Regular Sequential Serializability and Regular Sequential Consistency

Jeffrey Helt
Princeton University
United States
jhelt@cs.princeton.edu

Matthew Burke
Cornell University
United States
matthelb@cs.cornell.edu

Amit Levy
Princeton University
United States
aalevy@cs.princeton.edu

Wyatt Lloyd
Princeton University
United States
wlloyd@princeton.edu

Abstract

Strictly serializable (linearizable) services appear to execute transactions (operations) sequentially, in an order consistent with real time. This restricts a transaction's (operation's) possible return values and in turn, simplifies application programming. In exchange, strictly serializable (linearizable) services perform worse than those with weaker consistency. But switching to such services can break applications.

This work introduces two new consistency models to ease this trade-off: regular sequential serializability (RSS) and regular sequential consistency (RSC). They are just as strong for applications: we prove any application invariant that holds when using a strictly serializable (linearizable) service also holds when using an RSS (RSC) service. Yet they relax the constraints on services—they allow new, better-performing designs. To demonstrate this, we design, implement, and evaluate variants of two systems, Spanner and Gryff, relaxing their consistency to RSS and RSC, respectively. The new variants achieve better read-only transaction and read tail latency than their counterparts.

*CCS Concepts: • **Information systems → Parallel and distributed DBMSs**; **Distributed database transactions**.

*Keywords: distributed systems, consistency, databases

ACM Reference Format:
Jeffrey Helt, Matthew Burke, Amit Levy, and Wyatt Lloyd. 2021. Regular Sequential Serializability and Regular Sequential Consistency. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), October 26–29, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3477132.3483566

1 Introduction

Strict serializability [74] and linearizability [36] are exemplary consistency models. Strictly serializable (linearizable) services appear to execute transactions (operations) sequentially, in an order consistent with real time. They simplify building correct applications atop them by reducing the number of possible values services may return to application processes. This, in turn, makes it easier for programmers to enforce necessary application invariants.

In exchange for their strong guarantees, strictly serializable and linearizable services incur worse performance than those with weaker consistency [10, 25, 39, 53, 54]. For example, consider a read in a key-value store that returns the value written by a concurrent write. If the key-value store is weakly consistent, the read imposes no constraints on future reads. But if the key-value store is strictly serializable, the read imposes a global constraint on future reads—they all must return the new value, even if the write has not yet finished. Existing strictly serializable services guarantee this by blocking reads [22], incurring multiple round trips between clients and shards [94, 95], or aborting conflicting writes [94, 95]. These harm service performance, either by increasing abort rates or increasing latency.

Services with weaker consistency models [5, 7, 44, 54], however, offer application programmers with a harsh tradeoff. In exchange for better performance, they may break the invariants of applications built atop them.

This work introduces two new consistency models to ease this trade-off: regular sequential serializability (RSS) and regular sequential consistency (RSC). They allow services to achieve better performance while being *invariant-equivalent* to strict serializability and linearizability, respectively. For any application that does not require synchronized clocks, any invariant that holds while interacting with a set of strictly serializable (linearizable) services also holds when executing atop a set of RSS (RSC) services.

To maintain application invariants, a set of RSS (RSC) services must appear to execute transactions (operations) sequentially, in an order that is consistent with a broad set of causal constraints (e.g., through message passing). We prove formally this is sufficient for RSS (RSC) to be invariant-equivalent to strict serializability (linearizability).

To allow for better performance, RSS and RSC relax some of strict serializability and linearizability's real-time guarantees for causally unrelated transactions or operations, respectively. For example, when a read returns the value written by a concurrent write, instead of a global constraint, RSS

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SOSP '21, October 26–29, 2021, Virtual Event, Germany
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8709-5/21/10.
https://doi.org/10.1145/3477132.3483566

imposes a causal constraint—only reads that causally follow the first must return the new value.

But in addition to helping enforce invariants, strict serializability's (linearizability's) real-time guarantees help applications match their users' expectations. For instance, from interacting with applications on their local machine, users expect writes to be immediately visible to all future reads. Applications built atop weakly consistent services can violate these expectations, exposing anomalies.

Because RSS (RSC) relax some of strict serializability's (linearizability's) real-time constraints, applications built atop an RSS (RSC) service may expose more anomalies. But prior work suggests anomalies are rare in practice [57], and further, RSS and RSC include some real-time guarantees to make the chance of observing these new anomalies small. They should only be possible within short time windows (a few seconds). Thus, we expect the difference between RSS (RSC) and strict serializability (linearizability) to go unnoticed in practice.

To compose a set of RSS (RSC) services such that they appear to execute transactions (operations) in some global RSS (RSC) order, each must implement one other mechanism: a real-time fence. We show how the necessary fences can be invoked without changing applications.

Finally, to demonstrate the performance benefits permitted by RSS and RSC, we design, implement, and evaluate variants of two existing services: Spanner [22], Google's globally distributed database, and Gryff [18], a replicated key-value store. The variants implement RSS and RSC instead of strict serializability and linearizability, respectively.

Spanner-RSS improves read-only transaction latency by reducing the chances they must block for conflicting read-write transactions. Instead, Spanner-RSS allows read-only transactions to immediately return old values in some cases. As a result, in low- and moderate-contention workloads, Spanner-RSS reduces read-only transaction tail latency by up to 49% without affecting read-write transaction latency.

Gryff-RSC improves read latency with a different approach. By removing the write-back phase of reads, Gryff-RSC halves the number of round trips required between application processes and Gryff's replicas. As a result, for moderate- and high-contention workloads, Gryff-RSC reduces p99 read latency by about 40%. Further, because Gryff-RSC's reads always finish in one round, it offers larger reductions in latency (up to 50%) farther out on the tail.

In sum, this paper makes the following contributions:

- We define RSS and RSC, the first invariant-equivalent consistency models to strict serializability and linearizability.
- We prove that for any application not requiring synchronized clocks, any invariant that holds with strictly serializable (linearizable) services also holds with RSS (RSC).
- We design, implement, and evaluate Spanner-RSS and Gryff-RSC, which significantly improve read tail latency compared to their counterparts.

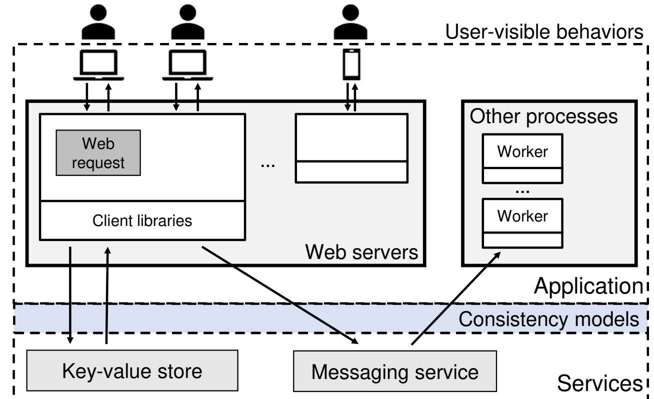


Figure 1. An application deployed in a data center. It comprises the processes running on user devices, Web servers, and asynchronous workers. They are supported by a pair of services. The services' consistency models significantly impact application correctness and performance.

2 Background and Motivation

In this section, we first describe the typical structure of applications and their interaction with supporting services. We then discuss the role consistency models play in these interactions. Finally, we demonstrate how existing consistency models offer difficult trade-offs to application programmers and service designers.

2.1 Distributed Applications

Distributed applications can be split into two parts: a set of processes executing application-specific logic and a set of services supporting them. The application-specific processes include those that respond interactively to users, such as those executing on a user's device and those they cooperate with synchronously or asynchronously, such as Web servers running in a nearby data center. The services provide generic, reusable functionality, such as data storage [16, 22, 53, 54] and messaging [82].

For example, consider a Web application deployed in a data center (Figure 1). A user interacts with a browser on their device. These interactions define the behaviors of the application. Under the hood, the browser sends HTTP requests to a set of Web servers. In processing a request, a server reads and modifies state in a key-value store and renders responses. There are also worker processes that these servers invoke asynchronously to perform longer running tasks [38]. The set of processes running application-specific logic are the *clients* of the services. The services are responsible for persisting application state, replicating it across data centers, and coordinating between application components.

2.2 Motivating Example: Photo-Sharing App

Throughout the paper, we consider a simple but illustrative example: a photo-sharing application. The application allows

	Invariants			Possible Anomalies				Performance
Consistency	$\mathcal{I}_1$	$\mathcal{I}_2$	$\mathcal{I}_{SS}$	$\mathcal{A}_1$	$\mathcal{A}_2$	$\mathcal{A}_3$	$\mathcal{A}_4$	**Latency**
Strict Serializability [74]	✓	✓	✓	never	never	never	always	↑↑
Regular Sequential Serializability	✓	✓	✓	never	never	temporarily	always	↑
Process-ordered Serializability [23, 55]	✓	✗	✗	never	always	always	always	–

$\mathcal{I}_1$: $\forall P, \forall i : i \in P.Album.list \implies P.Album.photos[i].data \neq null$
$\mathcal{I}_2$: $\forall W, \forall i : \text{Head}(W.PhotoQ) = i \land W.Photo.id = i \implies W.Photo.data \neq null$
$\mathcal{I}_{SS}$: Any invariant that holds with strict serializability.

$\mathcal{A}_1$: Alice adds two photos; later, only one photo is in her album.
$\mathcal{A}_2$: Alice adds a photo and calls Bob; Bob does not see the photo.
$\mathcal{A}_3$: Alice sees Charlie's photo and calls Bob; Bob does not see the photo.
$\mathcal{A}_4$: Alice tries to add a photo but never receives a response.

Table 1. Comparing consistency models by which invariants hold, which anomalies are prevented, and the latency of operations. $\mathcal{I}_1$ states an album never contains a photo with null data; $\mathcal{I}_2$ states a worker never reads a photo with null data.

users to backup and share their photos with other users while handling compression and other photo-processing functions.

In our example application, photos are organized into albums. Both photos and albums are stored in a globally distributed, transactional key-value store. Each photo and album has a unique key. A photo's key maps to a binary blob while an album's key maps to structured data containing the keys of all photos in the album. If a key is read but is not present, the key-value store simply returns `null`.

In addition to the transactional key-value store, the application uses a messaging service to enqueue requests for asynchronous processing. For example, when a user adds a high-resolution photo, the application enqueues the photo's key in the messaging service, requesting some worker process create lower-resolution thumbnails of the image.

When a user adds a new photo to an album, a Web server issues a read-write transaction: it creates a new key-value mapping for the photo, reads the album, and writes back the album after modifying its value to include a reference to the newly added photo. Then it enqueues a request for additional processing.

