$.

5.4 Restriction inference

When replicating a service under PoR consistency, the first step is to infer restrictions to ensure two important system properties, namely state convergence and invariant preservation. The major challenge we face is to identify a small set of restrictions for making the replicated service eventually converge and never violate invariants so that the amount of required coordination is minimal. With regard to state convergence, we take a similar methodology adopted in prior research [SPBZ11a, LPC+12, LLaC+14], which is to check operation commutativity. However, unlike RedBlue consistency, under which all operations that are not globally commutative must be totally ordered, PoR consistency only requires that an operation must be ordered w.r.t another one if they do not commute.

To always preserve application-specific invariants, instead of totally ordering all non-invariant safe shadow operations, i.e., those that potentially transition from a valid

state to an invalid one, we try to isolate the operations that exclusively contribute to an invariant violation from the rest. To do so, we introduce a new concept, called an `I-conflict set`, which defines a minimal set of shadow operations that lead to an invariant violation when they are running concurrently in a coordination-free manner. By minimal, we mean that by removing any shadow operation from such a set, the violation will no longer persist. To identify a minimal set of restrictions, we first perform an analysis over any subsets of the shadow operation set to discover all `I-conflict` sets. Then, for any such set, adding a restriction between any pair of its operations is sufficient to eliminate the problematic executions.

Next, we present the definitions, theorems, proofs and algorithms regarding the restriction set identification and refinement.

5.4.1 State convergence

The state convergence definition under the context of PoR consistency looks similar to Definition 5 of RedBlue consistency in Section 3.3. A PoR consistent replicated system is state convergent if all its replicas reach the same final state when the system becomes quiescent, i.e., for any pair of causal legal serializations of any R-order, L_1 and L_2, we have $S_0(L_1) = S_0(L_2)$, where S_0 is a valid initial state. We state the necessary and sufficient conditions to achieve this in the following theorem.

Theorem 3 *A PoR consistent system $\mathscr{S}$ with a set of restrictions R is **convergent**, if and only if, for any pair of its shadow operations u and v, $r(u, v) \in R$ if u and v don't commute.*

In order to prove this theorem, we need the assistance from three lemmas introduced in Chapter 3, namely, Lemmas 1, 2 and 3. The first two lemmas remain valid under PoR consistency with a minor change that RedBlue order is replaced with R-order, since their proofs remain unchanged. Lemmas 1 asserts that, given a legal serialization, swapping two adjacent shadow operations in this serialization that are not ordered by the

underlying R-order results in another legal serialization. Lemmas 2 asserts that given an R-order and one of its legal serializations, if there exists a pair of shadow operations u and v that is not ordered by the R-order, then there exists an adjacent pair of shadow operations between u and v in that serialization that are not ordered by the R-order. In contrast to the first two lemmas, we have to change the third lemma since RedBlue consistency achieves state convergence by requiring all blue shadow operations to be globally commutative, but PoR consistency only needs any pair of unordered shadow operations to commute. As such, we change Lemma 3 into a new lemma (Lemma 4), which asserts that two legal serializations of an R-order that differ in the order of exactly one pair of adjacent shadow operations are state convergent, if the two operations commute.

Lemma 4 *Assume $O_i = (U, <_i)$ and $O_j = (U, <_j)$ are both legal serializations of R-Order $O = (U, \prec)$ that are identical except for two adjacent operations u and v such that $u <_i v$ and $v <_j u$ and that u and v commute. Then $S_0(O_i) = S_0(O_j)$.*

Proof: Let P and Q be the greatest common prefix and suffix of O_i and O_j, respectively. Further, let $S_P = S_0(P)$, $S_{uv} = S_P + u + v$, and $S_{vu} = S_P + v + u$. By the definition of operation commutativity, $S_{uv} = S_{vu}$. It then follows from the definition of a deterministic state machine that $S_{uv}(Q) = S_{vu}(Q)$. By a similar argument, the final state reached by sequentially executing operations in O_i against S_0 according to $<_i$ is equal to the final state obtained by sequentially applying operations in Q against S_{uv} according to $<_i$, namely $S_0(O_i) = S_{uv}(Q)$. By a similar argument, we know $S_0(O_j) = S_{vu}(Q)$. Finally, we have $S_0(O_i) = S_0(O_j)$. ∎

After adapting these lemmas from Chapter 3 to PoR consistency, we use them to construct the proof of the state convergence theorem (Theorem 3) as follows:

Proof: ($\Leftarrow$:) We first show that if for any pair of non-commuting operations of $\mathscr{S}$, a restriction between this pair of operations is in R, then the PoR consistent system $\mathscr{S}$ is

convergent. To prove this, it is sufficient to show that any pair of legal serializations of their underlying R-order O, O_i and O_j, is state convergent, i.e., $S_0(O_i) = S_0(O_j)$. There are two cases to consider:

Case 1: $O_i = O_j$. The underlying deterministic state machine ensures that $S_0(O_i) = S_0(O_j)$.

Case 2: $O_i \neq O_j$, in which case $\exists u, v \in U$ such that $u <_i v$ and $v <_j u$. Since both O_i and O_j are legal serializations of O, it follows that $u \not\prec v$ and $v \not\prec u$. It then follows from Lemma 2 that we can find an adjacent pair of operations r, s in both O_i and O_j such that $r <_i s \wedge s <_j r \wedge r \not\prec s \wedge s \not\prec r$. We construct a new serialization O_{i+1} by duplicating O_i but swapping the order of r and s in O_{i+1}, i.e., Q_i and Q_{i+1} are identical, except that $r <_i s \wedge s <_{i+1} r$. By Lemma 1, O_{i+1} is also a legal serialization of O. It then follows from the hypothesis that r and s commute and from Lemma 4 that O_i and O_{i+1} are convergent.

If $O_{i+1} \neq O_j$, we continue the construction by finding an adjacent pair of operations whose order is different in O_{i+1}, O_j. By swapping the two operations, we obtain another legal serialization O_{i+2}. We can then continue to swap all such adjacent pairs until the last constructed serialization is equal to O_j. This is achievable since at every step the number of operation pairs in the corresponding newly constructed legal serialization whose orders are different in O_j decreases. At the end, the construction process results in a chain of legal serializations where the first one is O_i and the last is O_j, and any consecutive pair of legal serializations is identical except for the order of an adjacent pair of elements. It then follows Lemma 4 that every consecutive pair of serializations in the chain is state convergent, thus $S_0(O_i) = S_0(O_j)$.

($\Rightarrow$:) (Proof by Contradiction.) We show that if a PoR consistent system $\mathscr{S}$ with a restriction set R is convergent, then for any pair of non-commuting shadow operations, there must exist a restriction confining the two operations in R. Since $\mathscr{S}$ is convergent, we know that for any R-order of $\mathscr{S}$, any pair of causal legal serializations of that R-order

are convergent. We assume by contradiction that there exist two shadow operations u, v such that they don't commute and $r(u, v) \notin R$. By the definition of commutativity, there exists a state S such that $S + u + v \neq S + v + u$. We can find a state S_0 and a sequence of shadow operations of $\mathscr{S}$, $O'(P, <)$, such that $S_0(O') = S$. Then, we can construct a R-order $O(U, \prec)$, where

- $U = P \cup \{u, v\}$;
- for any pair of operations in P, m and n, $m < n \iff m \prec n$;
- for any operation m in P, $m \prec u$ and $m \prec v$.

It follows from the above construction that u, v are the maximal elements of O. It follows from the definition of causal legal serialization (Definition 13) that we can construct two causal legal serializations L_1 and L_2 of O such that $L_1 = O' + u + v$ and $L_2 = O' + v + u$. As $S_0(L_1) = S_0(O' + u + v)$, $S_0(L_1) = S_0(O') + u + v$. It follows from $S_0(O') = S$ that $S_0(L_1) = S + u + v$. By a similar argument, $S_0(L_2) = S + v + u$. It then follows from $S + u + v \neq S + v + u$ that $S_0(L_1) \neq S_0(L_2)$. As L_1 and L_2 are not convergent, $\mathscr{S}$ is not convergent. Contradiction is found. ∎

5.4.2 Invariant preservation

As presented when we proposed RedBlue consistency (Chapter 3), the methodology for identifying restrictions imposed on RedBlue orders for maintaining invariants is to check if a shadow operation is invariant safe or not. If not, to avoid invariant violations, the generation and replication of all non-invariant safe shadow operations must be coordinated. However, we observed that for some non-invariant safe shadow operations u, the corresponding violation only happens when a particular subset of non-invariant safe shadow operations (including u) are not partially ordered. To eliminate all invariant violating executions with a minimal amount of coordination, therefore, we need to precisely define, for each violation, the minimal set of non-invariant safe shadow operations

that are involved. We call this set an `invariant-conflict operation set`, or short, `I-conflict set`. We formally define this as follows.

Definition 15 (Invariant-conflict operation set) *A set of shadow operations G is an* invariant-conflict operation set *(or short,* I-conflict set*), if the following conditions are met:*

- $\forall u \in G$, *u is non-invariant safe;*
- $|G| > 1$*;*
- $\forall u \in G$, $\forall$ *sequence P consisting of all shadow operations in G except u, i.e., $P = (G \setminus \{u\}, <)$, and $\exists$ a reachable and valid state S, s.t. $S(P)$ is valid, and $S(P + u)$ is invalid.*

In the above definition, the last point asserts that G is minimal, i.e., removing one shadow operation from it will no longer lead to invariant violations. We will use the following example to illustrate the importance of minimality. Imagine that we have an auction on an item i being replicated across three sites such as US, UK and DE, and having initially a 5 dollar bid from *Charlie*. Suppose also that three shadow operations, namely, $placeBid'(i, Bob, 10)$, $placeBid'(i, Alice, 15)$, and $closeAuction'(i)$ are accepted concurrently at the three locations, respectively. After applying all of them against the same initial state at every site, we end up with an invalid state, where *Charlie* rather than *Bob* and *Alice* won the auction. This invariant violating execution involves three concurrent shadow operations, but one of the two bid placing shadow operations is not necessarily to be included in G, as even after excluding the request from either *Bob* or *Alice*, the violation still remains. According to Definition 15, $\{placeBid', closeAuction'\}$ is an `I-conflict set`, while $\{placeBid', placeBid', closeAuction'\}$ is not. Intuitively, avoiding invariant violations is to prevent all operations from the corresponding `I-conflict`

set from running in a coordination-free manner. The minimality property enforced in the `I-conflict set` definition allows us to avoid adding unnecessary restrictions.

Based on the above definition, we then can formulate the invariant preservation property into the following theorem.

