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

Cheng Li[†], Daniel Porto[*†], Allen Clement[†], Johannes Gehrke[‡], Nuno Preguiça[*], Rodrigo Rodrigues[*]
[†]Max Planck Institute for Software Systems (MPI-SWS), [*]CITI / DI-FCT-Universidade Nova de Lisboa, [‡]Cornell University

Abstract: Online services distribute and replicate state across geographically diverse data centers and direct user requests to the closest or least loaded site. While effectively ensuring low latency responses, this approach is at odds with maintaining cross-site consistency. We make three contributions to address this tension. First, we propose RedBlue consistency, which enables blue operations to be fast (and eventually consistent) while the remaining red operations are strongly consistent (and slow). Second, to make use of fast operation whenever possible and only resort to strong consistency when needed, we identify conditions delineating when operations can be blue and must be red. Third, we introduce a method that increases the space of potential blue operations by breaking them into separate generator and shadow phases. We built a coordination infrastructure called Gemini that offers RedBlue consistency, and we report on our experience modifying the TPC-W and RUBiS benchmarks and an online social network to use Gemini. Our experimental results show that RedBlue consistency provides substantial performance gains without sacrificing consistency.

1 Introduction

Scaling services over the Internet to meet the needs of an ever-growing user base is challenging. In order to improve user-perceived latency, which directly affects the quality of the user experience [32], 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 [9], resort to weaker consistency semantics like eventual consistency where state can temporarily diverge. Others, such as Yahoo!'s PNUTS [8], avoid state divergence by requiring all operations that update the service state to be funneled through a primary site and thus incurring increased latency.

This paper addresses the inherent tension between performance and meaningful consistency. A first step towards this goal is to allow multiple levels of consistency to coexist [19, 34, 35]: 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. This 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 paper we make the following three contributions to address this tension.

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 [9, 25, 38, 26, 12, 33, 34]. Red operations, in contrast, are serialized with respect to each other and require immediate cross-site coordination. 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 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.
3. We observe that the commutativity requirement limits the space of potentially blue operations. To address this, 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. Only shadow operations are colored red or blue. This allows for a fine-grained classification of operations and broadens the space of potentially blue operations.

We built a system called Gemini that coordinates RedBlue 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.

The rest of the paper is organized as follows: we po-

Consistency level	Example systems	Immediate response	State convergence	Single value	General operations	Stable histories	Classification strategy
Strong	RSM [20, 31]	no	yes	yes	yes	yes	N/A
Timeline/snapshot	PNUTS [8], Megastore [3]	reads only	yes	yes	yes	yes	N/A
Fork	SUNDR [24]	all ops	no	yes	yes	yes	N/A
Eventual	Bayou [38], Depot [26]	all ops	yes	no	yes	yes	N/A
	Sporc [12], CRDT [33]	all ops	yes	yes	no	yes	N/A
	Zeno [34], COPS [25]	weak/all ops	yes	yes	yes	no	no / N/A
Multi	PSI [35]	cset	yes	yes	partial	yes	no
	lazy repl. [19], Horus [39]	immed./causal ops	yes	yes	yes	yes	no
RedBlue	Gemini	Blue ops	yes	yes	yes	yes	yes

Table 1: Tradeoffs in geo-replicated systems and various consistency levels.

sition our work in comparison to existing proposals in §2. We define RedBlue consistency and introduce shadow operations along with a set of principles of how to use them in §4 and §5. We describe our prototype system in §6, and report on the experience transitioning three application benchmarks to be RedBlue consistent in §7. We analyze experimental results in §8 and conclude in §9.

2 Background and related work

Target end-to-end properties. To frame the discussion of existing systems that may be used for geo-replication, we start by informally stating some desirable properties that such solutions should support. The first property consists of ensuring a good user experience by providing **low latency** access to the service [32]. Providing low latency access implies that operations should proceed after contacting a small number of replicas, but this is at odds with other requirements that are often sacrificed by consistency models that privilege low latency. The first such requirement is preserving **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 [28]. Second, it is important that all replicas that have executed the same set of operations are in the same state, i.e., that they exhibit **state convergence**. Third, we want to avoid marked deviations from the conventional, single server semantics. In particular, operations should return a **single value**, precluding solutions that return a set of values corresponding to the outcome of multiple concurrent updates; the system should provide a set of **stable histories**, meaning that user actions cannot be undone; and it should provide support for **general operations**, not restricting the type of operations that can be executed. Fourth, the behavior of the system must obey its specification. This specification may be defined as a set of **invariants** that must be preserved, e.g., that no two users receive the same user id when registering. Finally, and orthogonally to the tension between low latency and user semantics, it is important for all operations executed at one replica to be propagated to all remaining replicas, a property we call **eventual propagation**.