2.3 Consistency Models

The correctness and performance of an application are heavily influenced by the consistency models of its supporting services. A *consistency model* is a contract between a service and its clients regarding the allowable return values for a given set of operations. Services with stronger consistency are generally restricted to fewer possible return values, so it is easier for programmers to build a correct application atop them. Restricting the allowable return values, however, often incurs worse performance in these services and consequently, in the application.

Invariants and anomalies. The stronger restrictions of stronger consistency models enable applications to more easily ensure correctness and provide better semantics to users. The correctness of an application is determined by its *invariants*, which are logical predicates that hold for all states of an application (i.e., the combined states of all application processes). The semantics for users are determined by the rate of *anomalies*, which are behaviors the user would not observe while accessing a single-threaded, monolithic application running on a local machine with no failures. Table 1 shows some invariants and anomalies for our application.

Application logic relies on invariants to function correctly. For the photo-sharing example, client-side application logic assumes that if an album contains a reference to a photo, the photo exists in the key-value store ($\mathcal{I}_1$). Similarly, workers that receive a photo's key through the messaging service assume fetching the key from the key-value store will not return `null` ($\mathcal{I}_2$).

Applications also attempt to present reasonable behaviors to users, which is quantified by the rate of anomalies. Unlike an invariant violation, the detection of an anomaly may require information that is beyond the application's state. For example, once Alice adds a new photo to an album, Bob not seeing it is an anomaly ($\mathcal{A}_2$). But detecting $\mathcal{A}_2$ requires the application either to have synchronized clocks to record the start and end times of Alice and Bob's requests or to somehow know that Alice communicated with Bob.

2.4 Strict Serializability is Too Strong

Strict serializability [74] is one of the strongest consistency models. A service that guarantees *strict serializability* appears to execute transactions sequentially, one at a time, in an order consistent with the transactions' real-time order. As a result, only transactions that are concurrent (i.e., both begin before either ends) may be reordered. Further, strict serializability is *composable*: clients may use multiple services, and

the resulting execution will always be strictly serializable because real-time order is universal to all services [36].[1]

Strict serializability ensures a large set of invariants hold. For example, it ensures both invariants hold for our photo-sharing application. For $\mathcal{I}_1$, the application logic writes a photo's data and adds it to an album in a single transaction T. Strict serializability then trivially ensures $\mathcal{I}_1$; because transactions appear to execute sequentially, any other transaction will be either before T and not see the photo or after T and see both the photo in the album and its data. For $\mathcal{I}_2$, the application logic first executes the add-new-photo transaction and then enqueues a request to process it in the messaging service. The real-time order and composability of strict serializability then ensures $\mathcal{I}_2$; the enqueue begins in real time after the add-new-photo transaction ends, and thus, any process that sees the enqueued request must subsequently see the writes of the add-new-photo transaction.

Strict serializability also mitigates anomalies. As shown in Table 1, $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$ never occur with a strictly serializable key-value store. Because strictly serializable transactions appear to execute sequentially, no writes are lost, and its real-time guarantees ensure Bob's transactions always follow Alice's after receiving her call.

Yet applications built atop strictly serializability services are not perfect. For instance, asynchronous networks, transient failures, and a lack of fate sharing among components can all cause anomalies that are beyond the scope of a consistency model. $\mathcal{A}_4$ in Table 1 shows one example, which could not occur on a local machine with no failures.

Strict serializability imposes performance costs. In exchange for ensuring invariants and preventing most anomalies, strict serializability imposes significant performance costs on services. For instance, consider anomaly $\mathcal{A}_3$ in Table 1, and assume Charlie is in the middle of adding a photo when Alice sees it. Strict serializability mandates that any subsequent read by any application server includes the photo, even if Bob is on a different continent and Charlie's transaction has not finished. As a result, the key-value store must ensure Alice's transaction only includes Charlie's photo once all subsequent reads will, too.

Existing services provide this guarantee through a variety of mechanisms. Some block read-only transactions during conflicting read-write transactions [22]. Others incur multiple round trips between an application server and a set of replicas [94, 95] or abort concurrent read-write transactions [94, 95]. These mechanisms reduce performance by increasing either read-only transaction latency or abort rates.

2.5 Process-Ordered Serializability is Too Weak

Because strict serializability incurs heavy performance overhead, many services provide a weaker consistency model. The next strongest is *process-ordered (PO) serializability*, which guarantees that services appear to execute transactions sequentially, in an order consistent with each client's process order [23, 55]. PO serializability is weaker than strict serializability because it does not guarantee that non-concurrent transactions respect their real-time order. Moreover, PO serializability is not composable. Thus, process orders across services can be lost.

Because PO serializability is weaker than strict serializability, it avoids some of its performance costs. For instance, there are read-only transaction protocols that can always complete in one round of non-blocking requests with constant metadata in services with PO serializability, while this is impossible with strict serializability [56].

PO serializability provides fewer invariants. In exchange for better performance, weaker consistency models, like PO serializability, present application programmers with a harsh trade-off: fewer invariants will hold with reasonable application logic.[2] For our photo-sharing example, $\mathcal{I}_1$ holds because like with strict serializability, PO-serializable services appear to execute transactions sequentially.

On the other hand, $\mathcal{I}_2$ does not hold because PO serializability is not composable. A worker seeing a photo in the message queue does not ensure its subsequent reads to the key-value store will include the writes of a preceding add-new-photo transaction because the message queue and key-value store are distinct services.

2.6 Non-Transactional Consistency Models

Our discussion above focuses on transactional consistency models. The same tension between application invariants and service performance exists for the equivalent non-transactional models. Linearizability [36] and sequential consistency [44] are the non-transactional equivalents of strict serializability and process-ordered serializability, respectively. If we temporarily ignore albums and assume application processes issue a single write to add a photo, invariant $\mathcal{I}_2$ holds with a linearizable key-value store but not with a sequentially consistent one. But linearizable services must employ mechanisms that hurt their performance to satisfy linearizability's constraints, for example, by requiring additional rounds of communication for reads (§7).

3 Regular Sequential Consistency Models

A consistency model's guarantees affect both application programmers and users. Stronger models place less burden on programmers (by guaranteeing more invariants) and users (by exposing fewer anomalies) but constrain service performance. In this work, we propose two new consistency

[1]This holds only with the reasonable assumption that individual transactions do not span multiple services. This makes each strictly serializable service equivalent to a linearizable "object" [36].

[2]We say "reasonable application logic" because one can always write a middleware layer that implements a stronger consistency model, X, atop a weaker one, Y, e.g., by taking the ideas of bolt-on consistency [11] to an extreme. But the resulting X middleware on Y service is simply an inefficient implementation of an X service.

models, regular sequential serializability (RSS) and regular sequential consistency (RSC), to diminish this trade-off.

RSS and RSC are invariant-equivalent to strict serializability and linearizability, respectively. Thus, they place no additional burden on application programmers.

While they do allow more anomalies, prior work suggests anomalies with much weaker models (e.g., eventual consistency) are rare in practice (e.g., at most six anomalies per million operations [57]). Thus, we expect the additional burden on users to be negligible.

In this section, we define RSS and RSC and prove their invariant-equivalence to strict serializability and linearizability. We first describe our formal model of distributed applications (§3.1) and the services they use (§3.2). We then define RSS and RSC (§3.3 and §3.4) and finally prove our main result (§3.5). (We demonstrate RSS and RSC allow for services with better performance in later sections.)

3.1 Applications and Executions

We model a distributed *application* as a collection of *n processes*. Processes are state machines [59, 60] that implement application logic by performing local computation, exchanging messages, and invoking operations on services.

An application's processes define a prefix-closed set of *executions*, which are sequences $s_0, \pi_1, s_1, \ldots$ of alternating *states* and *actions*, starting and ending with a state. An application state contains the state of each process—it is an n-length vector of process states. As part of a process's state, we assume it has access to a local clock, which it can use to set local timers, but the clock makes no guarantees about its drift or skew relative to those at other processes.

Each action is a step taken by exactly one process and is one of three types: *internal*, *input*, or *output*. Internal actions model local computation. Processes use input and output actions to interact with other processes (e.g., receiving and replying to a remote procedure call) and their environment (e.g., responding to a user gesture). As we will describe in the following section, a subset of the input and output actions are *invocations* and *responses*, respectively, which are used to interact with services.

Processes can also exchange messages with one another via unidirectional channels. To send a message to process P_j, P_i uses two actions: first, P_i uses an output action $sendto_{ij}(m)$ and later, an input action $sent_{ij}$ occurs, indicating m's transmission on the network. Similarly, to receive a message from P_i, P_j first uses an output action $recvfrom_{ij}$ and later, an input action $received_{ij}(m)$ occurs, indicating the receipt of m.

Given an execution α, we will often refer to an individual process's *sub-execution*, denoted $\alpha|P_i$. $\alpha|P_i$ comprises only P_i's actions and the ith component of each state in α.

Well-formed. An execution is *well-formed* if it satisfies the following: (1) Messages are sent before they are received; (2) A process has at most one (total) outstanding invocation (at a service) or $recvfrom_{ij}$ (at a channel); and (3) Processes do not take output steps while waiting for a response from a service. We henceforth only consider well-formed executions.

Equivalence. Two executions α and β are *equivalent* if for all P_i, $\alpha|P_i = \beta|P_i$. Intuitively, equivalent executions are indistinguishable to the processes.

3.2 Services

Databases, message queues, and other back-end *services* that application processes interact with are defined by their *operations* and a *specification* [36, 59]. An *operation* comprises pairs of *invocations*, specifying the operations and their arguments, and matching *responses*, containing return values. The specification is a prefix-closed set of sequences of invocation-response pairs defining the service's correct behavior in the absence of concurrency. A sequence S in specification $\mathfrak{S}$ defines a total order over its operations, denoted $<_S$.

Several services can be composed into a composite service by combining their specifications as the set of all interleavings of the original services' specifications. Notably, this means a service composed of constituent services that support transactions include those transactional operations but *does not* support transactions across its constituent services. In the results below, we assume the processes interact with an arbitrary (possibly composite) service.

3.3 Consistency Models

A *consistency model* specifies the possible responses a service may return in the face of concurrent operations. Before we define our new consistency models, we must define four preliminaries. For ease of presentation, two of our definitions, conflicts and reads-from, assume a key-value store interface. While these definitions could be made general, we leave precisely defining them for other interfaces to future work.

Complete operations. Given an execution α, we say an operation is *complete* if its invocation has a matching response in α. We denote $complete(\alpha)$ as the maximal subsequence of α comprising only complete operations [36].

Conflicting operations. Given read-write transaction W, we say a read-only transaction R *conflicts* with W if W writes a key that R reads. Given an execution α, we denote the set of read-only transactions in α that conflict with W as $C_\alpha(W)$. We define conflicts and $C_\alpha(w)$ analogously for non-transactional reads and writes.