Theorem 4 *Given a PoR consistent system $\mathscr{S}$ with a set of restrictions $R_{\mathscr{S}}$, for any execution of $\mathscr{S}$ that starts from a valid state, no site is ever in an invalid state, if the following conditions are met:*

- *for any its* `I-conflict` *set G, there exists a restriction $r(u, v)$ in $R_{\mathscr{S}}$, for at least one pair of shadow operations $u, v \in G$; and*
- *for any pair of shadow operations u and v, $r(u, v)$ in $R_{\mathscr{S}}$ if u and v don't commute.*

We prove the invariant preservation theorem by contradiction as follows:

Proof: We assume by contradiction that invariant violations are possible with a sufficient set of restrictions $R_{\mathscr{S}}$ in place. Let E be an invariant violating execution of $\mathscr{S}$ and $O(U, \prec)$ be a smallest R-order of E that triggers the violation. Let $O_i(U, <)$ be a causal legal serialization of R-order O at site i. As O_i violates the corresponding invariant, $S_0(O_i)$ is invalid. If U is empty, then $S_0(O_i) = S_0$, and O_i is in a valid state. This violates the assumption that O_i is in an invalid state. The theorem is proved.

Then we consider that U is non-empty. Let G be the set of shadow operations that are maximal according to $O(U, \prec)$, i.e., $G \subset U$, and $u \in G \Leftrightarrow \nexists v \in U$ s.t. $u \prec v$. The fact that U is not empty implies that G is not empty as well.

As follows, we will prove that G is an `I-conflict set`.

We first consider the case that G contains invariant-safe shadow operations. Let v be such an invariant safe shadow operation in G. If v is not the last operation in O_i, then it follows from Lemma 1 and 2 that we can swap v and any shadow operation u, s.t. $u \in G$, $u \neq v$ and $v < u$. This swapping process terminates when it produces a new legal

serialization O_j of the R-order O, where v appears as the last operation, i.e., $O_j = O'_j + v$. It also follows from the assertion that any pair of shadow operations that are not ordered in $\prec$ commute w.r.t each other and Lemma 4 that $S_0(O_j) = S_0(O_i)$. As $S_0(O_i)$ is invalid and $S_0(O'_j) + v = S_0(O_j)$, $S_0(O'_j) + v$ is invalid. It then follows from the fact that v is invariant safe and the invariant safe shadow operation definition (Definition 8) that the state before applying v must be invalid, i.e., $S_0(O'_j)$ is not valid. This implies that there exists a smaller R-order $O'(U \setminus \{v\}, \prec)$ than $O(U, \prec)$ that triggers the corresponding invariant. It contradicts with our assumption that O is a smallest R-order observing invalid state. Therefore, we only need to analyze the case when all shadow operations in G are non-invariant safe. The first condition of the `I-conflict set` definition is met.

We continue by checking if $|G| = 1$, i.e., G contains only a single non-invariant safe shadow operation. Let v be that operation. $O_i = O'_i + v$. As O_i is in an invalid state, $S_0(O'_i + v)$ is invalid. It follows from the assumption that O_i is the causal legal serialization of O at site i (where the generator of v was executed) and the correct shadow operation definition (Definition 6) that the state $S_0(O'_i)$, which v was created from, is also invalid. By similar logic as above, there exists a smaller invariant violating R-order than O, and contradiction is found. As a result, $|G| > 1$. The second condition of the `I-conflict set` definition is met.

Finally, we check if G also meets the third condition of the `I-conflict set` definition. Let $O'_i(U \setminus G, <)$ be a prefix of O_i excluding all operations of G from O_i. Let $S = S_0(O'_i)$. $\forall u \in G$, we can construct a legal serialization O_j of O such that $O_j = O'_i + T + u$, where T is a sequence consisting of all shadow operations in $G \setminus \{u\}$. It also follows from Lemma 1, 2, Lemma 4 and the assertion that any pair of unordered shadow operations commute that $S_0(O_i) = S_0(O_j)$. Since the underlying R-order O is a smallest R-Order violating the corresponding invariant, S and $S(T)$ are valid, and $S(T + u)$ is invalid.

As G meets all three conditions presented in the `I-conflict set` definition (Definition 15), G is an `I-conflict` set. It then follows from the assertion in the invariant

Algorithm 1 State convergence restrictions discovery

```
1: function SCRDISCOVER(T)  ▷ T: the set of shadow operations of the target system
2:     R ← {}                                                  ▷ R: the restriction set
3:     for i ← 0 to |T| − 1 do
4:         for j ← i to |T| − 1 do
5:             if T_i do not commute with T_j then
6:                 R ← R ∪ {r(T_i, T_j)}
7:             end if
8:         end for
9:     end for
       return R
10: end function
```

preserving theorem that for any `I-conflict` set, there exists a restriction defined over one pair of shadow operations in the set so that it is impossible to have all shadow operations in G to not be ordered w.r.t each other in the R-order O. Therefore, G cannot be a maximal element set of the R-order O. Contradiction is found. ∎

5.4.3 Identifying restrictions

As discussed in the previous subsection, the key to making a replicated system adopt PoR consistency and strike an appropriate balance between performance and consistency semantics is to identify a finest set of restrictions, which ensure both state convergence and invariant preservation. With regard to the former property, we design a state convergence restrictions discovery method (Algorithm 1), which performs an operation commutativity analysis between pairs of operations. If two operations do not commute, then a restriction between them is added to the returning result restriction set.

For discovering the required restrictions for invariant preservation, we have to exhaustively explore all `I-conflict` sets that trigger violations. However, it is very challenging to achieve this since there might exist infinite number of violating executions containing at least one I-conflict set. Therefore, the exploration may not guarantee to terminate. To solve this problem, we decide to take a more efficient approach, in which we collapse many similar executions of a replicated system into a single execution class. To do so, we

Algorithm 2 I-conflict set discovery

```
1: function ICSETDISCOVER(T, ℘(T)) ▷ T: the set of operations of the target system, ℘(T) is the power set of T
2:     if T.processed == true or |T| == 0 then
         return
3:     end if
4:     result ← false          ▷ true indicates that a subset of T is I-conflict set.
5:     for j ← 2 to |T| − 1 do
6:         let ℘(T)_j be a subset of ℘(T) s.t. each element in ℘(T)_j has j operations.
7:         for all T′ ∈ ℘(T)_j do ICSETDISCOVER(T′, ℘(T′))
8:             result ← result|T′.isIConflict
9:         end for
10:    end for
11:    if result == false then ▷ No subsets of T are I-conflict set, so we need to check T.
12:        if |T| == 1 then                                  ▷ Check self-conflicting
13:            if ¬(T_0.post ⟹ T_0.wpre) then                ▷ T_0 is the 0-th element in T.
14:                T.isIConflict ← true
15:            end if
16:        else if |T| > 1 then
17:            for i ← 0 to |T| − 1 do                       ▷ T_i is the i-th element in T.
18:                post ← ∧_{x∈T\{T_i}} x.post
19:                if ¬(post ⟹ T_i.wpre) then
20:                    T.isIConflict ← true
21:                    break
22:                end if
23:            end for
24:        end if
25:    end if
26:    T.processed ← true
27: end function
```

use programming language techniques such as weakest precondition and postcondition analysis. For every operation u, we denote $u.wpre$ as its weakest precondition, which is a condition on the initial state and the parameter values ensuring that u always preserves invariants. We also denote $u.post$ as the postcondition summarizing the final state after the execution of u against all possible valid state. We flag a set of operations T as `I-conflict` if either of the following two conditions is met: (a) T contains a single operation t and t is self-conflicting, i.e., $t.wpre$ is invalidated by $t.post$; and (b) $|T| > 1$,

Algorithm 3 Invariant preservation restrictions discovery

```
1: function IPRDISCOVER(T, R)        ▷ T: the set of shadow operations of the target system, R: the restriction set
2:     ICSETDISCOVER(T, ℘(T))                    ▷ Compute all I-conflict sets
3:     for all T′ ∈ ℘(T) do
4:         if T′.isIConflict ==true then
5:             if |T′| == 1 then
6:                 R ← R ∪ {r(T′_0, T′_0)}              ▷ Restrict self-conflicting operations
7:             else if ∀u, v ∈ T′, r(u, v) ∉ R then ▷ This set has not been restricted yet.
8:                 R ← R ∪ {r(T′_i, T′_j)}, where i ≠ j and T′_i, T′_j ∈ T′   ▷ Restrict any pair of operations in T′
9:             end if
10:        end if
11:    end for
       return R
12: end function
```

Algorithm 4 Restriction set discovery

```
1: function DISCOVER(T)        ▷ T: the set of shadow operations of the target system
2:     R ← {}                                  ▷ the set of restrictions we identify
3:     R ← R ∪ SCRDISCOVER(T)     ▷ Identify restrictions ensuring state convergence
4:     R ← R ∪ IPRDISCOVER(T, R)          ▷ Identify restrictions ensuring invariant preservation
       return R
5: end function
```

any subset of T is not `I-conflict` (but can be self-conflicting) and there exists an operation u from T such that $u.wpre$ can be invalidated by the compound postcondition of operations in $T \setminus \{u\}$. (This procedure is implemented by Algorithm 2.)

To find a restriction set, for each identified `I-conflict` set T, we add a restriction between any pair of operations from T if no pairs of operations from that set is ever restricted. Otherwise, T will be skipped. This is because the relevant violating executions, where all shadow operations from T are not restricted, have been already eliminated, and hence there is no need to analyze T. (This procedure is implemented by Algorithm 3.)

To summarize, we devise four algorithms to discover a set of restrictions for ensuring state convergence and invariant preservation. The entrance algorithm `DISCOVER` (Algo-

```
1 opA (){
2   if (!B && !C){
3     A = true;
4   }
5 }
```

(a) opA

```
1 opB (){
2   if (!A && !C){
3     B = true;
4   }
5 }
```

(b) opB

```
1 opC (){
2   if (!A && !B){
3     C = true;
4   }
5 }
```

(c) opC

Figure 5.2: Pseudocode for the switch example where `opA`, `opB` and `opC` control switches A, B and C, respectively and the invariant is that A, B and C cannot be switched on at the same time. Initially, all three switches are off.

rithm 4) takes a set of shadow operation T as input. It first calls `SCDISCOVER` (Algorithm 1) to compute a set of restrictions R for ensuring state convergence. Then, it feeds `IPRDISCOVER` (Algorithm 3) the shadow operation set T and the state convergence restriction set R. The algorithm `IPRDISCOVER` (Algorithm 3) first calls `ICSETDISCOVER` (Algorithm 2) to discover all `I-conflict` sets and then adds a restriction between any pair of shadow operations from an `I-conflict` set accordingly. At the end, the algorithm `DISCOVER` outputs a set of restrictions to ensure both state convergence and invariant preservation.