Table 1 summarizes several proposals of consistency definitions, which strike different balances between the requirements mentioned above. 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 [17], 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 [8] and snapshot consistency in Megastore [3]. 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 [23]. This allows for fast reads executed at the closest site but writes still pay a penalty for serialization. Fork consistency [24, 27] relaxes 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 [38] 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 [30] for a survey.) In practice, as shown in Table 1, systems that embrace eventual consistency have limitations. 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 [34], or by resorting to a last writer wins strategy, which often results in loss of one of the concurrent updates [25]. Other systems expose multiple values from divergent branches in operations replies either directly to the client [26, 9] or to an application-specific conflict resolution procedure [38]. Finally, some systems restrict operations by assuming that all operations in the system commute [12, 33], 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 [39], lazy replication [19], Zeno [34], and PSI [35]. 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 [18] 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 [41] 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 our work 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. This enables applications to make more use of fast operations. Some prior work also aims at increasing operation commutativity: Weihl exploited commutativity-based concurrency control for abstract data types [40]; operational transformation [10, 12] extends non-commutative operations with a transformation that makes them commute; Conflict-free Replicated Data Types (CRDTs) [33] design operations that commute by construction; Gray [15] 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 [36] 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 [16] and CRDT downstreams [33]. In contrast to these proposals, for each operation, we may generate different shadow operations based on the specifics of the execution. Also, shadow operations can run under different consistency levels, which is important because commutativity is not always sufficient to ensure safe weakly consistent operation.

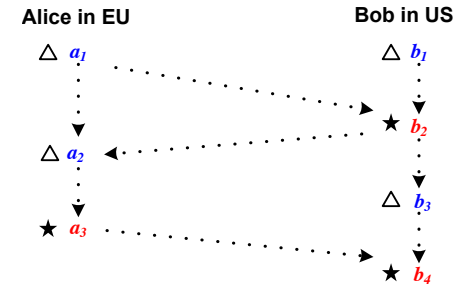
3 System model

We assume a distributed system with state fully replicated across k sites denoted $site_0 \ldots site_{k-1}$. We follow the traditional deterministic state machine model, where there is a set of possible states $\mathcal{S}$ and a set of possible operations O, each replica holds a copy of the current system state, and upon applying an operation each replica deterministically transitions to the next state and possibly outputs a corresponding reply.

In our notation, $S \in \mathcal{S}$ denotes a system state, and $u, v \in O$ denote operations. We assume there exists an initial state S_0. If operation u is applied against a system state S, it produces another system state S'; we will also denote this by $S' = S + u$. We say that a pair of operations u and v *commute* if $\forall S \in \mathcal{S}, S+u+v = S+v+u$. The system maintains a set of application-specific invariants. We say that state S is *valid* if it satisfies all these invariants. Each operation u is initially submitted at one site which we call u's *primary site* and denote $site(u)$; the system then later replicates u to the other sites.

4 RedBlue consistency

In this section we introduce RedBlue consistency, which 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. Eventual consistency [25, 38, 26, 12], 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 [17] or serializability [5] 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 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.



(a) RedBlue order O of operations

(b) Causal serializations of O

Figure 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.