Real-time order. Two actions in an execution α are ordered in real time [36, 74], denoted $\pi_1 \longrightarrow_\alpha \pi_2$, if and only if π_1 is a response, π_2 is an invocation, and π_1 precedes π_2 in α.

Causal order. Two actions are causally related [5, 7, 43, 44, 53, 54] in an execution α, denoted $\pi_1 \rightsquigarrow_\alpha \pi_2$ if any of the following hold: (1) *Process order:* π_1 precedes π_2 in a process's sub-execution; (2) *Message passing:* π_1 is a $sendto_{ij}(m)$ and π_2 is its corresponding $received_{ij}(m)$; (3) *Reads from:* π_1 is operation o_1's response, π_2 is o_2's invocation, and o_2 reads

a value written by o_1; or (4) *Transitivity:* there exists some action π_3 such that $\pi_1 \rightsquigarrow_\alpha \pi_3$ and $\pi_3 \rightsquigarrow_\alpha \pi_2$.

3.4 RSS and RSC

We now define our new consistency models, regular sequential serializability and regular sequential consistency. Their definitions are nearly identical but because supporting transactions has significant practical implications, we distinguish between the transactional and non-transactional versions.

Intuitively, RSS (RSC) guarantees a total order of transactions (operations) such that they respect causality. Further, like prior "regular" models [45, 80, 90], reads must return a value at least as recent as the most recently completed, conflicting write.

Regular Sequential Serializability. Let $\mathcal{T}$ be the set of all transactions and $\mathcal{W} \subseteq \mathcal{T}$ be the set of read-write transactions. An execution α_1 satisfies *RSS* if it can be extended to α_2 by adding zero or more responses such that there exists a sequence $S \in \mathfrak{S}$ where (1) S is equivalent to $complete(\alpha_2)$; (2) for all pairs of transactions $T_1, T_2 \in \mathcal{T}$, $T_1 \rightsquigarrow_{\alpha_1} T_2 \implies T_1 <_S T_2$; and (3) for all read-write transactions $W \in \mathcal{W}$ and transactions $T \in C_{\alpha_1}(W) \cup \mathcal{W}$, $W \rightarrow_{\alpha_1} T \implies W <_S T$.

Regular Sequential Consistency. Let O be the set of all operations and $\mathcal{W} \subseteq O$ be the set of writes. An execution α_1 satisfies *RSC* if it can be extended to α_2 by adding zero or more responses such that there exists a sequence $S \in \mathfrak{S}$ where (1) S is equivalent to $complete(\alpha_2)$; (2) for all pairs of operations $o_1, o_2 \in O$, $o_1 \rightsquigarrow_{\alpha_1} o_2 \implies o_1 <_S o_2$; and (3) for all writes $w \in \mathcal{W}$ and operations $o \in C_{\alpha_1}(w) \cup \mathcal{W}$, $w \rightarrow_{\alpha_1} o \implies w <_S o$.

3.5 RSS and RSC Maintain Application Invariants

This section presents a condensed version of the proof. For brevity, we assume here processes do not fail and all operations finish. The full proof is in our technical report [35].

Preliminaries. The results below reason about an application's invariants, which are assertions about its states. Formally, we say a state is *reachable* in application A if it is the final state of some execution of A. An *invariant* $\mathcal{I}_A$ is a predicate that is true for all of A's reachable states [59].

In the proofs below, it will be convenient to focus on the actions within an execution. Given an execution α, its *schedule*, $sched(\alpha)$, is the subsequence of just its actions.

Proof intuition. Our main results follows from two observations. First, Lemma 1 shows we can transform an execution in which the operations respect RSS into an execution in which they respect strict serializability without reordering any actions at any of the processes. Figure 2 shows an example. The key insight is that both RSS and strict serializability guarantee equivalence to a sequence in the service's specification, which by definition is strictly serializable. Second, Theorem 2 shows that the final states of the two executions

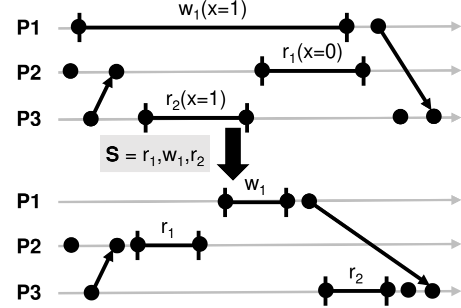


Figure 2. Example transformation from an RSS execution to a strictly serializable one. Lemma 1 proves such a transformation is possible with any RSS execution.

related by Lemma 1 are identical. It follows that invariants that hold in the first execution also hold in the second.

Lemma 1. *Suppose α is an execution of application A that satisfies RSS. Then there is an equivalent execution β of A that satisfies strict serializability.*

Proof. The proof proceeds in two steps. First, we construct a schedule β' from α's schedule α' without reordering any actions at any of the processes. Second, we construct the execution β from β' by inserting the states.

Step 1. Since α satisfies RSS, there exists a sequence $S \in \mathfrak{S}$ such that $<_S$ respects $\rightsquigarrow_\alpha$ and thus $\rightsquigarrow_{\alpha'}$. To get β', we reorder α' such that each action is ordered after the maximal (as defined by $<_S$) invocation or response action that causally precedes it. To do so, we define three relations.

First, let $\pi_1 \prec \pi_2$ if there is some invocation or response π_3 that causally precedes π_2 and that is strictly greater (by $<_S$) than all invocations and responses that causally precede π_1. Second, let $\pi_1 \equiv \pi_2$ if $\pi_1 \nprec \pi_2$ and $\pi_2 \nprec \pi_1$. Third, let $<_{\alpha'}$ be the total order of actions defined by α'. Then β' is the schedule found by ordering the actions such that $\pi_1 <_{\beta'} \pi_2$ if and only if $\pi_1 \prec \pi_2$ or $\pi_1 \equiv \pi_2$ and $\pi_1 <_{\alpha'} \pi_2$.

We show $\alpha'|P_i = \beta'|P_i$ for all P_i by contradiction, so assume some pair of actions π_1, π_2 from the same P_i were reordered in β'. Without loss of generality, assume π_2 is ordered before π_1 in α' but the reverse is true in β'. It is clear that $\pi_1 \not\equiv \pi_2$ because otherwise π_1 and π_2 would be ordered identically in α' and β'. Thus, it must be that $\pi_1 \prec \pi_2$.

Since $\pi_1 \prec \pi_2$, there must be some invocation or response π_3 that causally precedes π_2 and is greater than those that causally precede π_1. But since π_1 and π_2 are from the same process and $\pi_2 <_{\alpha'} \pi_1$ by assumption, $\pi_3 \rightsquigarrow_{\alpha'} \pi_1$ by the transitivity of $\rightsquigarrow_{\alpha'}$, contradicting the strictness in the definition of π_3. Thus, α' must be equivalent to β'.

Step 2. To get the execution β from β', we must define the processes' states. Since the order of each process's actions is the same in α' and β', each process will proceed through the same sequence of states. Thus, we can construct β's states from the sequences of each process's states in α.

To conclude, we show that β satisfies the stated properties. Since $\alpha'|P_i = \beta'|P_i$ for all P_i, it is clear that α is equivalent to β. Further, because we only reordered the states and actions in α to get β, β is clearly finite, and because $\rightsquigarrow_\alpha$ captures the sending and receiving of messages, β is well-formed. Finally, since S is a sequence of matching invocation-response pairs, the processes' interactions with the service in β are sequential, not overlapping in real time. Thus, β satisfies strict serializability. ∎

Theorem 2. *Suppose $\mathcal{I}_A$ is an invariant that holds for any execution β of A that satisfies strict serializability. Then $\mathcal{I}_A$ also holds for any execution α of A that satisfies RSS.*

Proof. Let α be an arbitrary execution of A that satisfies RSS. We must show that $\mathcal{I}_\mathrm{A}$ is true for the final state s of α.

By Lemma 1, there is an equivalent execution β that satisfies strict serializability. Let s' be the final state of β. Because $\alpha|P_i = \beta|P_i$ for all P_i, it is easy to see that $s' = s$. By assumption, $\mathcal{I}_\mathrm{A}$ is true of s', so $\mathcal{I}_\mathrm{A}$ is also true of s. ∎

We prove similar results for RSC and linearizability in our technical report [35].

4 Practical Implications

Lemma 1 shows we can transform any RSS execution into an equivalent strictly serializable one. Theorem 2 shows this is sufficient for RSS to maintain application invariants.

While this transformation preserves the order of each process's actions, however, the order of causally unrelated actions, e.g., the order of Alice and Bob's Web requests handled by different servers, may not be. In fact, this is why anomalies like $\mathcal{A}_2$ and $\mathcal{A}_3$ are possible with RSS and RSC.

Further, RSS (RSC) is defined with respect to a potentially composite service. The results above thus assume a set of distinct services can together guarantee RSS (RSC), even if processes interact with multiple services, but they do not specify how this is achieved.

In the remainder of this section, we first describe how multiple services can be composed such that their composition guarantees RSS (§4.1). We then discuss supporting applications whose processes interact via message passing (§4.2). For ease of exposition, the discussion focuses on RSS but applies equally to RSC.

4.1 Composing RSS Services

A set of RSS services must always ensure the values returned by their transactions reflect a global total order spanning all services. This is straightforward with strictly serializable services because real-time order is universal across services.

With RSS, however, some pairs of transactions, such as causally unrelated read-only transactions, may be reordered with respect to real time. As a result, the states observed by processes as they interact with multiple services can form cycles (e.g., P_1 reads $x = 1$ then $y = 0$ while P_2 reads $y = 1$ then $x = 0$), precluding a total order. Service builders thus must implement one additional mechanism, *real-time fences*, to allow a set of RSS services to globally guarantee RSS.

Function	Description
RegisterService(name, fence_f)	Register new service.
UnregisterService(name)	Unregister service.
StartTransaction(name)	Start txn at service.

Figure 3. `libRSS` Interface. `libRSS` helps RSS service builders implement composition by invoking the necessary real-time fences.

A real-time fence f_x at RSS service x provides the following guarantee: for each pair of transactions T_1 and T_2 at service x, if $T_1 \rightsquigarrow f_x$ and $f_x \rightarrow T_2$, then $T_1 <_{S_x} T_2$, where $<_{S_x}$ is the total order of x's transactions. Every transaction that causally precedes the fence must be serialized before any transaction that follows the fence in real time. Intuitively, a process that issues a real-time fence ensures all other processes observe state that is at least as new as the state it observed. Thus, if each process issues a fence at its previous service before interacting with another, the fences prevent cycles in the states observed by multiple processes as they cross service boundaries. (We discuss the service-specific implementation of real-time fences for Spanner-RSS and Gryff-RSC in Sections 5 and 7.)