5.4.4 Minimality

The invariant preservation theorem (Theorem 4) helps us verify whether a set of restrictions is sufficient to make a replicated system preserve invariants, but it doesn't preclude conservative cases, where unnecessary restrictions are present. The most promising solution is to prove that a set of restrictions is not only sufficient but also necessary. However, while playing with a few examples, we found that there might exist more than one effective restriction sets, where each of these sets is sufficient and any pair of them are not comparable, i.e., one is not included in the other, and vice versa. Therefore, to prove necessity becomes infeasible. As shown in Figure 5.2, to maintain the corresponding invariant, there are three incomparable coordination plans, namely $r(A, B)$, $r(B, C)$ or $r(A, C)$.

To overcome this challenge, we compromise our goal by proving the minimality of the restriction set we identify. There are a couple of criterions to define minimality, e.g., set inclusion, probability, cardinality and etc. In the context of PoR consistency, we define the minimality using set inclusion, since the cardinality solution is required to exhaustively search all effective restriction sets and this is not always possible.

Definition 16 (Minimality) *Given a PoR consistent system $\mathscr{S}$ with a set of restrictions $R_{\mathscr{S}}$ that preserves invariants, $R_{\mathscr{S}}$ is minimal if the following condition is met: for any restriction sets R' such that $R' \subsetneq R_{\mathscr{S}}$, there exists an execution of $\mathscr{S}$ against a valid state S_0 does not preserve invariants.*

The analysis algorithm we presented in the previous section would always output a minimal set of restrictions. We capture this in the following theorem:

Theorem 5 Minimality theorem: *Applying the restriction set discovery algorithm (Algorithm 4) to a system $\mathscr{S}$ generates a minimal set of restrictions for ensuring state convergence and invariant preservation under PoR consistency.*

Proof: We assume by contradiction that it is possible for the restriction set discovery algorithm (Algorithm 4) to generate a restriction $R_{\mathscr{S}}$, which is not minimal. Let $r(T_i, T_j)$ be one of the unnecessary restriction from $R_{\mathscr{S}}$. We know that any execution of $\mathscr{S}$ will not experience state divergence and invariant violation while removing $r(T_i, T_j)$ from $R_{\mathscr{S}}$. Let's consider the following two cases:

- $r(T_i, T_j)$ is produced by Algorithm 1, which finds restrictions for ensuring state convergence. It follows from the step pointed by the lines 5-6 in that algorithm that T_i and T_j do not commute w.r.t each other. It then follows from the operation commutativity concept and the state convergence definition that there exists an execution where T_i and T_j are not partially ordered and two causal legal serializations reach different states. Contradiction is found.

- $r(T_i, T_j)$ is produced by Algorithm 3, which finds restrictions for preserving invariants. It follows from the lines 5-8 in that algorithm that T_i and T_j belong to an `I-conflict set` T. If there exists a pair of operations from T other than $\langle T_i, T_j \rangle$ is restricted, then $r(T_i, T_j)$ should not be in $R_{\mathscr{S}}$. Contradiction is found. If no pairs of operations are restricted for T, then it follows from the I-conflict set definition (Definition 15) that removing $r(T_i, T_j)$ from $R_{\mathscr{S}}$ will make some executions of $\mathscr{S}$ observe invariant violations. Contradiction is found. ∎

5.5 Design and Implementation of Olisipo

In this section we provide a detailed explanation of the design and implementation of Olisipo, which adapts applications to run with SIEVE and Gemini under PoR consistency.

5.5.1 Design rationale

To minimize coordination overhead, in addition to applying the analysis presented in Section 5.4.3 for statically extracting a minimal set of restrictions, we aim to build an efficient coordination service for enforcing restrictions at runtime. This is challenging due for the following reason. We observed that there exist several coordination techniques/protocols that can be used for enforcing a given restriction, such as Paxos, distributed locking, or escrow techniques. However, depending on the frequency at runtime in which the system receives operations confined by a restriction, different coordination approaches lead to different performance tradeoffs. Therefore, the question we need to answer is: how to choose the cheapest protocol for enforcing a given restriction?

Consider the previously mentioned RUBiS example. In this example, maintaining the invariant that winners always match highest successful bidders requires a restriction between any pair of `placeBid'` and `closeAuction'` operations. The simplest coordination scheme would be forcing the two types of shadow operations to pay the same coordina-

tion cost for figuring out the existence of concurrent counterparts. However, this solution yields a very poor performance due to the imbalanced workload between the two types of shadow operations, i.e., `placeBid'` is more prevalent than `closeAuction'`. As a result, reducing the latency for `placeBid'` while maintaining the corresponding ordering constraint will comprise a better user experience.

In summary, we propose to build a specialized coordination service called Olisipo offering coordination policies, each of which presents a tradeoff between the cost of each operation and the overall cost. This service allows us to use runtime information about the relative frequency of operations to select an efficient coordination mechanism for a given restriction that has the lowest cost.

5.5.2 Coordination protocols

In this subsection, we present the two coordination techniques that we currently support in Olisipo and concrete scenarios where these mechanisms are more adequate. The two protocols we implemented are `symmetry (Sym)` and `asymmetry (Asym)`. Given a restriction $r(u, v)$ between two operations u and v, the `symmetry` protocol requires both u and v to coordinate with each other for establishing an order between them. In contrast, the `asymmetry` protocol provides different treatment for u and v by only requiring u (or v) to inform the counterpart operation in the restriction v (or u) about its existence, while allowing v (or u) to be executed fast without coordination if no u (or v) operations are running simultaneously. We further detail the two protocols as follows:

Sym. This protocol requires us to set up a logically centralized counter service, which maintains a counter for every shadow operation type present in a restriction $r(u, v)$, which we will refer to as c_u and c_v, and serializes reads and writes to these counters. Every such counter represents the total number of the corresponding operations that have been accepted by the underlying system. Additionally, every replica at different

data centers maintains a local copy of these counters, each of which represents the number of corresponding operations that have been observed by that replica. Initially, all local copies, as well as the global counters, have all values set to zero. Whenever an operation of type u is received by a replica, that replica contacts the counter service to increase the corresponding counter c_u and get a fresh copy of the counter maintained for v. Upon receiving the reply from the counter service, that replica can then compare the value of c_v with its local copy. If they are the same, then the replica can execute u without waiting. If the value is greater than the local copy, the local execution can only take place when all missing operations of type v have been locally replicated. Conversely, the same procedure is also applied to v. After replicating operations, the local copy of the counters will be brought to be up-to-date. In order to make the counter service fault tolerant, we leverage a Paxos-like state machine replication library (BFT-SMART [BSA14]) to replicate counters across geo-locations.

Asym. Unlike the above centralized solution, the asymmetry protocol implements distributed barrier in a decentralized manner as follows. Assume, for simplicity that u is the barrier. In this case whenever a replica r receives an operation u it would have to enter the barrier, and contact all other replicas to request participation. This requires all replicas in the system to stop processing operations of type v and enter the barrier. After receiving an acknowledgment of the barrier entrance from all replicas, r can execute the operation, and then notify all replicas that it has left the barrier (while at the same time propagating the effects of the operation u it has just executed). Such a coordination strategy might incur in a high overhead; however, it might be interesting when one of the two operations in the restriction is rarely submitted to the system. For instance, in the auction example, `closeAuction'` is a candidate for being used as barrier, since `placeBid'` dominates the operation space.

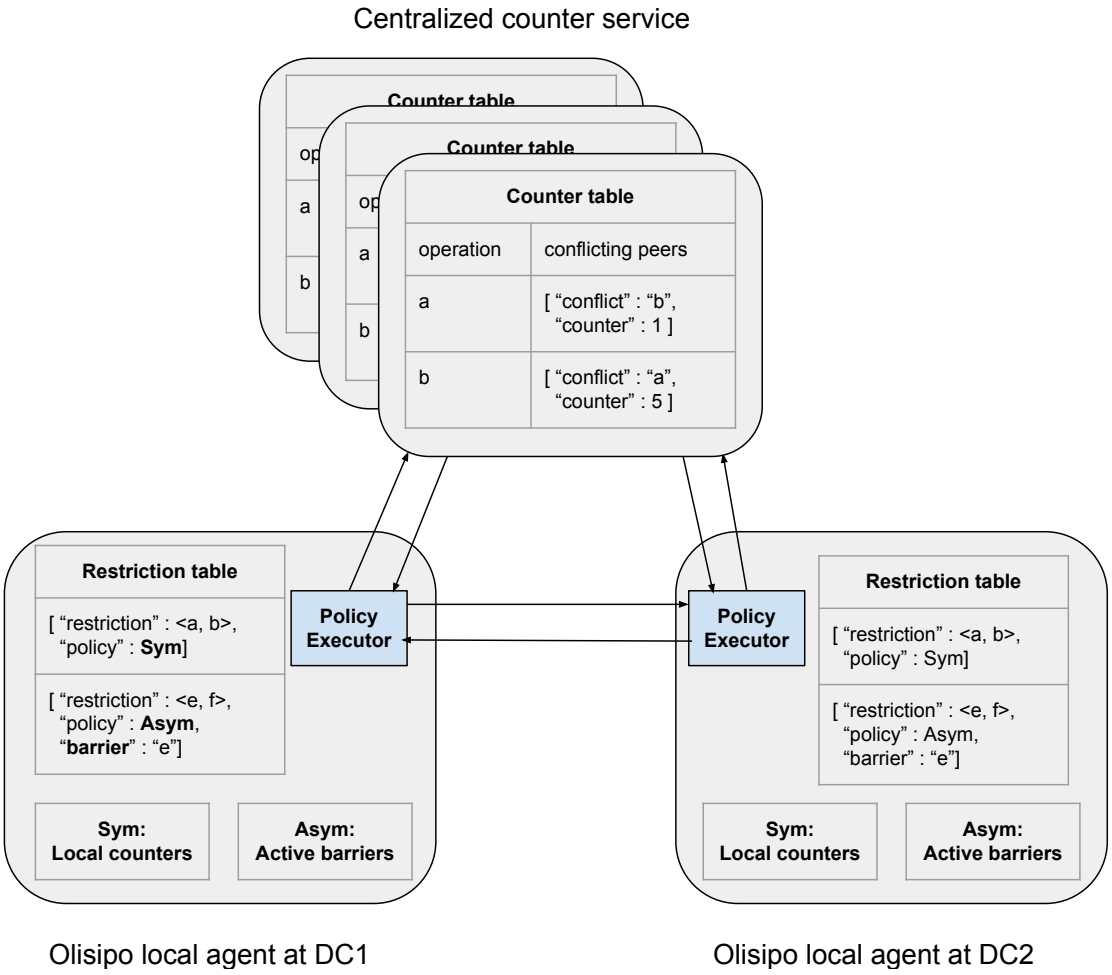


Figure 5.3: Olisipo architecture

5.5.3 Architecture

All design choices and details presented above lead to the high level system architecture depicted in Figure 5.3. The Olisipo architecture consists of a counter service replicated across data centers and a local agent deployed in every data center. While the counter service is required by executing the `Sym` protocol for keeping track of the number of different operations that have been accepted by the system, the local agent is responsible for placing coordination only when the corresponding operation is confined by restrictions. Every local agent keeps a restriction table, which defines all identified restrictions