4.1 Defining RedBlue consistency

The definition of RedBlue consistency has two components: (1) A RedBlue order, which defines a 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 $U = B \cup R$, where $B \cap R = \emptyset$, 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 a deterministic state machine that processes operations in a serial order. The serial order executed by site i is a causal serialization if it is compatible with the global RedBlue order and ensures causality for all operations initially executed at site i. 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 2 (Causal serialization)**.** *Given a site i, $O_i = (U, <)$ is an* i-causal serialization *(or short, a* causal serialization*) of RedBlue order $O = (U, \prec)$ if (a) O_i is a linear extension of O (i.e., $<$ is a total order compatible with the partial order $\prec$), and (b) for any two operations $u, v \in U$, if site$(v) = i$ and $u < v$ in O_i, then $u \prec v$.*

```
1  float balance, interest = 0.05;
2  func deposit( float money ):
3     balance = balance + money;
4  func withdraw ( float money ):
5     if ( balance - money >= 0 ) then:
6        balance = balance - money;
7     else print "failure";
8  func accrueinterest():
9     float delta = balance × interest;
10    balance = balance + delta;
```

Figure 2: Pseudocode for the bank example.

Definition 3 (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 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 [5]; in systems where every operation is labeled blue, RedBlue consistency allows the same set of behaviors as eventual consistency [38, 25, 26]. 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 §2, in particular, state convergence and invariant preservation, as discussed next.

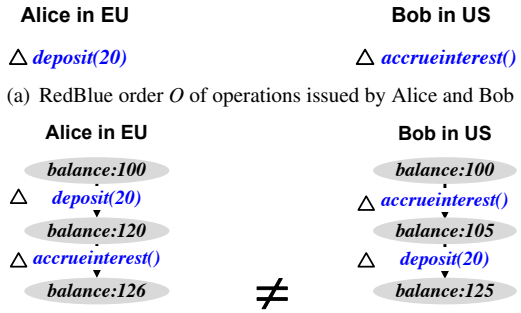
4.2 State convergence and a RedBlue bank

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 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 2). Let the `deposit` and `accrueinterest` operations be blue. Figure 3 shows a RedBlue order of deposits and interest accruals made by Alice and Bob and 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 4 (State convergence)**.** *A RedBlue consistent system is state convergent if all causal serializations of the underlying RedBlue order O reach the same state S.*

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 (`dep-`



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

(b) Causal serializations of O leading to diverged state

Figure 3: A RedBlue consistent account with initial balance of $100.

osit) 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.

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

The banking example and Theorem 1 highlight an important tension inherent to RedBlue consistency. On the one hand, low latency requires an abundance of blue operations. On the other hand, state convergence requires that blue operations commute with all other operations, blue or red. In the next section we introduce a method for addressing this tension by increasing commutativity.

5 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 §4.2. 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.

5.1 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 implementation of generator and shadow operations must obey some basic correctness requirements. 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.

Definition 5 (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, 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.

In practice, as exemplified in §7, 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. Unlike previous approaches [16, 33], 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.

5.2 Revisiting RedBlue consistency

Decomposing operations into generator and shadow components 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.

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 6 (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*

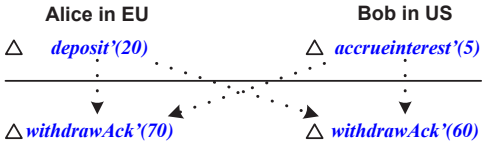
- *O_i is a total order;*
- *$(U, <)$ is a linear extension of O;*
- *For any $h_v(S) \in U$ generated by $g_v \in V_i$, S is the state obtained after applying the sequence of shadow operations preceding g_v in O_i;*
- *For any $g_v \in V_i$ and $h_u(S) \in U$, $h_u(S) < g_v$ in O_i iff $h_u(S) \prec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

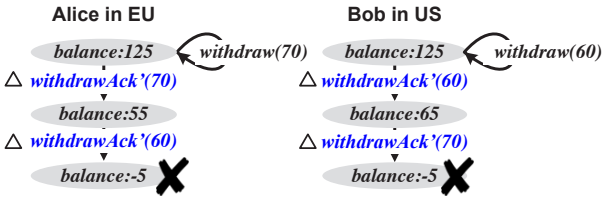
[1] All proofs can be found in a separate technical report [22].

```
1  func deposit' ( float money ):
2    balance = balance + money;
3  func withdrawAck' ( float money ):
4    balance = balance - money;
5  func withdrawFail' ():
6    /* no-op */
7  func accrueinterest' ( float delta ):
8    balance = balance + delta;
```

Figure 4: Pseudocode for shadow bank operations.



(a) RedBlue order O of banking shadow operations



(b) Invalid but convergent causal serializations of O

Figure 5: A RedBlue consistent bank with only blue operations. 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.