Although the need to implement a fence for each RSS service places an additional burden on service builders, using real-time fences to guarantee a global total order across services does not require changes to applications. The client libraries of the RSS services can insert real-time fences as necessary at run time. To this end, we implement a meta-library, `libRSS`, to aid service builders with composition. Figure 3 shows its interface.

At initialization, an RSS service's client library registers itself with the `libRSS` meta-library, passing it a unique name and a callback that implements its fence. The meta-library keeps an in-memory registry of all RSS services. During execution, the client library must simply notify the meta-library before starting a new transaction.

With these calls, the meta-library implements composition without intervention from application programmers. Every time an RSS client starts a transaction, the meta-library checks if the transaction is at the same service as the previous one, if any. If not, `libRSS` invokes the prior service's fence. In our technical report [35], we prove that if each service's real-time fence provides the guarantee described above and `libRSS` follows this simple protocol, then the composition of a set of RSS services globally guarantees RSS.

4.2 Capturing Causality

A meta-library that issues real-time fences is sufficient to guarantee RSS for applications whose processes interact

solely through a set of RSS services. But for those whose processes also interact through message passing, an RSS service must ensure causality is respected across these interactions.

For instance, recall our photo-sharing application and assume Alice is using her browser, which sends requests to Web servers that interact with an RSS key-value store. If one server reads and transmits a photo to Alice's browser and the browser subsequently reads the same photo via a second server, the key-value store must ensure causality is respected across the two transactions—the second must not return `null`. But if the store is unaware of the causal constraint between the two read-only transactions, then this may not be guaranteed.

One approach is to require application processes to issue a fence before such out-of-band interactions. For instance, the Web server must issue one before transmitting the response back to Alice's browser. Depending on the structure of the application, however, this may be inefficient.

A better approach is to use a context propagation framework [61] to pass metadata between the interacting processes. This would ensure the second Web server has the necessary metadata to convey causality before it interacts with the RSS store. This context must also include the name of the last RSS service the process interacted with, so `libRSS` can correctly implement composition.

5 Spanner-RSS

Spanner is a globally distributed, transactional database [22]. It uses synchronized clocks to guarantee strict serializability [74]. While Spanner is designed to provide low-latency read-only (RO) transactions most of the time, they may block, increasing tail latency significantly. Such increases in the tail latency of low-level services can translate into increases in common-case, user-visible latency [24].

Our variant of Spanner's protocol, Spanner-RSS, improves tail latency for RO transactions by relaxing the constraints on read-only transactions in accordance with RSS. (We prove it provides RSS in our technical report [35].)

Spanner background. Spanner is a multi-versioned key-value store. Keys are split across many shards, and shards are replicated using Multi-Paxos [46]. Clients atomically read and write keys at multiple shards using transactions.

Spanner's read-write (RW) transactions use two-phase locking [15] and a variant of two-phase commit [34]. Each shard's Paxos leader serves as a participant or coordinator. Further, using its TrueTime API, Spanner gives each transaction a commit timestamp that is guaranteed to be between the transaction's real start and end times.

During execution, clients acquire read locks and buffer writes. To commit, the client chooses a coordinator and sends its writes to the shards. Each participant then does the following: (1) ensures the transaction still holds its read locks, (2) acquires write locks, (3) chooses a prepare timestamp,

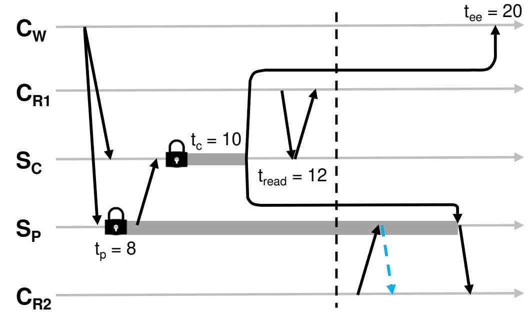


Figure 4. Example execution where Spanner-RSS's RO transaction returns before Spanner's. (Replication is omitted.) Client C_W is committing writes to two shards, S_C and S_P; S_C is the coordinator. C_{R1} reads C_W's writes before C_W's transaction finishes. Strict serializability still mandates C_{R2}'s read also includes them. Conversely, with Spanner-RSS, C_{R2}'s read returns immediately (shown by blue dotted line).

(4) replicates the prepare success, and (5) notifies the coordinator. Assuming all participants succeed, the coordinator finishes similarly: It checks read locks, acquires write locks, chooses the transaction's commit timestamp, and replicates the commit success. Finally, the coordinator releases its locks and sends the outcome to the client and participants.

To guarantee strict serializability, each participant ensures its prepare timestamp is greater than the timestamps of any previously committed or prepared transactions. The coordinator chooses the commit timestamp similarly but also ensures it is greater than the transaction's start time and greater than or equal to all of the prepare timestamps. Combined with commit wait [22], this ensures the transaction's commit timestamp is between its start and end times.

Strict serializability unnecessarily blocks ROs. Many workloads are dominated by reads [16, 73, 83]. Thus, Spanner also includes an optimized RO transaction protocol to make the majority of transactions as fast as possible. Spanner's RO transactions are strictly serializable but unlike RW transactions, only require one round trip between a client and the participant shards. As a result, RO transactions have significantly lower latency than RW transactions.

RO transactions, however, must sometimes block to ensure strict serializability. RO transactions in Spanner read at a client assigned timestamp $t_{\text{read}} = \textit{TT.now.latest}$, which TrueTime guarantees is after $T_{\text{RO}}.\textit{start}$. When a read arrives at a shard with t_{read} greater than the prepare timestamp of some conflicting RW transaction T_{RW}, it must block until the shard learns if T_{RW} commits at some time t_c or aborts. Otherwise, T_{RO} risks violating strict serializability: if $t_c < t_{\text{read}}$ because $T_{\text{RW}}.\textit{end} < T_{\text{RO}}.\textit{start}$ ($t_c < T_{\text{RW}}.\textit{end} < T_{\text{RO}}.\textit{start} < t_{\text{read}}$), then strict serializability mandates that T_{RO} includes T_{RW}'s writes. Because they must potentially wait while a RW transaction executes two-phase commit, blocked RO transactions can have significantly higher latency.

Algorithm 1 Spanner-RSS Client

```
1:  state t_min ← 0
2:  function Client::ROTransaction(K)
3:      S ← ShardLeaders(K)
4:      t_read ← TrueTime::Now.Latest
5:      send ROCommit(K, t_read, t_min) to all s ∈ S
6:      wait receive ROFastReply(P_s, V_s) from all s ∈ S
7:      P, V ← ⋃_{s∈S} P_s, ⋃_{s∈S} V_s
8:      t_snap ← CalculateSnapshotTS(K, V)
9:      while CheckSnapshot(P, t_snap) ≠ COMMIT do
10:         wait for ROSlowReply(i, d, t_c, V′) from s ∈ S
11:         P, V ← UpdatePrepared(P, V, i, d, t_c, V′)
12:     t_min ← max(t_min, t_snap)
13:     return ReadAtTimestamp(V, t_snap)

14: function Client::CalculateSnapshotTS(K, V)
15:     t_snap ← 0
16:     for k ∈ K do
17:         V′ ← {(t_c, k′, v) ∈ V : k = k′}
18:         t_earliest ← min_{(t_c,k′,v)∈V′} t_c
19:         t_snap ← max(t_snap, t_earliest)
20:     return t_snap

21: function Client::CheckSnapshot(P, t_snap)
22:     t′_p ← min_{(i,t_p)∈P} t_p
23:     if t′_p ≤ t_snap then d ← WAIT else d ← COMMIT
24:     return d
```

One potential optimization to improve RO transaction latency would be to include the earliest client-side end time t_{ee} for each RW transaction. Then, RO transactions could avoid blocking if $t_{\text{read}} < t_{\text{ee}}$. Unfortunately, strict serializability disallows this optimization because it requires T_{RO} to observe T_{RW} even before t_{ee} if there is some other RO transaction that finishes before T_{RO} and includes any of T_{RW}'s writes.

Figure 4 shows an example. Because C_{R1}'s read observes C_W's RW transaction at S_C, strict serializability requires all future reads at both shards to include C_W's writes. Thus, C_{R2}'s read must block until C_W's RW transaction commits.

In contrast, RSS allows this optimization. C_{R1}'s transaction only imposes a constraint on reads that causally follow it, so C_{R2}'s read may still return an older value.

Spanner-RSS. Spanner-RSS is our variant of Spanner that improves tail RO transaction latency by often avoiding blocking, even when there are conflicting RW transactions. Intuitively, a RO transaction can avoid blocking by observing a state of the database as of some timestamp t_{snap} that is before its read timestamp t_{read} if it can infer the state satisfies regular sequential serializability.

Observing this state from before t_{read} is safe under RSS when three conditions are met: (1) there are no unobserved writes from a conflicting RW transaction that could have

Algorithm 2 Spanner-RSS Shard

```
1:  state 𝒫 ← {(i, ℓ, t_p, t_ee, W), ...}                         ▷ Prepared txns
2:  state 𝒟 ← {(t_c, k, v), ...}                                  ▷ Database
3:  function Shard::ROCommitRecv(c, K, t_read, t_min)
4:      wait until t_read ≤ Paxos::MaxWriteTS
5:      P ← {(i, ℓ, t_p, t_ee, W) ∈ 𝒫 | R ∩ W ≠ ∅ ∧ t_p ≤ t_read}
6:      B ← {(i, ℓ, t_p, t_ee, W) ∈ P | t_p ≤ t_min ∨ t_ee ≤ t_read}
7:      wait until all p ∈ B commit or abort
8:      V ← ReadAtTimestamp(𝒟, K, t_read)
9:      Q ← {(i, t_p) : (i, ℓ, t_p, t_ee, W) ∈ P \ B}
10:     send ROFastReply(Q, V) to c
11:     while P ≠ ∅ do
12:         wait until some p ∈ P commits or aborts
13:         if p = (i, ℓ, t_p, t_ee, W) commits at t_c then
14:             V ← ReadAtTimestamp(𝒟, K ∩ W, t_c)
15:             send ROSlowReply(i, COMMIT, t_c, V) to c
16:         else
17:             send ROSlowReply(i, ABORT, 0, ∅) to c
18:         P ← P \ {p}
```

ended before T_{RO} started; (2) there are no causal constraints that require T_{RO} to observe a write at a timestamp later than t_{snap}; and (3) its results are consistent with a sequential execution of transactions.