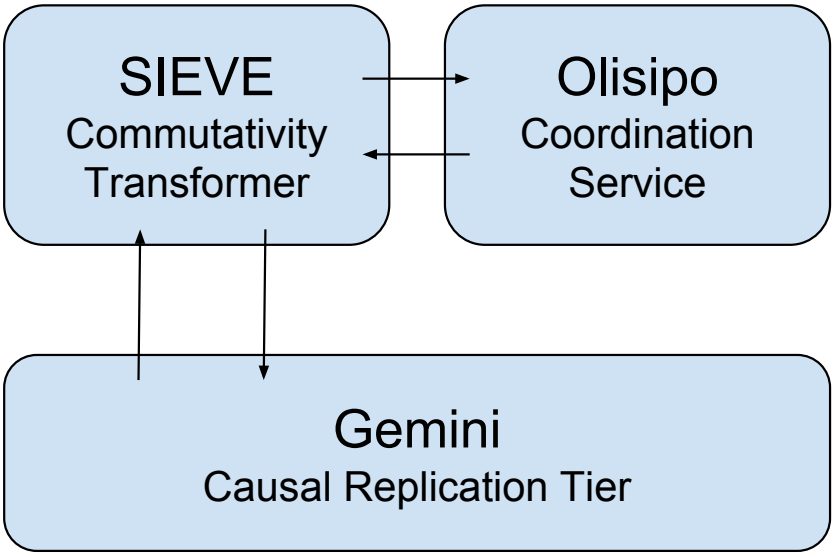


Figure 5.4: Olisipo connected with SIEVE and Gemini

between pairs of operations and the corresponding coordination policy. In addition, every agent also stores some meta data required for different protocols: With regard to the `Sym` protocol, it maintains a local copy of the replicated counter service, which is used for learning if the local counters lag behind the global counters, which means the corresponding data centers have to wait until all missing operations have been locally incorporated. For the `Asym` protocol, every agent maintains a list of active barriers, which are used for locally deciding if relevant operations blocked on such barriers can proceed.

5.5.4 Implementation

We implemented Olisipo using Java ($2.8k$ lines of code)[2], and BFT-SMART [SAB] for replicating the state of the centralized counter service, MySQL as the backend storage, and Netty as the communication library [Net]. As shown in Figure 5.4, we integrated Olisipo with Gemini and SIEVE so that Gemini serves as the underlying causally con-

[2]The number of lines of code is measured by `cloc` [cod].

sistent replication tier while SIEVE is used to produce commutative shadow operations at runtime. The source code of Olisipo is available at [oli].

Workflow. A user issues her request to an application server located at the closest data center, which runs an instance of SIEVE (introduced in Chapter 4) and a local agent shown in Figure 5.3. SIEVE intercepts the communication between the app server and the backend MySQL database and executes the corresponding generator operation. When the execution ends, SIEVE produces a commutative shadow operation that accumulates side effects of that request, and then asks the local Olisipo agent for placing coordination if needed before committing and replicating that shadow operation. To do so, the Olisipo agent looks up the restriction table to determine if that operation is confined by any restriction. If so, then the `policy executor` of Olisipo orders that operation with respect to all its conflicting operations that are running concurrently at other data centers. This is achieved by executing different protocols according to the lookup result. When conflicting operations are serialized, SIEVE sends these operations to Gemini for replicating them across all data centers while respecting the established order.

5.6 Evaluation

Concerning the evaluation of Olisipo, we focus on two main aspects. First, we want to understand if the methodology for inferring restrictions presented in Section 5.4 is effective when applied to real world applications, i.e., it finds a minimal set of restrictions. Second, we explore the impacts on user observed latency and system throughput introduced by three factors: adopting PoR consistent replication, using different protocols, and adding more restrictions.

5.6.1 Case study

We apply the analysis (Algorithm 4) to an extended version of RUBiS (which implements a `closeAuction` operation for declaring auction winners) to identify a set of restrictions comprising a minimal amount of coordination without sacrificing either state convergence or invariant preservation. This subsection reports our experience on conducting such analysis and the final static result we obtained.

State convergence. As we deploy RUBiS alongside SIEVE, all shadow operations generated at runtime commute w.r.t each other and there is no need to restrict any pair of shadow operations. The final output of the state convergence restriction discovery method (Algorithm 1) is an empty restriction set.

Invariant preservation. We determined four invariants of the extended version of RUBiS, namely (a) identifiers assigned by the system are unique; (b) nicknames chosen by users are unique; (c) item stock must be non-negative; and (d) the auction winner must be associated with the highest bid across all accepted bids. We continued by performing the `I-conflict set` analysis (Algorithm 2) against all RUBiS shadow operations. With regard to the first invariant, since we take advantage of the coordination-free unique identifier generation method offered by SIEVE, no `I-conflict sets` were found for violating it. In contrast, for the remaining three invariants, we identified the following `I-conflict` sets:

- $\{registerUser', registerUser'\}$. Invariant (b) would be violated if the two operations proposed the same *nickname* and were submitted to different sites simultaneously;
- $\{storeBuyNow', storeBuyNow'\}$. Invariant (c) would be violated if both operations simultaneously deducted a positive number from *stock* while *stock* was not enough;

App	RedBlue consistency	PoR consistency
RUBiS	$r(registerUser', registerUser')$ $r(storeBuyNow', storeBuyNow')$ $r(placeBid', placeBid')$ $r(closeAuction', closeAuction')$ $r(placeBid', closeAuction')$ $r(registerUser', storeBuyNow')$ $r(registerUser', placeBid')$ $r(registerUser', closeAuction')$ $r(storeBuyNow', placeBid')$ $r(storeBuyNow', closeAuction')$	$r(registerUser', registerUser')$ $r(storeBuyNow', storeBuyNow')$ $r(placeBid', closeAuction')$

Table 5.1: Restrictions over pairs of shadow operations that are required when replicating the extended RUBiS under RedBlue or PoR consistency

- $\{placeBid', closeAuction'\}$. Invariant (d) would be violated if both operations were submitted at the same time to different sites, and $placeBid'$ carried a higher bid than all accepted bids.

Each of the three above `I-conflict sets` covers a class of violating executions of the respective invariant. To eliminate the corresponding violations, we added three restrictions, namely $r(registerUser', registerUser')$, $r(storeBuyNow', storeBuyNow')$ and $r(placeBid', closeAuction')$, which are summarized in Table 5.1. This set is a minimal restriction set since it is sufficient to ensure the two important properties and none of these restrictions can be removed. In contrast, compared to the PoR consistency solution, replicating RUBiS via RedBlue consistency would require more restrictions, since the definition states that all non-invariant safe shadow operations must be red (strongly consistent), i.e., the four shadow operations presented in the above list must be restricted in a pair-wise fashion, as shown in Table 5.1.

5.6.2 Experimental setup

Deployment parameters. We run experiments on Amazon EC2 [Amaa] using `m4.2xlarge` virtual machine instances located in three sites: US Virginia (US-East),

	US-East	US-West	EU-FRA
US-East	0.299 ± 0.042 ms 1052.0 ± 0.0 Mbps	71.200 ± 0.021 ms 47.4 ± 1.6 Mbps	88.742 ± 1.856 ms 29.6 ± 5.6 Mbps
US-West	66.365 ± 0.006 ms 47.4 ± 1.6 Mbps	0.238 ± 0.003 ms 1050.7 ± 4.1 Mbps	162.156 ± 0.179 ms 17.4 ± 1.7 Mbps
EU-FRA	88.168 ± 0.035 ms 36.2 ± 0.1 Mbps	162.163 ± 0.157 ms 20.1 ± 0.1 Mbps	0.226 ± 0.003 ms 1052.0 ± 0.0 Mbps

Table 5.2: Average round trip latency and bandwidth between Amazon datacenters (obtained in Dec 2015).

US California (US-West) and EU Frankfurt (EU-FRA), which are the latest generation of General Purpose Instances. Table 5.2 shows the average round trip latency and observed bandwidth between every pair of sites. Each VM has 8 virtual cores and 32GB of RAM. VMs run Debian 8 (Jessie) 64 bit, MySQL 5.5.18, Tomcat 6.0.35, and OpenJDK 8 software.

Configuration and workloads. Unless stated otherwise, in all experiments, we deploy the BFT-SMART library under the crash-fault-tolerance model (CFT) with 3 replicas across three sites, and assign the replica at EU-FRA to act as the leader of the consensus protocol. We replicate RUBiS under PoR consistency across three sites using Olisipo, SIEVE, and Gemini, while running an unreplicated strongly consistent RUBiS in the EU-FRA site as a baseline. We refer to the first setup as "Olisipo-3-datacenter", and to the second setup as "Unreplicated". For all experiments, emulated clients are equally distributed across three sites and connect to their closest data center according to physical proximity.

We choose to run the bidding mix workload of RUBiS, where 15% of user interactions are updates. To allow the client emulator to issue the newly introduced `closeAuction` requests, we have to slightly change the transition table equipped with the original RUBiS code by assigning a positive probability value for this request. The new transition table can be found here [Oli15]. For all experiments we vary the workload by increasing

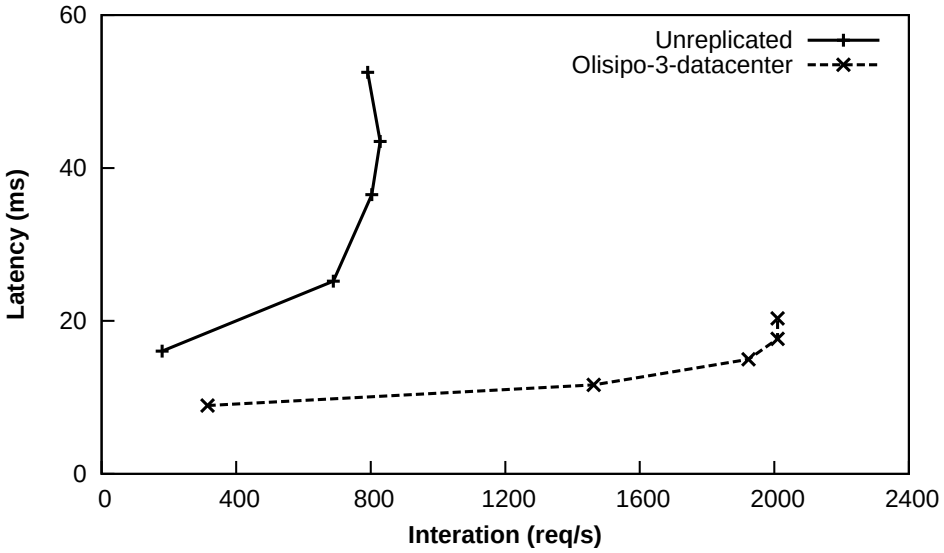


Figure 5.5: Throughput versus latency curves for the RUBiS bidding mix.

the number of concurrent client threads in every client emulator. We also disable the `thinking time` option for issuing requests so that there is no waiting time between two contiguous requests from the same client thread. With regard to the data set, we populate it via the following parameters: the RUBiS database contains 33,000 items for sale, 1 million users, and 500,000 old items.