5.3 Shadow banking and invariants

Figure 4 shows the shadow operations for the banking example. Note that the `withdraw` operation maps to two distinct shadow operations that may be labeled as blue or red independently—`withdrawAck'` and `withdrawFail'`.

Figure 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 operations blue.

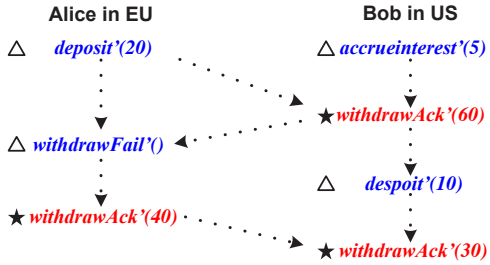
In this example, such a labeling would allow Alice and Bob to successfully withdraw $70 and $60 at their local branches, thus ending up with a final balance of $-5. This violates the fundamental invariant that a bank balance should never be negative.

To determine which 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.

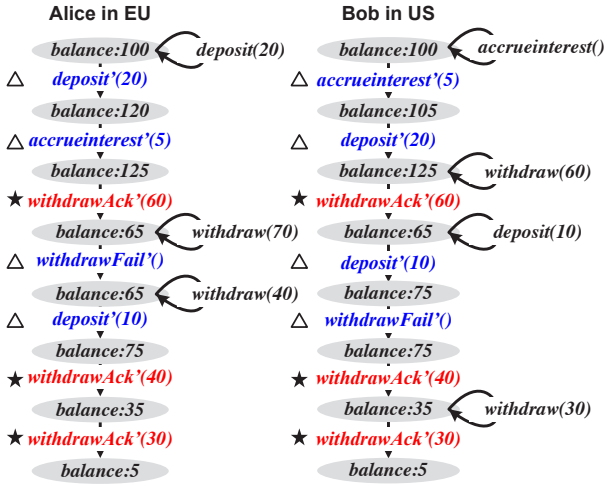
Definition 7 (Invariant safe)**.** *Shadow operation $h_u(S)$ is invariant safe if for all valid states S and S', the state $S' + h_u(S)$ is also valid.*

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

Theorem 2. *If all shadow operations are correct and all blue shadow operations are invariant safe and globally commutative, then for any execution of that system that is RedBlue consistent, no site is ever in an invalid state.*



(a) RedBlue order O of banking shadow operations



(b) Convergent and invariant preserving causal serializations of O

Figure 6: A RedBlue consistent bank with correctly labeled shadow operations and initial balance of $100.

What can be blue? What must be red? The combination of Theorems 1 and 2 leads to the following procedure 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. Figure 6 shows a RedBlue order with appropriately labeled shadow operations and causal serializations for the two sites.

5.4 Discussion

Shadow operations 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 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.

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.

6 Gemini design & implementation

We implemented the Gemini storage system to provide RedBlue consistency. The prototype consists of 10K lines of java code and uses MySQL as its storage backend. 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 site 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 an appropriate application transaction, which is implemented as a single Gemini 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 temporary table. 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 operation is accepted or rejected. Additionally, accepted shadow operations are appended to the end of the local 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.

6.1 Optimistic concurrency control

Gemini relies on optimistic concurrency control (OCC) [5] to run generator operations without blocking.

Gemini uses timestamps to determine if operations can complete successfully. Timestamps are logical clocks [20] 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 coordinator must determine if the operation (a) reads a coherent system snapshot and (b) obeys the ordering constraints of a causal serialization, as described in §5. To do this, the coordinator checks the timestamps of the data items readd and written by the completing operation, and compares them to the timestamps associated with operations completing concurrently and the remote shadow operations that were being applied simultaneously at that site.

Upon successful completion of a generator operation 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 operations 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, it is assigned a new timestamp that is the entry-wise max of the timestamp assigned to the shadow operation in the initial site and the local timestamps of accessed data objects. This captures dependencies that span local and remote operations.

Read-only shadow operations. As a performance optimization, a subset of blue shadow operations can be marked read-only. Read-only shadow operations receive special treatment from the coordinator: once the generator operation passes the coherence and causality checks, 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.

6.2 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 con-

tributions of this paper, 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 [20, 31] can be employed to make each component robust to failures.