Spanner-RSS ensures each of these conditions are met. To ensure (1), RW transactions include a client-side earliest end time t_{ee}. To ensure (2), RO transactions include a minimum read time t_{min}. Finally, to ensure (3), before completing a RO transaction, clients ensure all returned values reflect precisely the state of the database at t_{snap}. Algorithms 1 and 2 show the full protocol.

Estimating, including, and enforcing t_{ee} for RW transactions. Each RW transaction includes an earliest client-side end time t_{ee}. The client estimates t_{ee} and includes it when it initiates two-phase commit (not shown). The shards then store t_{ee} while the transaction is prepared but not yet committed or aborted (Alg. 2, line 1). The client later ensures t_{ee} is less than the actual client-side end time by waiting until $t_{\text{ee}} <$ *TT.now.earliest*.

Enforcing a minimum timestamp for RO transactions. Each RO transaction includes a minimum read timestamp t_{min} to ensure it obeys any necessary causal constraints. Each client tracks this minimum timestamp and updates it after every transaction to include new constraints. After a RW transaction, it is set to the transaction's commit timestamp (not shown). After a RO transaction, it is set to be at least the transaction's snapshot time (Alg. 1, line 12). t_{min} thus captures the causal constraints on this RO transaction; it must observe a state that is at least as recent as its last write and any writes the client previously observed.

Avoiding blocking on shards with t_{ee} and t_{min}. Using t_{ee} and t_{min}, shards can infer when it is safe for a RO transaction to skip observing a prepared-but-not-committed RW

transaction (Alg. 2, line 6). It is safe unless the prepared transaction either must be observed due to a causal constraint ($t_p \leq t_{min}$) or could have ended before the RO transaction began ($t_{ee} \leq t_{read}$).

Reading at t_{snap}. Although each shard can now infer when a RO transaction can safely skip a prepared RW transaction, the values returned by multiple shards may not necessarily reflect a complete, consistent snapshot at t_{snap}. Thus, clients and shards take four additional steps to ensure a client always returns a consistent snapshot.

First, as in Spanner, a shard waits to process a RO transaction until its Paxos safe time is greater than t_{read} (Alg. 2, line 4) [22]. As a result, all future Paxos writes, and thus all future RW prepare timestamps, will be larger than t_{read}. Thus, the shard ensures it is returning information that is valid until at least t_{read}. (The Paxos safe time at leaders can be advanced immediately if it is within the leader's lease.) Second, shards include the commit timestamps t_c for the returned values (Alg. 2, lines 2, 8, and 10). Third, they return the prepare timestamp t_p for each skipped RW transaction with $t_p \leq t_{read}$ (Alg. 2, lines 9-10). (Because writes use locks, there is at most one per key.) Fourth, when a skipped RW transaction commits, a shard sends the commit timestamp and the written values in a slow path (Alg. 2, lines 13-15).

Before returning, the client examines the commit and prepare timestamps to see if the shards returned values that are all valid at some snapshot time. Specifically, it sets t_{snap} to the earliest time for which it has a value for all keys (Alg. 1, lines 15-20). Then, it sees if any prepared transactions have timestamps less than t_{snap} (Alg. 1, lines 22-23). If they all prepared after t_{snap}, the RO transaction returns immediately (Alg. 1, line 13).

If some transaction prepared before t_{snap}, however, the client must wait for slow replies from the shards (Alg. 1, lines 9-10). As the client learns of commits and aborts through the slow replies, it moves transactions out of the prepared set (Alg. 1, line 11), updates the values it will return (if $t_c \leq t_{snap}$), and potentially advances the earliest prepared timestamp (Alg. 1, line 22). Note that t_{snap} remains the same, so the latter continues until $t_{snap} < t'_p$, which is guaranteed by the time the final slow reply is received.

Performance discussion. Spanner-RSS's RO transaction latency is never worse and often better than Spanner's. When there are no conflicting RW transactions, RO transactions in both will return consistent results at t_{read}. When there are conflicting transactions, however, Spanner-RSS will often send fast replies quickly while Spanner blocks. Further, the fast replies let Spanner-RSS complete the RO transaction right away unless one of the shards returns a value with a commit timestamp that is greater than the prepared timestamp of a skipped RW transaction. Even then, the slow replies from Spanner-RSS's shards will be sent at the same time Spanner would unblock.

5.1 Real-Time Fences

As described above, to ensure the order of transactions reflects causality, a client tracks and enforces a minimum read timestamp t_{min}. Using t_{min}, a client ensures its next transaction will be ordered after any transaction that causally precedes it by ensuring the next transaction reflects a state of the database that is at least as recent as t_{min}.

A real-time fence must provide a slightly stronger guarantee. It must ensure that all transactions that causally precede it are serialized before any transaction that follows it in real time, regardless of the latter transaction's originating client. While this is guaranteed for future RW transactions since they already respect their real-time order, the same is not true of future RO transactions. Thus, when executed at a client with a minimum read timestamp t_{min}, a fence in Spanner-RSS must ensure that all future RO transactions reflect a state that is at least as recent t_{min}.

To achieve this, Spanner-RSS's real-time fences leverage the following observation: If L is the maximum difference between t_c and t_{ee} for any RW transaction, then a RO transaction that starts after $t_c + L$ will reflect all writes with timestamps less than or equal to t_c. After $t_c + L$, a RO transaction cannot skip reading a RW transaction with commit timestamp t_c (since $t_{ee} \leq t_c + L < t_{read}$).

As a result, fences in Spanner-RSS are simple. To ensure all future RO transactions reflect a state that is at least as recent as t_{min}, a fence blocks until $t_{min} + L < \mathit{TT.now.earliest}$.

6 Spanner-RSS Evaluation

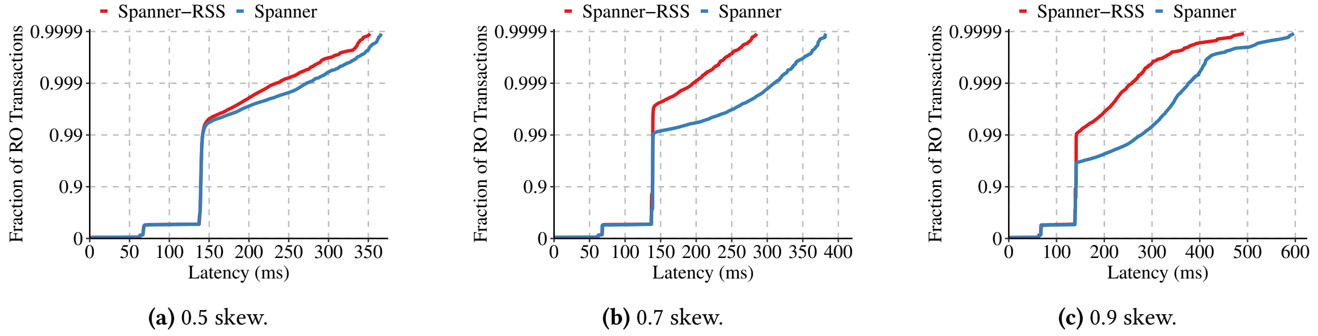
Our evaluation of Spanner-RSS aims to answer two questions: Does Spanner-RSS improve tail latency for read-only transactions (§6.1), and what performance overhead does Spanner-RSS's read-only transaction protocol impose (§6.2)?

We implement the Spanner and Spanner-RSS protocols in C++ using TAPIR's experimental framework [94, 95]. Each shard is single-threaded. The implementation reuses TAPIR's implementation of view-stamped replication [71] instead of Multi-Paxos [46] but is otherwise faithful. Our code and experiment scripts are available online [3].

The implementation includes two optimizations not presented in Section 5: First, a skipped, prepared transaction's writes are returned in the fast path instead of the slow path, allowing the client to return faster in some cases, e.g., if it learns the transaction already committed at a different shard.

Second, when a transaction blocks as part of wound-wait [78], it estimates how long it blocked and advances its local estimate of t_{ee} by that amount. The coordinator then aggregates the shards' t_{ee} values and returns the maximum to the client, which waits until it has passed. This reduces the chance a RO transaction will be blocked by a RW transaction whose t_{ee} has become inaccurate because of lock contention.

Unless otherwise specified, experiments ran on Amazon's EC2 platform [1]. Each `t2.large` instance has 2 vCPUS and



(a) 0.5 skew. **(b)** 0.7 skew. **(c)** 0.9 skew.

Figure 5. Spanner-RSS offers better tail latency for RO transactions on Retwis. In contrast to Spanner's, its RO transactions can often avoid blocking when there are concurrent, conflicting RW transactions.

8 GB RAM. We use three shards with three replicas each. One shard leader is in each of California, Virginia, and Ireland, and the replicas are in the other two data centers. The round trip times are as follows: CA-VA is 62 ms, CA-IR is 136 ms, and VA-IR is 68 ms. Our emulated TrueTime error is 10 ms, the p99.9 value observed in practice [22].

To calculate t_{ee} for RW transactions, clients use the round-trip latencies above. In our implementation, clients use them to calculate, for each set of participants, the coordinator choice that yields the minimum commit latency. It stores these choices and the commit latencies, and the latter is used to calculate t_{ee}. To avoid increasing RW latency, the round-trip latencies above are the minimum observed, and clients calculate t_{ee} with respect to *TT.now.earliest*.

Each client executes the Retwis workload [47] over a database of ten million keys and values. Retwis clients execute transactions in the following proportions: 5% add-user, 15% follow/unfollow, 30% post-tweet, and 50% load-timeline. The first three are RW transactions, and the last is RO. We generate keys according to a Zipfian distribution [37] with skews ranging from 0.5 to 0.9. Such read-write ratios and skews are representative of real workloads [19, 92].

Unless otherwise specified, we generate load with a fixed number of partly open clients [79]. Partly open clients use three parameters to model user behavior: sessions arrive at rate λ according to a Poisson process; after each transaction in a session, the client chooses to stay with probability p; and if it does, it waits for a think time H. The clients use a separate t_{min} for each session. We set $H = 0$ since this yields the worst performance for Spanner-RSS. Further, we set $p = 0.9$, so the average session length is ten transactions, matching measurements from real deployments [83]. Finally, for each workload, we set λ such that the offered load is 70-80% of the maximum throughput. Each data center contributes an equal fraction of the load.

6.1 Spanner-RSS Reduces RO Tail Latency

We first compare the latency distributions for RO and RW transactions with Spanner and Spanner-RSS as skew varies. Spanner-RSS's RO transactions have lower latency than Spanner's due to less blocking during conflicting RW transactions. These improvements do not harm RW transaction latency because Spanner-RSS's protocol simply requires passing around an extra timestamp with RW transactions.