5.6.3 Experimental results

In this part, we analyze the results obtained from running the experiments stated above concerning the following aspects.

Overall performance

We start by looking into the overall performance comparison between a 3 site Olisipo deployment of RUBiS, which offers fine-grained tradeoffs between consistency and performance, and a single site original code deployment, which provides strong consistency. Figure 5.5 shows the overall average latency and throughput curves of the two experi-

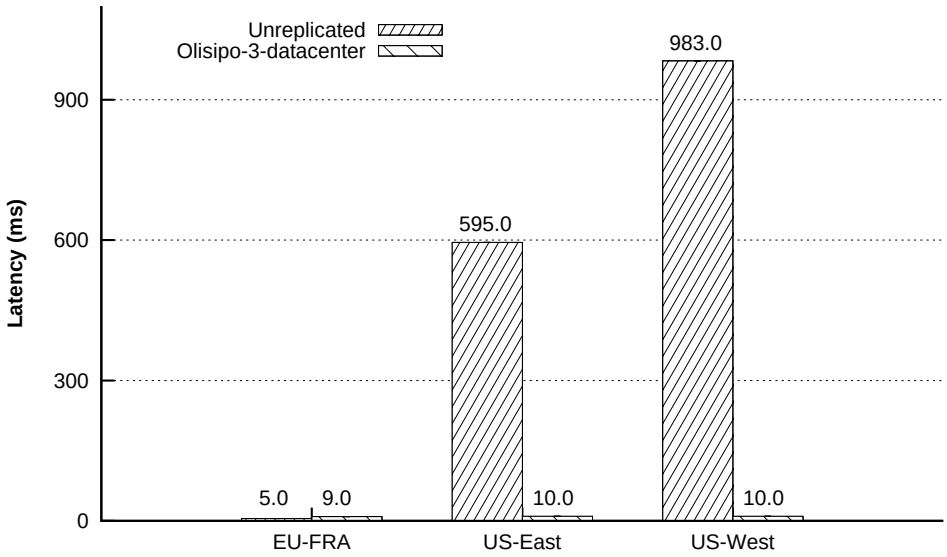


Figure 5.6: Overall average latency bar graph for users located in three sites.

ments. Olisipo significantly outperforms the unreplicated RUBiS deployment in two dimensions, namely, Olisipo reduces average latency (44.3% lower for the first data point from left to right) and improves peak throughput (142.8% higher). The performance gains come from the fact that Olisipo is able to execute non-conflicting requests in a coordination-free manner and to employ an efficient coordination policy when needed for processing conflicting requests.

User perceived latency

The major concern of designing Olisipo is to reduce the user perceived latency. In order to understand the effectiveness of Olisipo on this front, we break down the overall latency shown in Figure 5.5 into the following categories.

Intra-data center. First, we analyze the average latency for users at each data center. As shown in Figure 5.6, all users except those in EU-FRA observe notably lower latency in the Olisipo experiment, compared to the users from the same locations in the unreplicated experiment. This improvement is because, in Olisipo, most of the requests

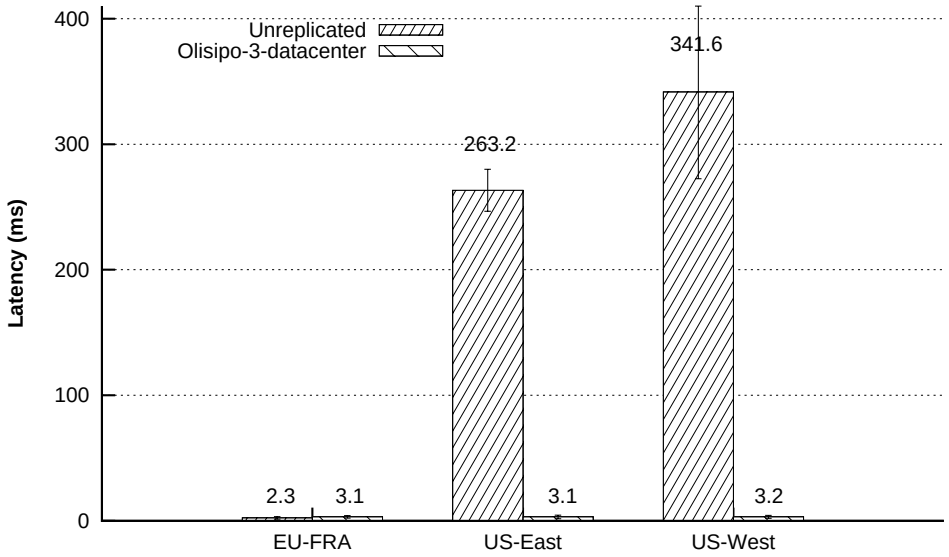


Figure 5.7: Average latency bar graph of a RUBiS request `storeComment` for users located at three sites. In the context of PoR consistency, this request is non-conflicting and hence does not require coordination.

are handled locally, while in the unreplicated RUBiS, requests from users at the two US data centers have to be redirected to EU-FRA, which incurs expensive inter-datacenter communication. Unlike users at these two data centers, we observed that users at EU-FRA in the Olisipo experiment experience a slightly higher latency than users from the same region accessing an unreplicated RUBiS. This can be explained by the additional work required for incorporating remote shadow operations into the local causal serialization and placing coordination when needed for serializing conflicting requests. Note that although the user observed latency for Olisipo at EU-FRA is almost twice as large as the latency of the unreplicated experiment, the absolute number (9 ms) is reasonably low.

Latency of non-conflicting requests. Among all non-conflicting requests in RUBiS, we chose one representative request called `storeComment` as the illustrating example, which places a comment on a user profile. As depicted in Figure 5.7, the conclusion we can draw from this graph is consistent with the one regarding Figure 5.6. However,

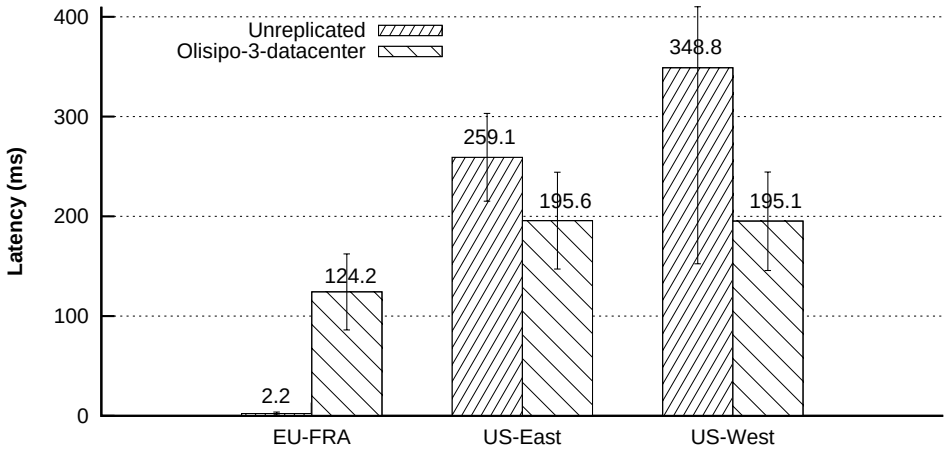


Figure 5.8: Average latency bar graph of a RUBiS request `storeBuyNow` for users located at three sites. In the context of PoR consistency, `storeBuyNow` conflicts w.r.t itself and is regulated by the `Sym` protocol when being replicated.

the major difference between these two figures is that users from EU-FRA in both experiments have almost identical latency. This is because the `storeComment` request requires no coordination and the cost of generating and applying the corresponding shadow operation is modest.

Latency of conflicting requests. Third, we shift our attention from non-conflicting requests to conflicting ones. As introduced before, Olisipo uses two different protocols (`Sym` and `Asym`) to coordinate conflicting requests. We start by analyzing the latency of requests handled by the `Sym` protocol. The illustrative example we selected is `storeBuyNow`, which is conflicting with respect to itself. As shown in Figure 5.8, the user observed latency of the `storeBuyNow` request at all three sites is significantly higher than the latency of `storeComment` (shown in Figure 5.7), which is a non-conflicting request. This is because most of the lifecycle of these requests was spent asking the centralized counter service for granting permissions, which consists of 3 replicas spanning three sites and executing a Paxos-like consensus protocol. Additionally, user observed latency at EU-

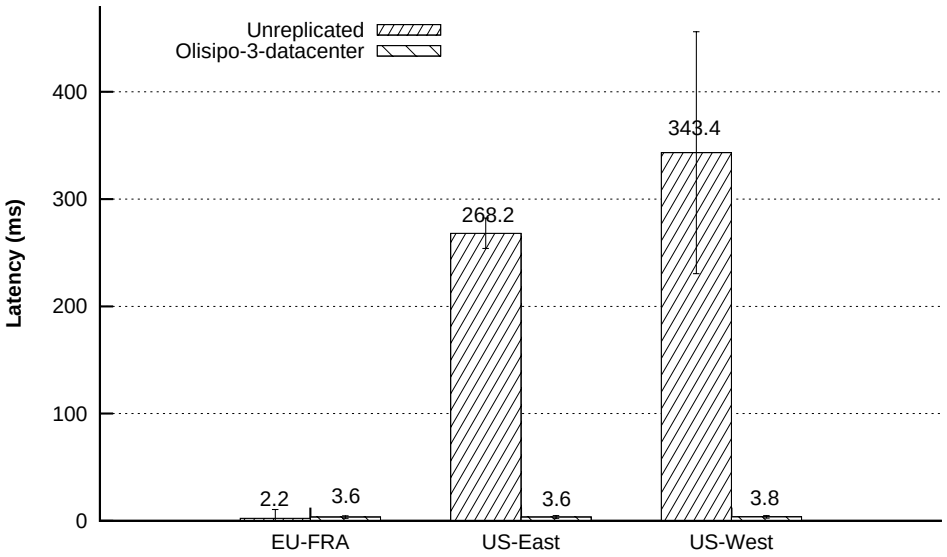


Figure 5.9: Average latency bar graph of a `placeBid` request for users locating in three sites, which is conflicting with `closeAuction`. This request is regulated by the `Asym` protocol but is not a barrier.

FRA is lower than the remaining two sites, since the leader of the consensus protocol is co-located with EU-FRA users.