Site failure. Our Gemini prototype relies on a simple token exchange for coordinating red epochs. To avoid halting the system upon a site failure, a fault tolerant consensus protocol like Paxos [21] 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 [26, 2, 38, 28] or reliable multicast [13].

Cross-session monotonicity. The proxy that each user connects to enforces the monotonicity of user requests within a session [37]. 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 [26] in order to replay them when connecting to a new site.

7 Case studies

In this section we report on our experience in modifying three existing applications—the TPC-W shopping cart benchmark [7], the RUBiS auction benchmark [11], and the Quoddy social networking application [14]—to work with RedBlue consistency. The two main tasks to fulfill this goal are (1) decomposing the application into generator and shadow operations and (2) labeling the shadow operations 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 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 §5.3, and the application changes measured in lines 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 applicaations RedBlue consistent.

7.1 TPC-W

TPC-W [7] 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 [29] 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 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 7(a).

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 7(b) shows the generator operation of `doBuyConfirm`, and Figures 7(c) and 7(d) 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 number of items 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 29 shadow operations in TPC-W, we find that 27 can be blue and only two must be red. To label shadow operations, we

Application	Original					RedBlue consistent extension				
	user requests	transactions			LOC	shadow operations				LOC changed
		total	read-only	update		blue no-op	blue update	red	LOC	
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 2: Original applications and the changes needed to make them RedBlue consistent.

```
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();}
```

(a) Original transaction that commits changes to database.

```
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 sp.exec(UPDATE itemTb SET item.stock+ = delta);
12     else sp.discard(); return;
13  else sp.exec(UPDATE itemTb SET item.stock− = cart.item.qty);
14  sp.exec(DELETE FROM cartTb WHERE cId=cartId AND id=cart.item.id);
15  L_TS = getCommitOrder();
16  sp.discard();
17  if replenished return (doBuyConfirmIncre′(orderId,cartId,
          cart.item.id,cart.item.qty,cost,delta,L_TS));
18  else return (doBuyConfirmDecre′(orderId,cartId,cart.item.Id,
          cart.item.qty,cost,L_TS));}
```

(b) 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);}
```

(c) 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);}
```

(d) Shadow doBuyConfirmDecre' (Red) that decrements the stock value.

Figure 7: Pseudocode for the product purchase transaction in TPC-W. For simplicity the pseudocode assumes that the corresponding shopping cart only contains a single type of item.

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 7(d)) 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 7(c)) *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 [6] and was circumvented by modifying the ID generation code, so that IDs become a pair $\langle appproxy\_id,\ seqnumber \rangle$, which makes these operations trivially blue.

7.2 RUBiS

RUBiS [11] emulates an online auction website modeled after eBay [1]. 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.

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 §7.1. 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.

7.3 Quoddy

Quoddy [14] 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.

7.4 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 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.

8 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.

8.1 Experimental setup

We run experiments on Amazon EC2 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 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.

	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: Average round trip latency and bandwidth between Amazon sites.

8.2 Microbenchmark

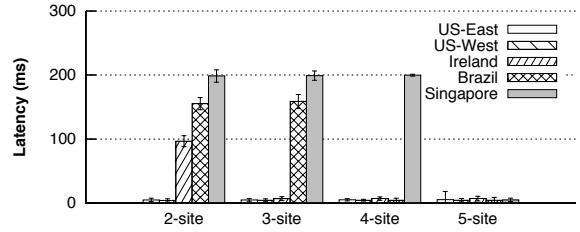
We begin the evaluation with a simple microbenchmark designed to stress the costs and benefits of partitioning operations into red and blue sets. Each user issues requests accessing a random record from a MySQL database. Each request maps to a single shadow operation; we say a request is blue if it maps 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.

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.

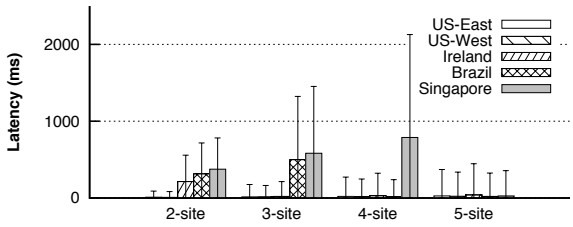
8.2.1 User observed latency

The primary benefit of using Gemini 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 latency. In the following experiments each user issues a single outstanding request at a time.