Figure 5 compares the tail latency distributions of RO transactions at three skews. (We omit the distributions for RW transactions after verifying they are identical.) Spanner-RSS improves RO tail latency in all cases. When contention is low (Figure 5a), Spanner's RO transactions offer low tail latency; up to p99, their latency is bounded only by wide-area latency. Above this, however, it starts increasing. At p99.9, Spanner-RSS offers a 14% (38 ms) reduction in tail latency.

Spanner-RSS offers larger improvements at higher skews. In Figure 5b, latency consistently decreases by at least 76 ms above p99.5. This is up to a 45% reduction; at p99.9, it is a 37% (114 ms) reduction.

With a skew of 0.9, (Figure 5c), Spanner's RO transaction latency starts increasing at lower percentiles. As a result, Spanner-RSS reduces p99 RO latency by 49% (135 ms). With high contention, however, improvements farther out on the tail (e.g., above p99.95) are more inconsistent. Increased waiting by RW transactions for wound-wait [78] make the earliest end time estimates less accurate. Further, each session's t_{min} advances more rapidly and in turn, increases the chance a RO transaction must block.

6.2 Spanner-RSS Imposes Little Overhead

We now compare the two protocol under heavy load to quantify the overhead Spanner-RSS incurs from its additional protocol complexity. Because the number and size of its additional messages is small, Spanner-RSS's performance should be comparable to Spanner's.

To stress the implementations, we use a uniform workload, set the TrueTime error to zero, and place all shards in one data center. Since it does not depend on wide-area latencies, we ran this experiment on CloudLab's Utah platform [26]. Each `m510` machines has 8 physical cores, 64 GB RAM, and a 10 Gbit NIC. Inter-data-center latency is less than 200 µs. We use eight shards, so each leader has a dedicated physical CPU on one server.

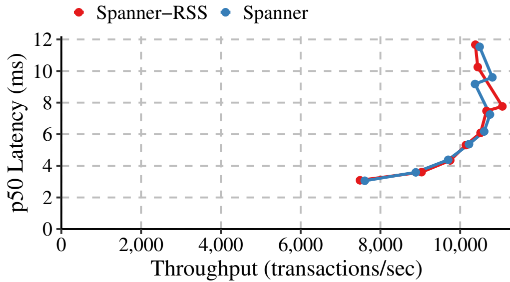


Figure 6. Spanner-RSS does not significantly impact Spanner's performance at high load.

Figure 6 compares the throughput and median latency as we increase the number of closed-loop clients. As shown, Spanner-RSS does not significantly impact the server's maximum throughput. Spanner-RSS's is within a few hundreds of transactions per second of Spanner's, and its latency is within a few milliseconds.

7 Gryff-RSC

Gryff is a geo-replicated key-value store that supports non-transactional reads, writes, and atomic read-modify-writes (rmws) [18]. It provides linearizability using a hybrid shared register and consensus protocol. Reads and writes are executed using a shared register protocol to provide bounded tail latency whereas rmws are executed using a consensus protocol, which is necessary for correctness.

We introduce Gryff-RSC, which provides regular sequential consistency and is able to reduce the bound on tail latency from two round trips to a quorum of replicas to one round trip. This section gives an overview of Gryff-RSC's design and evaluation. We describe the full design and prove it guarantees RSC in our technical report [35].

7.1 Gryff-RSC Design Overview

Read operations in Gryff consist of an initial read phase that contacts a quorum of replicas to learn of the most-recent value they know of for a given key. If the quorum returns the same values, then the read finishes. If the quorum returns different values, however, the read continues to a second, write-back phase that writes the most-recent value to a quorum before the read ends. This second phase is necessitated by linearizability: once this read ends, any future reads must observe this or a newer value.

Regular sequential consistency relaxes this constraint: before the write finishes, only causally later reads must observe this or a newer value. This enables Gryff-RSC's reads to always complete in one phase. On a read, instead of immediately writing the observed value back to a quorum, a Gryff-RSC client piggybacks it onto the first phase of its next operation. Replicas write the piggybacked value before processing the next operation. Causally later operations by the same client are thus guaranteed to see this or a newer value. By transitivity then, causally later operations at other clients, e.g., by the reads-from relation, will also observe this or a newer value.

Piggybacking a read's second phase onto the next operation ensures a client's next operation can be serialized after all operations that causally precede it. Similarly, a real-time fence must ensure all future operations, including those from other clients, are ordered after any operation that causally precedes it. By RSC, future writes and rmws are already required to respect their real-time order, but the same is not true of future reads. Thus, to execute a real-time fence in Gryff-RSC, a client writes back the key-value pair, if any, that would have been piggybacked onto its next operation. This guarantees future reads return values that are at least as recent as any operation that causally precedes the fence.

7.2 Gryff-RSC Evaluation

Our evaluation of Gryff-RSC aims to answer two questions: Does Gryff-RSC offer better tail read latency on important workloads (§7.3), and what are the performance costs of Gryff-RSC's protocol (§7.4)?

We implement Gryff-RSC in Go using the same framework as Gryff [18], and our code and experiment scripts are available online [2]. We keep all of Gryff's optimizations enabled. All experiments ran on the CloudLab [26] machines described in Section 6.2, and we emulate a wide-area environment. We use five replicas, one in each emulated geographic region, because with Gryff's optimizations, reads already always finish in one round trip with three replicas. An equal fraction of the clients are in each region. Table 2 shows the emulated round-trip times.

	CA	VA	IR	OR	JP
CA	0.2				
VA	72.0	0.2			
IR	151.0	88.0	0.2		
OR	59.0	93.0	145.0	0.2	
JP	113.0	162.0	220.0	121.0	0.2

Table 2. Emulated round-trip latencies (in ms).

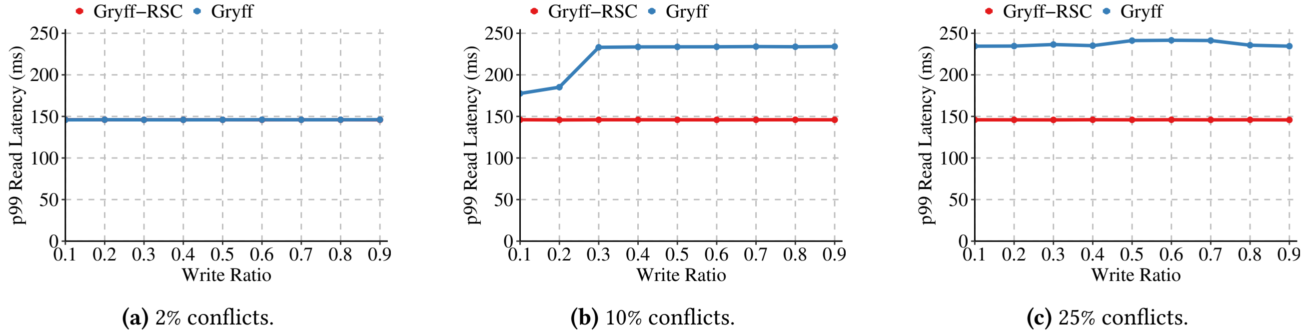
We generate load with 16 closed-loop clients. With this number, servers are moderately loaded. Each client executes the YCSB workload [21], which includes just reads and writes. We vary the rate of conflicts and the read-write ratio.

7.3 Gryff-RSC Reduces Read Tail Latency

Figure 7 compares Gryff and Gryff-RSC's p99 read latency across a range of conflict percentages and read-write ratios. We omit similar plots for writes because write performance is identical in the two systems.

With few conflicts (Figure 7a), nearly all of Gryff's reads complete in one round, so Gryff-RSC cannot offer an improvement. p99 latency for both systems is 145 ms.

As Figures 7b and 7c show, however, as the rate of conflicts increases, more of Gryff's reads must take its slow path, incurring two wide-area round trips. This increases Gryff's



(a) 2% conflicts. **(b)** 10% conflicts. **(c)** 25% conflicts.

Figure 7. For moderate- and high-contention workloads, Gryff-RSC offers roughly a 40% reduction in p99 read latency compared to Gryff. As the conflict ratio increases, Gryff-RSC's benefits start at lower write ratios.

p99 latency by 61% (from 145 ms to 234 ms). On the other hand, Gryff-RSC's reads only require one round trip, so p99 latency remains at 145 ms. At lower write ratios, the magnitude of Gryff-RSC's improvement over Gryff increases with the rate of conflicts.

Further, because reads always finish in one round, Gryff-RSC offers even larger latency improvements farther out on the tail (not shown). For instance, with 10% conflicts and a 0.3 write ratio, Gryff-RSC reduces p99.9 latency by 49% (from 290 ms to 147 ms).

7.4 Gryff-RSC Imposes Negligible Overhead

We also quantify the performance overhead of Gryff-RSC's piggybacking mechanism, but we omit the plots due to space constraints. We compare Gryff and Gryff-RSC's throughput and median latency as we increase the number of clients. As in Section 6.2, we disable wide-area emulation. With a 10% conflict ratio, we run two workloads: 50% reads-50% writes and 95% reads-5% writes (matching YCSB-A and YCSB-B [21]). In both cases, Gryff-RSC's throughput and latency are within 1% of Gryff's, suggesting the overhead from Gryff-RSC's protocol changes are negligible.

8 Related Work

This section discusses related work on consistency models, explicitly reasoning about invariants, equivalence results, and strictly serializable and linearizable services.

Consistency models. Due to their implications for applications and services, consistency models have been studied extensively. In general, given an application, more invariants hold and fewer anomalies are possible with stronger models. But weaker models allow for better-performing services.

RSS and RSC are distinct from prior works because they are the first models that are invariant-equivalent to strict serializability and linearizability, respectively. They achieve this by guaranteeing that transactions (operations) appear to execute sequentially, in an order consistent with a set of causal constraints. Prior works are not invariant-equivalent to strict serializability (linearizability) because either they do not guarantee equivalence to a sequential execution or do not capture all of the necessary causal constraints.

The discussion below generally proceeds from the strongest to the weakest consistency models. Since we have already discussed strict serializability [74], process-ordered serializability [23, 55], linearizability [36], and sequential consistency [44] extensively, we focus here on other models. (We also provide a technical comparison between RSS, RSC, and their proximal models in our technical report [35].)

Like RSS, CockroachDB's consistency model (CRDB) [85] lies between strict serializability and PO serializability. CRDB guarantees conflicting transactions respect their real-time order [85], but it gives no such guarantee for non-conflicting transactions, which can lead to invariant violations. For instance, in a slight modification to our photo-sharing application, assume clients issue a single write to add a photo and included in this write is a logical timestamp comprising a user ID and a counter. Further, assume clients can issue a RO transaction for a user's photos. With CRDB, if Alice adds two photos and those transactions execute at different Web servers, a RO transaction that is concurrent with both may only return the second photo. If the application requires a user's photos to always appear in timestamp order, then it would be correct with a strictly serializable database but not with CRDB.