We continue by analyzing the average latency of requests that are coordinated by the `Asym` protocol. Unlike the `Sym` protocol, any pair of operations confined in a restriction will be treated differently by the `Asym` protocol, namely one acts as a distributed barrier and the other proceeds if no active barriers are running. In the case study section (Section 5.6.1), we assign the `Asym` protocol to regulate the $r(placeBid', closeAuction')$ restriction, while selecting the less frequent shadow operation $closeAuction'$ to work as a barrier. As shown in Figure 5.9, the average latency measured for the *placeBid* request, which produces $placeBid'$, looks very similar to the results obtained for non-conflicting requests shown in Figure 5.7. This is because the ratio of *closeAuction* to *placeBid* is very low (2.7%) and most of the time the *placeBid* request commits immediately without waiting for joining or leaving barriers.

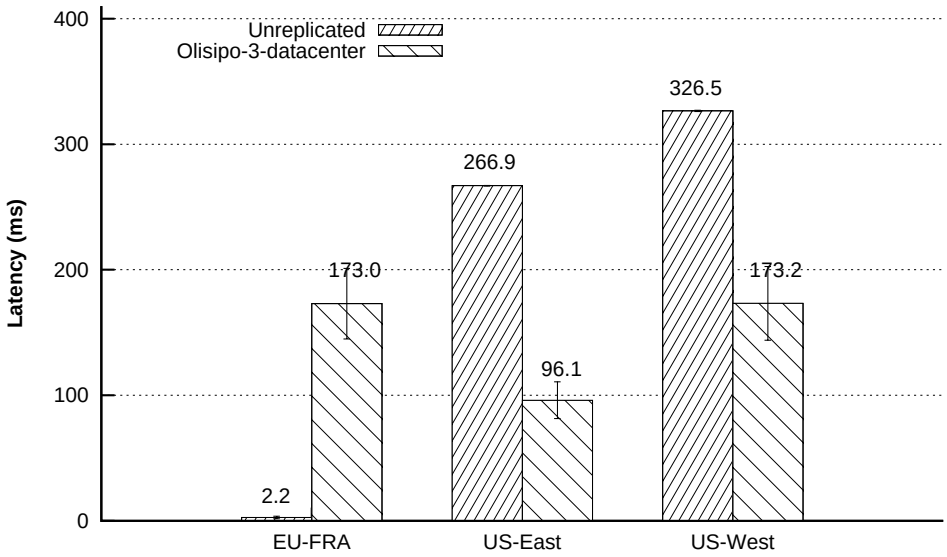


Figure 5.10: Average latency bar graph of a `closeAuction` request, which is conflicting with `placeBid`. This request is handled by the `Asym` protocol and acts as the barrier.

Next, we consider the barrier request *closeAuction* handled by the `Asym` protocol. As expected, compared to *placeBid*, the average latency of *closeAuction* is remarkably higher due to the coordination across sites, through which this request forces all sites not to process incoming *placeBid* requests and collects results of all relevant completed *placeBid* requests. As shown in Figure 5.10, users issuing *closeAuction* observed a latency slightly higher than the maximal RTT between their primary data center and the remaining data centers. For example, as shown in Table 5.2, the maximal RTT on average for US-East users is 88.7 ms, while the average latency of *closeAuction* observed by the same group of users is 96.1 ms.

Impact of different protocols

As motivated in the design of Olisipo, the purpose of offering different coordination protocols is to improve runtime performance by taking into account the workload characteristics. To validate this, we first deploy an experiment denoted by `Olisipo-Correct-Usage`,

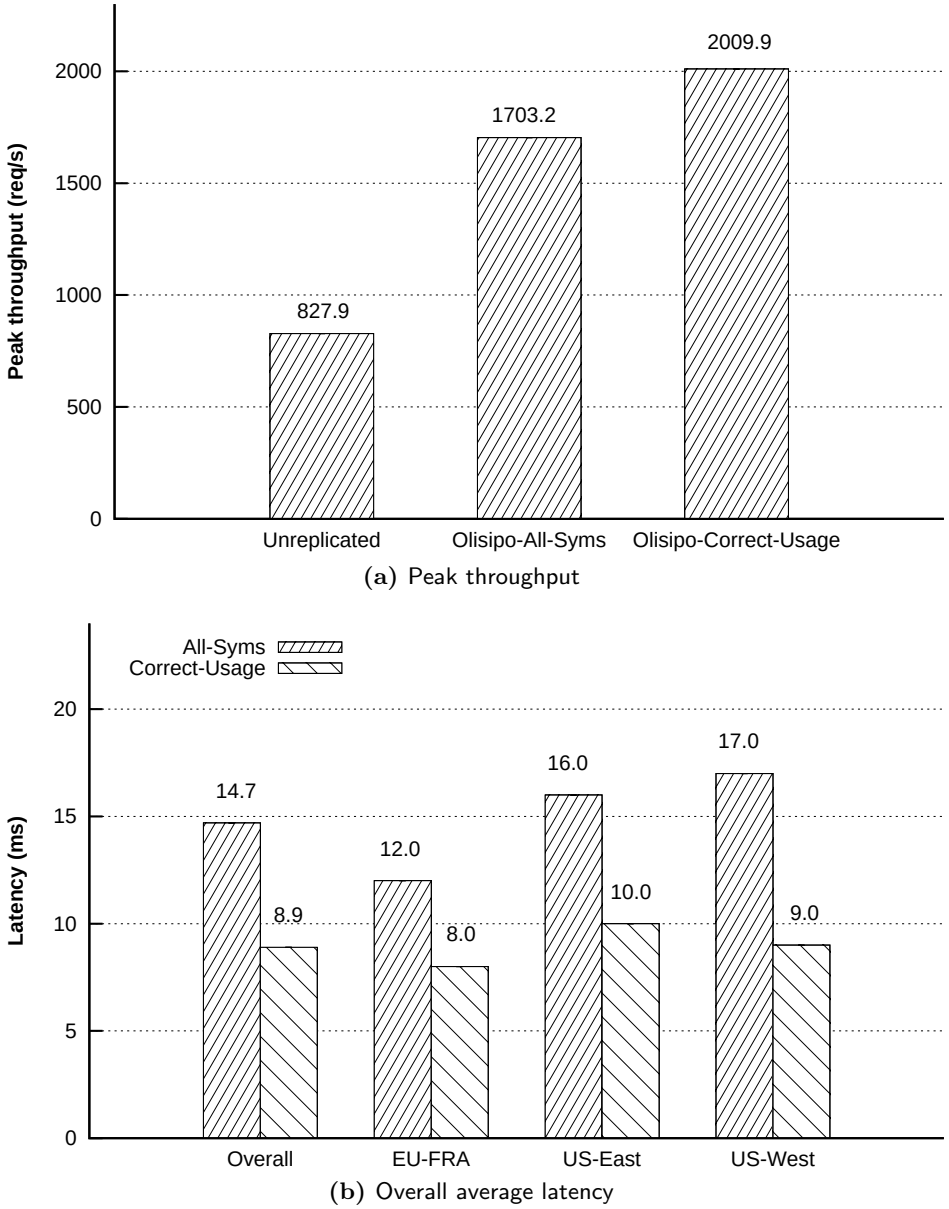


Figure 5.11: Peak throughput and overall average latency bar graphs of systems using different protocols.

in which we take into account the runtime information that $closeAuction'$ occurs sparsely and assign the `Asym` protocol to regulate the restriction $r(placeBid', closeAuction')$. We then deploy another experiment denoted by `Olisipo-All-Syms`, in which the restriction $r(placeBid', closeAuction')$ is handled by the `Sym` protocol. Figure 5.11 sum-

marizes the comparison of peak throughput and average latency among three experiments, namely `Unreplicated`, `Olisipo-All-Syms` and `Olisipo-Correct-Usage`. The `Olisipo-All-Syms` setup improves the peak throughput of the unreplicated RUBiS system by 105.7%, because of the coordination-free execution of non-conflicting requests. However, compared to `Olisipo-Correct-Usage`, the performance of `Olisipo-All-Syms` degrades in two dimensions, namely a 15.3% decrease in peak throughput and a 65.2%, 50.0%, 60.0%, 88.9% increase in request latency for all, EU-FRA, US-East, US-West users, respectively. The reason for this performance loss is as follows: every `placeBid'` shadow operation in `Olisipo-All-Syms` requires a communication step between its primary site and the centralized counter service for being coordinated, while most of time `placeBid'` shadow operations in `Olisipo-Correct-Usage` work as non-conflicting requests provided that `closeAuction` requests sparsely arrive in the system.

Impact of the number of restrictions

The last aspect of our evaluation is to explore the impact on latency and throughput introduced by varying the number of restrictions. To this end, we deploy a baseline experiment denoted by `RedBlue`, in which we replicate RUBiS via the PoR consistency framework but with the set of restrictions (shown in Table 5.1) we identified in the context of RedBlue consistency. The comparison between the unreplicated RUBiS, RedBlue consistent RUBiS and PoR consistent RUBiS is summarized in Figure 5.12. The improvement on scalability by RedBlue consistency looks similar to the result we obtained in Chapter 3. For example, as shown in Figure 5.12(a), a 3 site RedBlue replication improves peak throughput offered by the unreplicated strongly consistent solution by 99.7%. However, compared to PoR consistent RUBiS, due to the unnecessary restrictions enforced by RedBlue consistency, RedBlue consistent RUBiS achieves worse performance, namely a 19.2% decrease in peak throughput and a 67.8%, 62.5%, 60.0%, 88.9% increase in request latency for all, EU-FRA, US-East, US-West users, respectively.