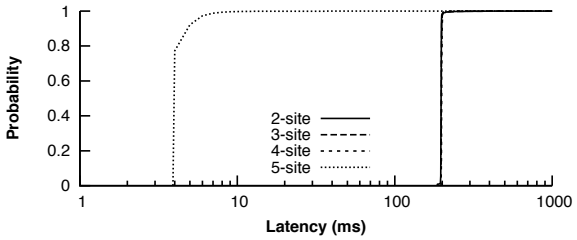
Figure 8(a) shows that the average latency for blue requests is dominated by the latency between the user and the closest site; as expected, average latency decreases as additional sites appear close to the user. Figure 8(b) shows that this trend also holds for red requests. The average latency and standard deviation, however, are higher for red requests than for blue requests. To understand this effect, we plot in Figures 8(c) and 8(d) 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. For red requests, in the $k = 2$ through $k = 4$ site configurations, four 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, since there is now



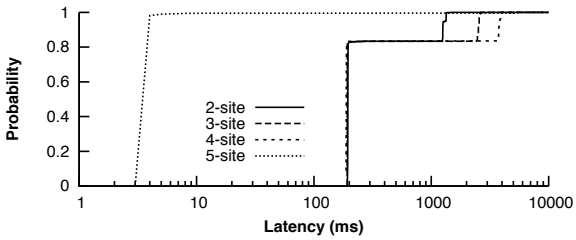
(a) Blue request latency for all users as number of sites increases



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



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



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

Figure 8: (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. (c) and (d) show the CDF of latencies for blue and red requests issued by users in Singapore as the number of sites is increased, respectively.

a much smaller fraction of requests that need to wait four seconds for the token to return.

8.2.2 Peak throughput

We now shift our attention to the throughput implications of RedBlue consistency. Figure 9 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 9 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.

8.3 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.

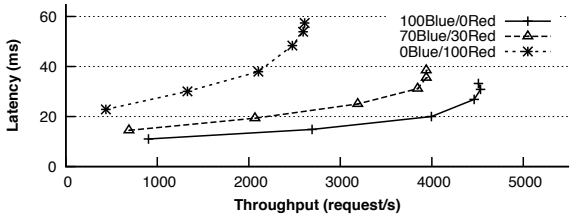


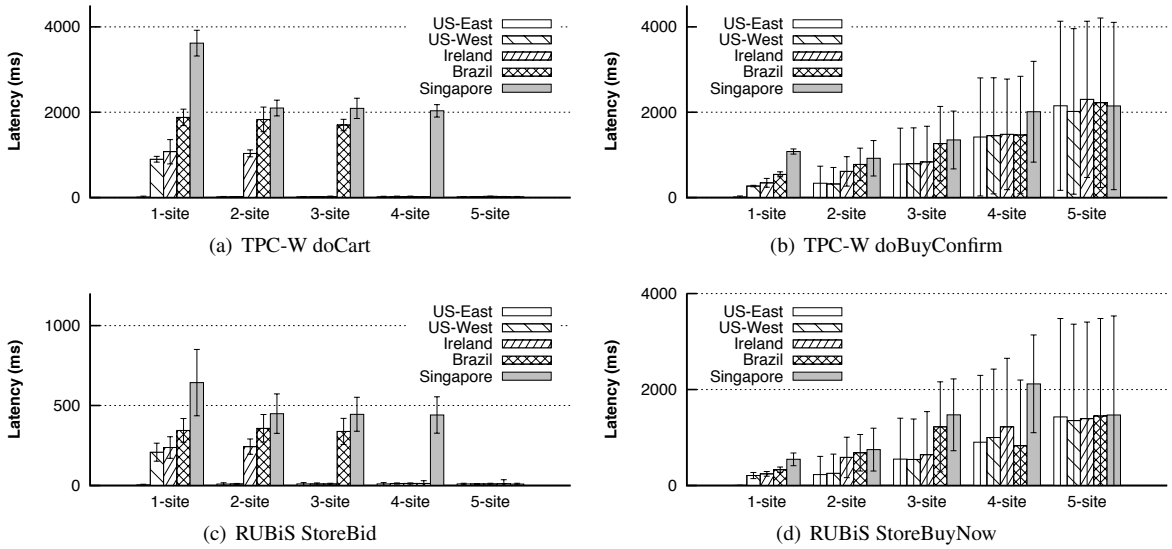
Figure 9: Throughput versus latency graph for a 2 site configuration with varying red-blue workload mixes.