Similarly, like RSC, OSC(U) [48] lies between linearizability and sequential consistency. OSC(U) guarantees writes respect their real-time order. Reads, however, may return stale values [48], so some pairs of reads (e.g., those invoked by different processes that also communicate via message passing) may return values inconsistent with their causal order. As a result, the non-transactional version of $\mathcal{I}_2$ discussed in Section 2.6 does not hold with OSC(U). On the other hand, OSC(U) allows services to achieve much lower read latency than what is currently achievable with RSC.

PO serializability and sequential consistency impose fundamental performance constraints on services [52], so many weaker models, both transactional [4, 7, 10, 14, 27, 65, 74, 75, 84, 88] and non-transactional [5, 13, 20, 52, 54, 81, 86], have been developed. These weaker models allow for services

with much better performance than what is currently achievable with RSS or RSC. For example, a causal+ storage system can process all operations without synchronous, cross-data-center communication [54]. But application invariants break with these models because they do not guarantee equivalence to a sequential execution of either transactions or operations. Thus, they present developers with a harsh trade-off between service performance and application correctness.

Based on the observation that some invariants hold with weaker consistency models, other work proposes combinations of weak and strong guarantees for different operations [42, 50, 51, 87]. This allows these services to offer dramatically better performance for a subset of operations. Maintaining application correctness while using these services, however, requires application programmers to choose the correct consistency for each operation.

Finally, three other works use causal or real-time constraints in innovative ways [12, 62, 93]. First, Δ-causal messaging applies real-time guarantees to a different domain where messages have limited, time-bounded relevance (e.g., video streaming) [12]. Second, real-time causal strengthens causal consistency by ensuring writes respect their real-time order [62]. But because real-time causal does not capture all necessary causal constraints, $\mathcal{I}_2$ would not hold.

Third, TACT gives application developers fine-grained control over its consistency [93]. For instance, an application can set a different staleness bound for each invocation to control how old (in real time) the values returned by the operation may be. (Setting zero for all operations provides strict serializability.) Compared to RSS, TACT's fine-grained control allows for services with better performance but requires developers to choose the correct bounds when ensuring their application's correctness.

Reasoning about explicit invariants. Several tools and techniques have been proposed for reasoning about the correctness of applications that run on services with weaker consistency [8, 17, 33, 49, 69, 76]. For example, SIEVE [49] uses static and dynamic analysis of Java application code to determine the necessary consistency level for operations to maintain a set of explicitly written invariants. Brutschy et al. [17] describe a static analysis tool for identifying non-serializable application behaviors that are possible when running on a causally consistent service. Gotsman et al. [33] introduce a proof rule (and accompanying static analysis tool [69]) that enables modular reasoning about the consistency level required to maintain explicit invariants.

These tools and techniques help application programmers ensure explicit invariants hold when using services with weaker consistency. In contrast, RSS (RSC) services ensure the same application invariants as strictly serializable (linearizable) services. This makes it easier to build correct applications because programmers can write code without stating invariants, running static analyses, or writing proofs.

Equivalence results. Other works have leveraged the notion of equivalence, or indistinguishability, to prove interesting theoretical results [9, 30, 32, 58]. In fact, our results here are inspired by them. But while we leverage some of their ideas and techniques, these works apply equivalence to different ends, e.g., to prove bounds on clock synchronization [58] or show there are fundamental differences in the performance permitted by different consistency models [9].

Strictly serializable services. Spanner is a globally distributed, strictly serializable database [22]. Since its publication, other such services have been developed [41, 63, 67, 68, 77, 85, 89, 91, 94, 95]. These services has largely focused on improving the throughput [77, 89] and latency [41, 63, 77, 91, 94, 95] of read-write transactions, which can incur multiple inter-data-center round trips in Spanner.

Because they only require one round trip between a client and the participant shards, Spanner's RO transactions continue to perform as well or better than those of other services. These improvements are thus orthogonal to those offered by Spanner-RSS, and in fact, weakening the consistency of these other services to RSS may allow for designs that combine their improved RW transaction performance with RO transactions that are competitive with Spanner-RSS's.

Linearizable services. Gryff is a recent geo-replicated storage system that combines shared registers and consensus [18]. Many other protocols have been developed to provide replicated and linearizable storage [6, 28, 29, 31, 40, 46, 64, 66, 70–72, 96]. Weakening the consistency of these other services to RSC is likely to enable new variants of their designs that improve their performance.

9 Conclusion

Existing consistency models offer a harsh trade-off to application programmers; they often must choose between application correctness and performance. This paper presents two new consistency models, regular sequential serializability and regular sequential consistency, to ease this trade-off. RSS and RSC maintain application invariants while permitting new designs that achieve better performance than their strictly serializable or linearizable counterparts. To this end, we design variants of two existing systems, Spanner-RSS and Gryff-RSC, that guarantee RSS and RSC, respectively. Our evaluation demonstrates significant (40% to 50%) reductions in tail latency for read-only transactions and reads.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Rodrigo Rodrigues, for their helpful comments and feedback. We are also grateful to Khiem Ngo for his comments on an earlier version of this paper. This work was supported by the National Science Foundation under grant CNS-1824130.

References

[1] 2021. Amazon Elastic Compute Cloud. https://aws.amazon.com/ec2/.

[2] 2021. Gryff-RSC. https://github.com/princeton-sns/gryff-rs/.

[3] 2021. Spanner-RSS. https://github.com/princeton-sns/spanner-rss/.

[4] Atul Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* Ph.D. Dissertation. MIT, Cambridge, MA. Advisor(s) Barbara Liskov.

[5] Mustaque Ahamad, Gil Neiger, James E Burns, Prince Kohli, and Phillip W Hutto. 1995. Causal Memory: Definitions, Implementation, and Programming. *Distributed Computing* 9, 1 (1995), 37–49.

[6] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. 2020. WPaxos: Wide Area Network Flexible Consensus. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2020), 211–223.

[7] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro. 2016. Cure: Strong Semantics Meets High Availability and Low Latency. In *IEEE International Conference on Distributed Computing Systems.* IEEE, Nara, Japan, 405–414.

[8] Jade Alglave and Patrick Cousot. 2017. Ogre and Pythia: An Invariance Proof Method for Weak Consistency Models. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, Paris, France, 3–18.

[9] Hagit Attiya and Jennifer L. Welch. 1994. Sequential Consistency versus Linearizability. *ACM Transactions on Computer Systems* 12, 2 (May 1994), 91–122.

[10] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2016. Scalable Atomic Visibility with RAMP Transactions. *ACM Transactions on Database Systems* 41, 3, Article 15 (July 2016).

[11] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Bolt-on Causal Consistency. In *ACM International Conference on Management of Data.* ACM, New York, NY, 761–772.

[12] Roberto Baldoni, Achour Mostefaoui, and Michel Raynal. 1996. Causal Delivery of Messages with Real-Time Data in Unreliable Networks. *Real-Time Systems* 10, 3 (May 1996), 245–262.

[13] Valter Balegas, Diogo Serra, Sérgio Duarte, Carla Ferreira, Marc Shapiro, Rodrigo Rodrigues, and Nuno Preguiça. 2015. Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants. In *IEEE Symposium on Reliable Distributed Systems.* IEEE, Montreal, Canada, 31–36.

[14] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *ACM International Conference on Management of Data.* ACM, San Jose, CA, 1–10.

[15] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA.

[16] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. 2013. TAO: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference.* USENIX, San Jose, CA, 49–60.

[17] Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin Vechev. 2018. Static Serializability Analysis for Causal Consistency. In *ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, Philadelphia, PA, 90–104.

[18] Matthew Burke, Audrey Cheng, and Wyatt Lloyd. 2020. Gryff: Unifying Consensus and Shared Registers. In *USENIX Symposium on Networked Systems Design and Implementation.* USENIX, Santa Clara, CA.

[19] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. 2020. HotRing: A Hotspot-Aware In-Memory Key-Value Store. In *USENIX Conference on File and Storage Technologies.* USENIX, Santa Clara, CA, 239–252.

[20] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proceedings of the Very Large Data Bases Endowment* 1, 2 (Aug. 2008), 1277–1288.

[21] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *ACM Symposium on Cloud Computing.* ACM, Indianapolis, IN, 143–154.

[22] 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. 2013. Spanner: Google's Globally Distributed Database. *ACM Transactions on Computer Systems* 31, 3, Article 8 (Aug. 2013), 22 pages.

[23] Khuzaima Daudjee and Kenneth Salem. 2004. Lazy Database Replication With Ordering Guarantees. In *IEEE International Conference on Data Engineering.* IEEE, Boston, MA, 424–435.

[24] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Communications of the ACM* 56, 2 (Feb. 2013), 74–80.

[25] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. In *ACM Symposium on Operating Systems Principles.* ACM, Stevenson, WA, 205–220.

[26] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *USENIX Annual Technical Conference.* USENIX, Renton, WA, 1–14.

[27] Sameh Elnikety, Fernando Pedone, and Willy Zwaenepoel. 2004. *Generalized Snapshot Isolation and a Prefix-consistent Implementation.* Technical Report IC/2004/21. School of Computer and Communication Sciences, EPFL, Lausanne, Switzerland.

[28] Vitor Enes, Carlos Baquero, Alexey Gotsman, and Pierre Sutra. 2021. Efficient Replication via Timestamp Stability. In *ACM European Conference on Computer Systems.* ACM, Virtual Event, 178–193.

[29] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. 2020. State-Machine Replication for Planet-Scale Systems. In *ACM European Conference on Computer Systems.* ACM, Heraklion, Greece, 15 pages.

[30] Michael Fischer, Nancy Lynch, and Michael Paterson. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM* 32, 2 (1985), 374–382.

[31] Vasilis Gavrielatos, Antonios Katsarakis, Vijay Nagarajan, Boris Grot, and Arpit Joshi. 2020. Kite: Efficient and Available Release Consistency for the Datacenter. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, San Diego, CA, 1–16.

[32] Kenneth Goldman and Katherine Yelick. 1993. *A Unified Model for Shared-Memory and Message-Passing Systems.* Technical Report WUCS-93-35. Department of Computer Science and Engineering, Washington University in St. Louis, St. Louis, MO.

[33] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. 'Cause I'm Strong Enough: Reasoning about Consistency Choices in Distributed Systems. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, St. Petersburg, FL, 371–384.

[34] Jim Gray. 1978. *Notes on data base operating systems.* Springer, Berlin, Germany, 393–481.