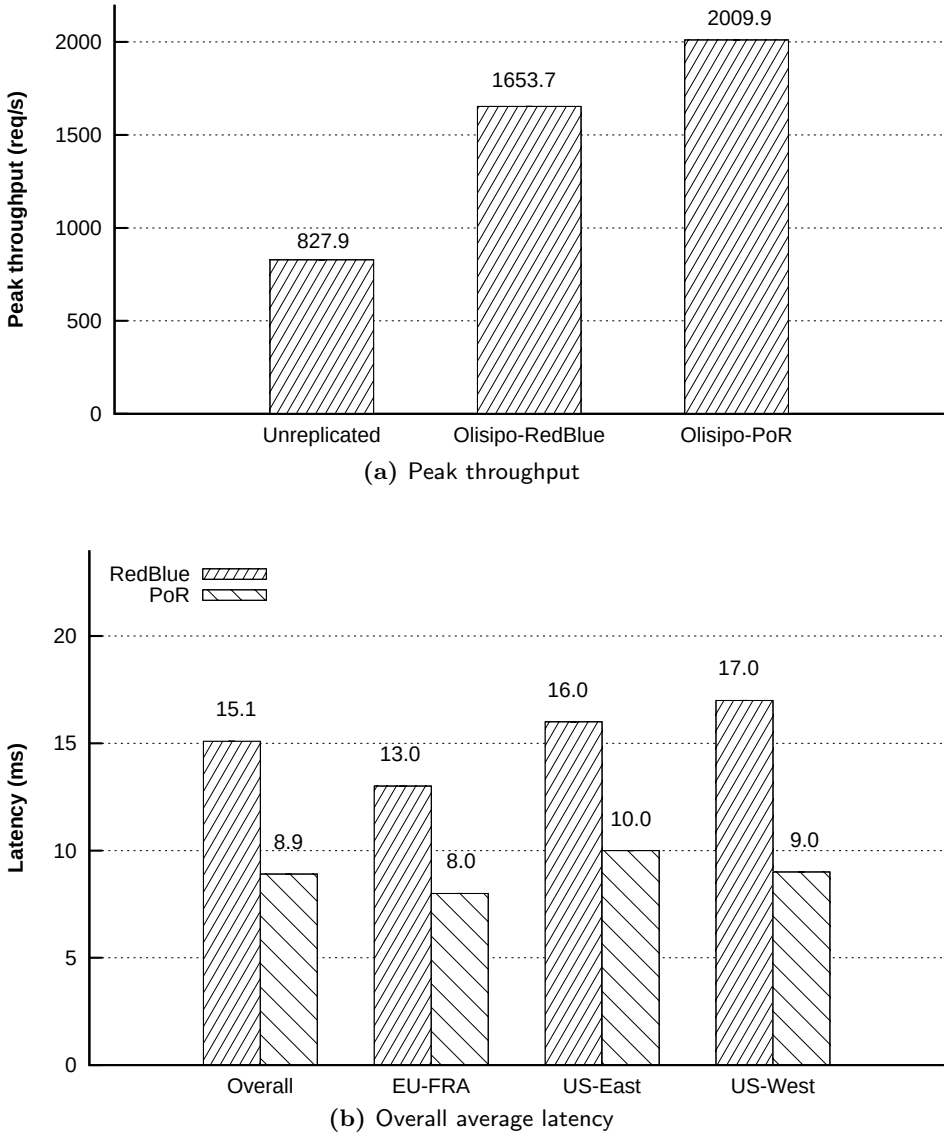


Figure 5.12: Peak throughput and overall average latency bar graphs of RedBlue consistency and PoR consistency.

5.7 Limitations and future work

While adapting applications to use PoR consistency and Olisipo significantly outperforms the usage of RedBlue consistency and SIEVE, there are several interesting unexplored avenues for future work:

First, there might exist some restrictions that do not need to be symmetric, e.g., given two operations u, v, the order restriction where u always follows v is both sufficient and necessary to maintain all target system properties. We have not explored the existence of asymmetric restrictions, nor assessed the impact of having asymmetric restrictions on the coordination cost. We leave this exploration to our future work.

Second, the adoption of PoR consistency requires the programmer to manually apply the previously described static analysis (seen in Algorithm 4) for determining the possibility of diverging state or violating invariants, in order to obtain the minimal set of restrictions. To free this programming burden, we plan to develop a tool to automate this analysis.

Third, the current implementation of Olisipo only embraces two different coordination protocols, each of which is suitable for a certain workload. In the future, we plan to incorporate other coordination protocols into Olisipo, so that the programmer can make a better choice.

Fourth, we plan to add an agent to Olisipo, which dynamically measures the frequencies of different operations and makes runtime decisions for switching from a protocol to another more efficient one.

5.8 Summary

In this chapter, we proposed a research direction for building fast and consistent geo-replicated systems that employ a minimal amount of coordination in order to achieve both invariant preservation and state convergence. To this end, we first defined a new generic consistency model called PoR consistency, which maps consistency requirements to fine-grained restrictions over pairs of operations. Second, we developed a static analysis to infer, for a given application, a minimal set of restrictions for ensuring the two previously mentioned properties, in which no restrictions can be removed and no new restrictions need to be added. Third, we built an efficient coordination service called

Olisipo for coordinating conflicting operations. Our evaluation of running RUBiS with different setups shows that the joint work of PoR consistency and Olisipo significantly improves the system performance of geo-replicated systems.

6 Conclusion

In the recent few years, (geo-)replication has been widely adopted to build scalable services that offer low latency access and high throughput, in order to meet their unprecedented user demands. However, this goal is often negatively affected by the coordination required to ensure application-specific properties such as state convergence and invariant preservation. This dissertation shows that differentiating the consistency requirements for various operations and executing operations with different amounts of coordination can make replicated services fast as possible while ensuring their targeted consistency semantics.

In short, our approach consists of the following three major components: (a) RedBlue consistency, a novel consistency definition, which offers a coarse-grained choice between executing an operation under either strong consistency or weak (causal) consistency; (b) SIEVE, a tool that automatically makes a decision on which consistency level to be assigned to an operation in the context of RedBlue consistency; and (c) PoR consistency, another novel consistency definition generalizes the tradeoffs behind RedBlue consistency, offers a fine-grained choice in consistency requirements for various operations and reduces the amount of required coordination when possible.

RedBlue consistency allows strongly and weakly consistent operations to coexist in a single system and defines a set of sufficient conditions to determine the appropriate consistency levels for various operations by analyzing whether running operations in parallel can make state diverge or become invalid. In short, an operation must be red (strongly

consistent) if either it does not commute with any other operation or it potentially breaks invariants in the presence of concurrency; otherwise, it can be blue (causally consistent). To address the problem that many original operations do not naturally commute, we propose to decouple the execution of an operation into a generator operation to decide the changes, which has no side effects, and a shadow operation to apply the identified changes in a commutative fashion across all replicas. Finally, we built Gemini, which is a distributed coordination and replication tier for making web applications RedBlue consistent.

To the best of our knowledge, SIEVE is the first system to automate the choice of consistency levels offered by multi-level consistency in a replicated system. It relieves the programmer from having to (a) construct commutative shadow operations; and (b) reason about the behaviors that weak consistency introduces, only requiring the programmer to write the system invariants that must be preserved and provide a small number of annotations regarding merge semantics. To automate step (a), we leverage CRDTs to translate every SQL statement into commutative forms. To automate step (b), we rely on weakest precondition analysis techniques to determine sufficient conditions, under which the corresponding shadow operations can be invariant safe. At runtime, an efficient evaluation of such conditions will tell whether strong consistency or weak consistency should be used.

PoR consistency has a broader view of the tradeoffs between maintaining targeted consistency semantics and improving performance, by expressing this semantics as visibility restrictions between pairs of operations. Weakening or strengthening the consistency semantics in the context of PoR consistency is achieved by imposing fewer or more restrictions over relevant operations. In order to minimize the amount of required coordination, we developed a concept called `I-conflict set`, which captures the finest composition of shadow operations corresponding to an invariant violation, and a sequence of algorithms to explore `I-conflict` sets and add the relevant restrictions. Finally, to help

programmers make use of PoR consistency, we built an efficient coordination service called Olisipo, which allows the programmer to choose the most lightweight protocol for replicating operations while serializing any pair of operations that need to be restricted.

Bibliography

[AC09] Farhana Aleen and Nathan Clark. Commutativity Analysis for Software Parallelization: Letting Program Transformations See the Big Picture. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 241–252, New York, NY, USA, 2009. ACM.

[ACHM11] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency Analysis in Bloom: a CALM and Collected Approach. In *Proceedings of the Fifth Biennial Conference on Innovative Data Systems Research*, CIDR'11, 2011.

[ACHM14] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and David Maier. Blazes: Coordination Analysis for Distributed Programs. In *Proceedings of the IEEE 30th International Conference on Data Engineering*, ICDE'14, 2014.

[ALaR13] Sérgio Almeida, João Leitão, and Luís Rodrigues. ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 85–98, New York, NY, USA, 2013. ACM.

[Amaa] Amazon Elastic Compute Cloud (EC2). https://aws.amazon.com/ec2/. [Online; accessed Feb-2016].

[Amab] Amazon Web Services (AWS) - Cloud Computing Services. http://aws.amazon.com/. [Online; accessed Feb-2016].

[ANB+94] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and P.W. Hutto. Causal Memory: Definitions, Implementation and Programming. Technical report, Georgia Institute of Technology, 1994.

[BAC+13] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 49–60, Berkeley, CA, USA, 2013. USENIX Association.

[BBC+11] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research*, CIDR'11, pages 223–234, 2011.

[BDF+15] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Putting Consistency Back into Eventual Consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 6:1–6:16, New York, NY, USA, 2015. ACM.

[BFF+14] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination Avoidance in Database Systems. *Proc. VLDB Endow.*, 8(3):185–196, November 2014.

[BGY13] Sebastian Burckhardt, Alexey Gotsman, and Hongseok Yang. Understanding Eventual Consistency. Technical Report MSR-TR-2013-39, March 2013.

[BHG87] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. 1987.

[bin] Microsoft Bing Web Page. http://www.bing.com/. [Online; accessed Feb-2016].

[BKL+10] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a Needle in Haystack: Facebook's Photo Storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[BRCA09] Fabrício Benevenuto, Tiago Rodrigues, Meeyoung Cha, and Virgílio Almeida. Characterizing User Behavior in Online Social Networks. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '09, pages 49–62, New York, NY, USA, 2009. ACM.

[BSA14] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. State Machine Replication for the Masses with BFT-SMART. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 355–362, Washington, DC, USA, 2014. IEEE Computer Society.

[CCA08] Emmanuel Cecchet, George Candea, and Anastasia Ailamaki. Middleware-based Database Replication: The Gaps Between Theory and Practice. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 739–752, New York, NY, USA, 2008. ACM.

[CDE+12] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-distributed Database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[CKZ+13] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 1–17, New York, NY, USA, 2013. ACM.

[cod] Count Lines of Code. http://cloc.sourceforge.net/. [Online; accessed Dec-2015].

[con02] TPCW consortium. TPC Benchmark-W Specification v. 1.8. http://www.tpc.org/tpcw/spec/tpcw_v1.8.pdf, 2002. [Online; accessed Feb-2016].

[CRS+08] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[DHJ+07] 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 *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[eba12] Ebay website. http://www.ebay.com/, 2012. [Online; accessed Feb-2016].

[EG89] C. A. Ellis and S. J. Gibbs. Concurrency Control in Groupware Systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, pages 399–407, New York, NY, USA, 1989. ACM.

[EJ09] Cecchet Emmanuel and Marguerite Julie. RUBiS: Rice University Bidding System. http://rubis.ow2.org/, 2009. [Online; accessed Feb-2016].

[Fac] Welcome to Facebook - Log In, Sign Up or Learn More. https://www.facebook.com/. [Online; accessed Feb-2016].

[FB213] Facebook Annual Report 2013. http://investor.fb.com/annuals.cfm, 2013. [Online; accessed Aug-2014].

[FJL+97] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *IEEE/ACM Trans. Netw.*, 5(6):784–803, December 1997.

[Fog12] Fogbeam Labs. Quoddy Code Repository. http://fogbeam.github.io/Quoddy/, 2012. [Online; accessed Feb-2016].