8.3.1 Configuration and workloads

In all case studies 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.

TPC-W. TPC-W [7] 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.



(a) TPC-W doCart (b) TPC-W doBuyConfirm

(c) RUBiS StoreBid (d) RUBiS StoreBuyNow

Figure 10: Average latency for selected TPC-W and RUBiS user interactions. Shadow operations for doCart and StoreBid are always blue; for doBuyConfirm and StoreBuyNow they are red 98% and 99% of the time respectively.

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

8.3.2 Prevalence of blue and red shadow operations

Table 4 shows the distribution of blue and red shadow operations during 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.

8.3.3 User observed latency

We first explore the per request latency for a set of exemplar red and blue 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.

From TPC-W we select doBuyConfirm (discussed in detail in §7.1) 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.

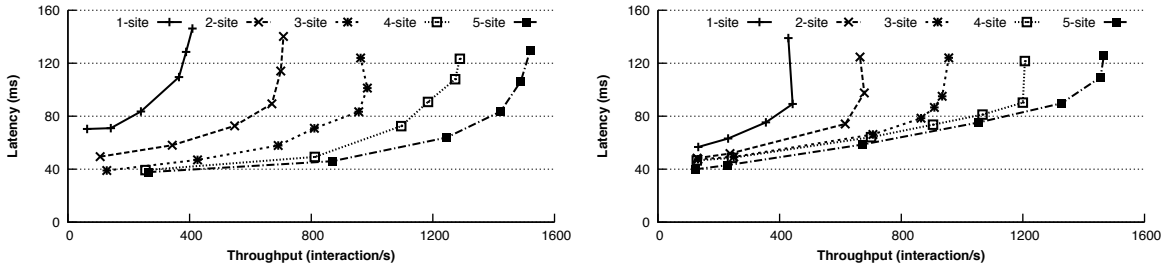
Figures 10(a) and 10(c) show that the latency trends for blue shadow operations are consistent with the results from the microbenchmark—observed latency is directly proportional to the latency to the closest site. The raw latency values are higher than the round-trip time from the user to the nearest site because processing each request involves sending one or more images to the user.

For red requests, Figures 10(b) and 10(d) show that llatency 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 token holding interval. We note that the token passing scheme used by Gemini is simple and additional work is needed to identify an optimal strategy for regulating red shadow operations.

8.3.4 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 11 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



(a) TPC-W shopping mix

(b) RUBiS bidding mix

Figure 11: 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.

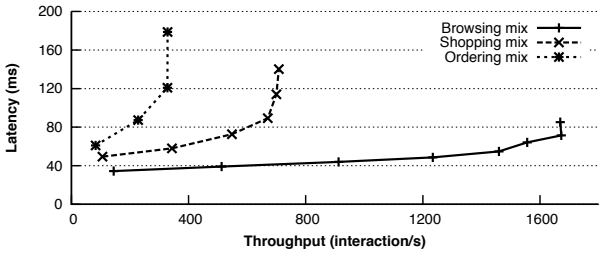


Figure 12: TPC-W: Throughput vs. latency graph for TPC-W with Gemini spanning two sites when running the three workload mixes.

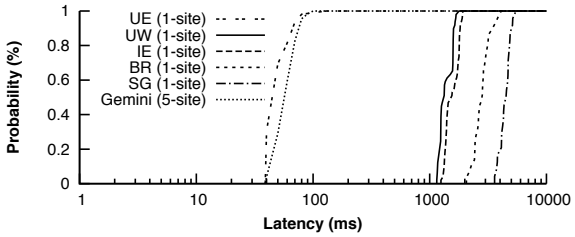


Figure 13: User latencies CDF for the addFriend request in single site Quoddy and 5-site Gemini deployments.

	TPC-W shopping		RUBiS biding	
	Original	Gemini	Original	Gemini
Thput. (inter/s)	409	386	450	370
Avg. latency	14 ms	15 ms	6 ms	7 ms

Table 5: Performance comparison between the original code and the Gemini version for both TPC-W and RUBiS within a single site.