[FZFF10] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. SPORC: Group Collaboration Using Untrusted Cloud Resources. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–, Berkeley, CA, USA, 2010. USENIX Association.

[Gem] Gemini Code Repository. https://github.com/pandaworrior/RedBlue_consistency. [Online; accessed Feb-2016].

[GHOS96] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The Dangers of Replication and a Solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 173–182, New York, NY, USA, 1996. ACM.

[Gooa] Google Webpage. www.google.com. [Online; accessed Feb-2016].

[Goob] Google. Balancing Strong and Eventual Consistency with Google Cloud Datastore. https://cloud.google.com/developers/articles/balancing-strong-and-eventual-consistency-with-google-cloud-datastore. [Online; accessed Jan-2014].

[Gra81] Jim Gray. The Transaction Concept: Virtues and Limitations (Invited Paper). In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB '81, pages 144–154. VLDB Endowment, 1981.

[GYF+15] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'Cause I'm strong enough: reasoning about consistency choices in distributed systems. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'15, New York, NY, USA, 2015. ACM.

[GYG+14] Ashish Gupta, Fan Yang, Jason Govig, Adam Kirsch, Kelvin Chan, Kevin Lai, Shuo Wu, Sandeep Govind Dhoot, Abhilash Rajesh Kumar, Ankur Agiwal, Sanjay Bhansali, Mingsheng Hong, Jamie Cameron, Masood Siddiqi, David Jones, Jeff Shute, Andrey Gubarev, Shivakumar Venkataraman, and Divyakant Agrawal. Mesa: Geo-replicated, Near Real-time, Scalable Data Warehousing. *Proc. VLDB Endow.*, 7(12):1259–1270, August 2014.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[jav13] The Web Page of JavaParser. http://javaparser.github.io/javaparser/, November 2013. [Online; accessed Feb-2016].

[Jdb] JDBC Driver for MySQL. http://dev.mysql.com/downloads/connector/j/. [Online; accessed Feb-2016].

[KHAK09] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency Rationing in the Cloud: Pay Only when It Matters. *Proc. VLDB Endow.*, 2(1):253–264, August 2009.

[KPF+13] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-data Center Consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, New York, NY, USA, 2013. ACM.

[KR11] Deokhwan Kim and Martin C. Rinard. Verification of Semantic Commutativity Conditions and Inverse Operations on Linked Data Structures. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 528–541, New York, NY, USA, 2011. ACM.

[Kun07] Viktor Kuncak. *Modular Data Structure Verification.* PhD thesis, MIT, 2007.

[Lam78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

[Lam98] Leslie Lamport. The Part-time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[Lam05] Leslie Lamport. Generalized Consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[Lam06] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, October 2006.

[LFKA11] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don'T Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[LFKA13] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger Semantics for Low-latency Geo-replicated Storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 313–328, Berkeley, CA, USA, 2013. USENIX Association.

[Li] Harry C. Li. Practical Consistency Tradeoffs. [Invited talk at *the 31st Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (PODC'12), 2012].

[Li214] SIEVE Example Files. http://www.mpi-sws.org/~chengli/atc2014files/, 2014. [Online; accessed Feb-2016].

[Lin06] Greg Linden. Marissa Mayer at Web 2.0. http://glinden.blogspot.pt/2006/11/marissa-mayer-at-web-20.html, 2006. [Online; accessed Feb-2016].

[LKMS04] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure Untrusted Data Repository (SUNDR). In *Proceedings of the 6th Conference*

on Symposium on Opearting Systems Design & Implementation - Volume 6, OSDI'04, pages 9–9, Berkeley, CA, USA, 2004. USENIX Association.

[LLaC+14] Cheng Li, João Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the Choice of Consistency Levels in Replicated Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 281–292, Berkeley, CA, USA, 2014. USENIX Association.

[LLaC+15] Cheng Li, João Leitão, Allen Clement, Nuno Preguiça, and Rodrigo Rodrigues. Minimizing Coordination in Replicated Systems. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15, pages 8:1–8:4, New York, NY, USA, 2015. ACM.

[LLSG92] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing High Availability Using Lazy Replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, November 1992.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[LMA+14] Jed Liu, Tom Magrino, Owen Arden, Michael D. George, and Andrew C. Myers. Warranties for Faster Strong Consistency. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 503–517, Berkeley, CA, USA, 2014. USENIX Association.

[LPC+12] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference*

on Operating Systems Design and Implementation, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association.

[LPS09] Mihai Letia, Nuno M. Preguiça, and Marc Shapiro. CRDTs: Consistency without Concurrency Control. *CoRR*, abs/0907.0929, 2009. http://arxiv.org/abs/0907.0929.

[MAK13] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, New York, NY, USA, 2013. ACM.

[Mic] Microsoft US — Devices and Services. www.microsoft.com/. [Online; accessed Feb-2016].

[MS02] David Mazières and Dennis Shasha. Building Secure File Systems out of Byzantine Storage. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02, pages 108–117, New York, NY, USA, 2002. ACM.

[MSL+11] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud Storage with Minimal Trust. *ACM Trans. Comput. Syst.*, 29(4):12:1–12:38, December 2011.

[MyS] MySQL, The World's Most Popular Open Source Database. https://www.mysql.com/. [Online; accessed Feb-2016].

[Net] Netty IO. http://netty.io/. [Online; accessed Feb-2016].

[oli] Olisipo code repository. https://github.com/pandaworrior/VascoRepo. [Online; accessed Dec-2015].

[Oli15] Modified RUBiS Transition Table. http://www.mpi-sws.org/~chengli/olisipofiles/workload/vasco_transitions_3.xls, 2015. [Online; accessed Dec-2015].

[PMSL09] Nuno Preguica, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. A Commutative Replicated Data Type for Cooperative Editing. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, pages 395–403, Washington, DC, USA, 2009. IEEE Computer Society.

[PS99] Fernando Pedone and André Schiper. Generic Broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing*, DISC '99, 1999.

[PST+97] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 288–301, New York, NY, USA, 1997. ACM.

[ria] Using Data Types – Riak Documentation. http://docs.basho.com/riak/latest/dev/using/data-types/. [Online; accessed Feb-2016].

[Rit12] Antonio Rito da Silva et al. Project Fenix applications and Information Systems of Instituto Superior Técnico. https://fenix-cvs.ist.utl.pt, 2012. [Online; accessed Feb-2016].

[RKB+15] Sudip Roy, Lucja Kot, Gabriel Bender, Bailu Ding, Hossein Hojjat, Christoph Koch, Nate Foster, and Johannes Gehrke. The Homeostasis Protocol: Avoiding Transaction Coordination Through Program Analysis. In *Proceedings of the 2015 ACM SIGMOD International Conference on Man-*

agement of Data, SIGMOD '15, pages 1311–1326, New York, NY, USA, 2015. ACM.

[RKF+14] Sudip Roy, Lucja Kot, Nate Foster, Johannes Gehrke, Hossein Hojjat, and Christoph Koch. Writes that Fall in the Forest and Make no Sound: Semantics-Based Adaptive Data Consistency. *CoRR*, abs/1403.2307, 2014.

[SAB] João Sousa, Eduardo Alchieri, and Alysson Bessani. BFT-SMART Code Repository. https://github.com/bft-smart/library. [Online; accessed Dec-2015].

[SB09] Eric Schurman and Jake Brutlag. Performance Related Changes and their User Impact. http://slideplayer.com/slide/1402419/, 2009. Presented at *Velocity Web Performance and Operations Conference*. [Online; accessed Feb-2016].

[Sch90] Fred B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[SFK+09] Atul Singh, Pedro Fonseca, Petr Kuznetsov, Rodrigo Rodrigues, and Petros Maniatis. Zeno: Eventually Consistent Byzantine-fault Tolerance. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 169–184, Berkeley, CA, USA, 2009. USENIX Association.

[SIE] SIEVE Code Repository. https://github.com/pandaworrior/SIEVE. [Online; accessed Feb-2016].

[SLSV95] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. Transaction Chopping: Algorithms and Performance Studies. *ACM Trans. Database Syst.*, 20(3):325–363, September 1995.

[Sob08] Jason Sobel. Scaling Out. https://www.facebook.com/note.php?note_id=23844338919, 2008. [Online; accessed Feb-2016].

[SPAL11] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional Storage for Geo-replicated Systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[SPBZ11a] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A Comprehensive Study of Convergent and Commutative Replicated Data Types. Technical Report 7506, INRIA, January 2011.

[SPBZ11b] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. In *SSS*, 2011.

[SQL99] *SQL-99 Complete, Really.* CMP Books, 1999.

[SS05] Yasushi Saito and Marc Shapiro. Optimistic Replication. *ACM Comput. Surv.*, 37(1):42–81, March 2005.

[sta] statista. Number of worldwide active Amazon customer accounts from 1997 to 2014 (in millions). http://www.statista.com/statistics/237810/number-of-active-amazon-customer-accounts-worldwide/. [Online; accessed Feb-2016].

[Sto10] Dan Stocker. Delta Transactions. http://collectiveweb.wordpress.com/2010/03/01/delta-transactions/, 2010. [Online; accessed Feb-2016].

[Sul13] Danny Sullivan. Google Still World's Most Popular Search Engine By Far, But Share Of Unique Searchers Dips Slightly. http://searchengineland.com/google-worlds-most-popular-search-engine-148089, 2013. [Online; accessed Aug-2014].

[TDP+94] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, PDIS '94, pages 140–149, Washington, DC, USA, 1994. IEEE Computer Society.

[TPC11] TPC-W Code Repository. https://github.com/davidmartinho/fenix-framework/tree/master/examples/tpcw, 2011. [Online; accessed Feb-2016].

[TPK+13] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based Service Level Agreements for Cloud Storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 309–324, New York, NY, USA, 2013. ACM.

[TTP+95] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM.

[Vog09] Werner Vogels. Eventually Consistent. *Commun. ACM*, 52(1):40–44, January 2009.

[vRBM96] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: A Flexible Group Communication System. *Commun. ACM*, 39(4):76–83, April 1996.

[Web] Why Web Performance Matters : Is Your Site Driving Customers Away? http://www.mcrinc.com/Documents/Newsletters/201110_why_web_performance_matters.pdf. [Online; accessed Feb-2016].

[Wei88] W. E. Weihl. Commutativity-Based Concurrency Control for Abstract Data Types. *IEEE Trans. Comput.*, 1988.

[YV00] Haifeng Yu and Amin Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 21–21, Berkeley, CA, USA, 2000. USENIX Association.

[ZPZ+13] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. Transaction Chains: Achieving Serializability with Low Latency in Geo-distributed Storage Systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 276–291, New York, NY, USA, 2013. ACM.

[ZSS+15] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 263–278, New York, NY, USA, 2015. ACM.