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

8.4 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. [4]. 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 13 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. The significantly higher latencies for remote regions are associated with the images and javascripts that Quoddy distributes as part of processing the addFriend request.

8.5 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 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.

9 Conclusion

In this paper, we presented a principled approach to building geo-replicated systems that are fast as possible

and consistent when needed. Our approach is based on our novel notion of RedBlue consistency allowing both strongly consistent (red) operations and eventually consistent (blue) operations to coexist, a concept of shadow operation enabling the maximum usage of blue operations, and a labeling methodology for precisely determining which operations to be assigned which consistency level. Experimental results from running benchmarks with our system Gemini show that RedBlue consistency significantly improves the performance of geo-replicated systems.

Acknowledgments

We sincerely thank Edmund Wong, Rose Hoberman, Lorenzo Alvisi, our shepherd Jinyang Li, and the anonymous reviewers for their insightful comments and suggestions. The research of R. Rodrigues has received funding from the European Research Council under an ERC starting grant. J. Gehrke was supported by the National Science Foundation under Grant IIS-1012593, the iAd Project funded by the Research Council of Norway, a gift from amazon.com, and a Humboldt Research Award from the Alexander von Humboldt Foundation. N. Preguiça is supported by FCT/MCT projects PEst-OE/EEI/UI0527/2011 and PTDC/EIA-EIA/108963/2008.

References

[1] Ebay website. http://www.ebay.com/, 2012.

[2] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. Hutto. Causal memory: Definitions, implementation and programming. Technical report, Georgia Institute of Technology, 1994.

[3] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.

[4] F. Benevenuto, T. Rodrigues, M. Cha, and V. Almeida. Characterizing user behavior in online social networks. In *IMC*, 2009.

[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. 1987.

[6] E. Cecchet, G. Candea, and A. Ailamaki. Middleware-based database replication: the gaps between theory and practice. In *SIGMOD*, 2008.

[7] T. consortium. Tpc benchmark-w specification v. 1.8. http://www.tpc.org/tpcw/spec/tpcw_v1.8.pdf, 2002.

[8] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. In *VLDB*, 2008.

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

[10] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *SIGMOD*, 1989.

[11] C. Emmanuel and M. Julie. Rubis: Rice university bidding system. http://rubis.ow2.org/, 2009.

[12] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. Sporc: group collaboration using untrusted cloud resources. In *OSDI*, 2010.

[13] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Trans. Netw.*, 1997.

[14] Fogbeam Labs. Quoddy code repository, 2012. http://code.google.com/p/quoddy/.

[15] J. Gray. The transaction concept: Virtues and limitations. In *VLDB*, 1981.

[16] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD*, 1996.

[17] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 1990.

[18] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency rationing in the cloud: pay only when it matters. In *VLDB*, 2009.

[19] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 1992.

[20] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 1978.

[21] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 1998.

[22] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. Technical report, MPI-SWS. http://www.mpi-sws.org/chengli/rbTR.pdf, 2012.

[23] H. Li. Practical consistency tradeoffs. In *PODC*, 2012.

[24] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (sundr). In *OSDI*, 2004.

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

[26] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: cloud storage with minimal trust. In *OSDI*, 2010.

[27] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *PODC*, 2002.

[28] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *SOSP*, 1997.

[29] A. Rito da Silva et al. Project fenix applications and information systems of instituto superior tcnico. https://fenix-cvs.ist.utl.pt, 2012.

[30] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 2005.

[31] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 1990.

[32] E. Schurman and J. Brutlag. Performance relatedchanges and their user impact. Presented at velocity web performance and operations conference, 2009.

[33] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *SSS*, 2011.

[34] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: eventually consistent byzantine-fault tolerance. In *NSDI*, 2009.

[35] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, 2011.

[36] D. Stocker. Delta transactions. http://collectiveweb.wordpress.com/2010/03/01/delta-transactions/, 2010.

[37] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. Session guarantees for weakly consistent replicated data. In *PDIS*, 1994.

[38] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, 1995.

[39] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Commun. ACM*, 1996.

[40] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Comput.*, 1988.

[41] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *OSDI*, 2000.