[35] Jeffrey Helt, Matthew Burke, Amit Levy, and Wyatt Lloyd. 2021. Regular Sequential Serializability and Regular Sequential Consistency. arXiv:2109.08930 [cs.DC]

[36] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (July 1990), 463–492.

[37] W. Hörmann and G. Derflinger. 1996. Rejection-Inversion to Generate Variates from Monotone Discrete Distributions. *ACM Transactions on Modeling and Computer Simulation* 6, 3 (July 1996), 169–184.

[38] Qi Huang, Petchean Ang, Peter Knowles, Tomasz Nykiel, Iaroslav Tverdokhlib, Amit Yajurvedi, Paul Dapolito, IV, Xifan Yan, Maxim Bykov, Chuen Liang, Mohit Talwar, Abhishek Mathur, Sachin Kulkarni, Matthew Burke, and Wyatt Lloyd. 2017. SVE: Distributed Video Processing at Facebook Scale. In *ACM Symposium on Operating Systems Principles*. ACM, Shanghai, China, 87–103.

[39] Patrick Hunt, Mahadev Konar, Flavio Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free coordination for Internet-scale systems. In *USENIX Annual Technical Conference*. USENIX, Boston, MA, 1–14.

[40] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. 2020. Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Lausanne, Switzerland, 201–217.

[41] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: Multi-data Center Consistency. In *ACM European Conference on Computer Systems*. ACM, Prague, Czech Republic, 113–126.

[42] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. 1992. Providing High Availability Using Lazy Replication. *ACM Transactions on Computer Systems* 10, 4 (Nov. 1992), 360–391.

[43] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21, 7 (1978), 558–565.

[44] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *ACM Transactions on Computer Systems* C-28, 9 (Sept. 1979), 690–691.

[45] Leslie Lamport. 1986. On Interprocess Communication: Parts I and II. *Distributed Computing* 1, 2 (June 1986), 77–101.

[46] Leslie Lamport. 1998. The part-time parliament. *ACM Transactions on Computer Systems* 16, 2 (1998), 133–169.

[47] Costin Leau. 2013. Spring Data Redis - Retwis-J. https://docs.spring.io/spring-data/data-keyvalue/examples/retwisj/current/.

[48] Kfir Lev-Ari, Edward Bortnikov, Idit Keidar, and Alexander Shraer. 2017. Composing Ordered Sequential Consistency. *Inform. Process. Lett.* 123 (July 2017), 47–50.

[49] Cheng Li, Joao Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. 2014. Automating the Choice of Consistency Levels in Replicated Systems. In *USENIX Annual Technical Conference*. USENIX, Philadelphia, PA, 281–292.

[50] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *USENIX Symposium on Operating Systems Design and Implementation*. USENIX, Hollywood, CA, 265–278.

[51] Cheng Li, Nuno Preguiça, and Rodrigo Rodrigues. 2018. Fine-grained consistency for geo-replicated systems. In *USENIX Annual Technical Conference*. USENIX, Boston, MA, 359–372.

[52] Richard J. Lipton and Jonathan S. Sandberg. 1988. *PRAM: A scalable shared memory*. Technical Report TR-180-88. Department of Computer Science, Princeton University, Princeton, NJ.

[53] Wyatt Lloyd, Michael Freedman, Michael Kaminsky, and David Andersen. 2013. Stronger Semantics For Low-latency Geo-replicated Storage. In *USENIX Symposium on Networked Systems Design and Implementation*. USENIX, Lombard, IL, 313–328.

[54] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *ACM Symposium on Operating Systems Principles*. ACM, Cascais, Portugal, 401–416.

[55] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. 2016. The SNOW Theorem and Latency-Optimal Read-Only Transactions. In *USENIX Symposium on Operating Systems Design and Implementation*. USENIX, Savannah, GA, 135–150.

[56] Haonan Lu, Siddhartha Sen, and Wyatt Lloyd. 2020. Performance-Optimal Read-Only Transactions. In *USENIX Symposium on Operating Systems Design and Implementation*. USENIX, Virtual Event, 333–349.

[57] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. 2015. Existential Consistency: Measuring and Understanding Consistency at Facebook. In *ACM Symposium on Operating Systems Principles*. ACM, Monterey, CA, 295–310.

[58] Jennifer Lundelius and Nancy Lynch. 1984. An Upper and Lower Bound for Clock Synchronization. *Information and Control* 62, 2 (1984), 190–204.

[59] Nancy A. Lynch. 1996. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

[60] Nancy A. Lynch and Mark R. Tuttle. 1987. Hierarchical Correctness Proofs for Distributed Algorithms. In *ACM Symposium on Principles of Distributed Computing*. ACM, Vancouver, British Columbia, Canada, 137–151.

[61] Jonathan Mace and Rodrigo Fonseca. 2018. Universal Context Propagation for Distributed System Instrumentation. In *ACM European Conference on Computer Systems*. ACM, Porto, Portugal, 1–18.

[62] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. 2011. *Consistency, Availability, and Convergence*. Technical Report TR-11-22. Department of Computer Science, University of Texas at Austin, Austin, TX.

[63] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. 2013. Low-latency Multi-datacenter Databases Using Replicated Commit. *Proceedings of the Very Large Data Bases Endowment* 6, 9 (July 2013), 661–672.

[64] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. 2008. Mencius: Building Efficient Replicated State Machines for WANs. In *USENIX Symposium on Operating Systems Design and Implementation*. USENIX, San Diego, CA, 369–384.

[65] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. 2017. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *USENIX Symposium on Networked Systems Design and Implementation*. USENIX, Boston, MA, 453–468.

[66] Iulian Moraru, David Andersen, and Michael Kaminsky. 2013. There Is More Consensus in Egalitarian Parliaments. In *ACM Symposium on Operating Systems Principles*. ACM, Farmington, PA, 358–372.

[67] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. 2014. Extracting More Concurrency from Distributed Transactions. In *USENIX Symposium on Operating Systems Design and Implementation*. USENIX, Broomfield, CO, 479–494.

[68] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. 2016. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *USENIX Symposium on Operating Systems Design and Implementation*. USENIX, Savannah, GA, 517–532.

[69] Mahsa Najafzadeh, Alexey Gotsman, Hongseok Yang, Carla Ferreira, and Marc Shapiro. 2016. The CISE Tool: Proving Weakly-Consistent Applications Correct. In *Workshop on the Principles and Practice of Consistency for Distributed Data*. ACM, London, United Kingdom, Article 2, 3 pages.

[70] Khiem Ngo, Siddhartha Sen, and Wyatt Lloyd. 2020. Tolerating Slowdowns in Replicated State Machines using Copilots. In *USENIX Symposium on Operating Systems Design and Implementation*. USENIX, Virtual Event, 583–598.

[71] Brian Oki and Barbara Liskov. 1988. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *ACM Symposium on Principles of Distributed Computing*.

ACM, Toronto, Ontario, Canada, 8–17.

[72] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference*. USENIX, Philadelphia, PA, 305–319.

[73] Ruoming Pang, Ramon Caceres, Mike Burrows, Zhifeng Chen, Pratik Dave, Nathan Germer, Alexander Golynski, Kevin Graney, Nina Kang, Lea Kissner, Jeffrey L. Korn, Abhishek Parmar, Christopher D. Richards, and Mengzhi Wang. 2019. Zanzibar: Google's Consistent, Global Authorization System. In *USENIX Annual Technical Conference*. USENIX, Renton, WA, 33–46.

[74] Christos H. Papadimitriou. 1979. The Serializability of Concurrent Database Updates. *Journal of the ACM* 26, 4 (Oct. 1979), 631–653.

[75] Calton Pu and Avraham Leff. 1991. Replica Control in Distributed Systems: As Asynchronous Approach. In *ACM International Conference on Management of Data*. ACM, Denver, CO, 377–386.

[76] Azalea Raad, Marko Doko, Lovro Rožić, Ori Lahav, and Viktor Vafeiadis. 2019. On Library Correctness under Weak Memory Consistency: Specifying and Verifying Concurrent Libraries under Declarative Consistency Models. *Proceedings of the ACM on Programming Languages* 3, POPL, Article 68 (Jan. 2019), 31 pages.

[77] Kun Ren, Dennis Li, and Daniel J. Abadi. 2019. SLOG: Serializable, Low-Latency, Geo-Replicated Transactions. *Proceedings of the Very Large Data Bases Endowment* 12, 11 (July 2019), 1747–1761.

[78] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis, II. 1978. System Level Concurrency Control for Distributed Database Systems. *ACM Transactions on Database Systems* 3, 2 (June 1978), 178–198.

[79] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. 2006. Open Versus Closed: A Cautionary Tale. In *USENIX Symposium on Networked Systems Design and Implementation*. USENIX, San Jose, CA, 239–252.

[80] Cheng Shao, Evelyn Pierce, and Jennifer L. Welch. 2011. Multiwriter Consistency Conditions for Shared Memory Registers. *SIAM Journal on Computing* 40, 1 (2011), 28–62.

[81] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free Replicated Data Types. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*. Springer, Grenoble, France, 386–400.

[82] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary, Laurent Demailly, Thomas Fersch, Liat Atsmon Guz, Andrzej Kotulski, Sachin Kulkarni, Sanjeev Kumar, Harry Li, Jun Li, Evgeniy Makeev, Kowshik Prakasam, Robbert Van Renesse, Sabyasachi Roy, Pratyush Seth, Yee Jiun Song, Benjamin Wester, Kaushik Veeraraghavan, and Peter Xie. 2015. Wormhole: Reliable Pub-Sub to Support Geo-replicated Internet Services. In *USENIX Symposium on Networked Systems Design and Implementation*. USENIX, Oakland, CA, 351–366.

[83] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. 2013. F1: A Distributed SQL Database That Scales. *Proceedings of the Very Large Data Bases Endowment* 6, 11 (Aug. 2013), 1068–1079.

[84] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional Storage for Geo-Replicated Systems. In *ACM Symposium on Operating Systems Principles*. ACM, Cascais, Portugal, 385–400.

[85] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *ACM International Conference on Management of Data*. ACM, Portland, OR, 1493–1509.

[86] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. 1994. Session guarantees for weakly consistent replicated data. In *International Conference on Parallel and Distributed Information Systems*. IEEE, Austin, TX, 140–149.

[87] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. 2013. Consistency-Based Service Level Agreements for Cloud Storage. In *ACM Symposium on Operating Systems Principles*. ACM, Farmington, PA, 309–324.

[88] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. 1995. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *ACM Symposium on Operating Systems Principles*. ACM, Copper Mountain, CO, 172–182.

[89] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2014. Fast Distributed Transactions and Strongly Consistent Replication for OLTP Database Systems. *ACM Transactions on Database Systems* 39, 2 (May 2014), 11:1–11:39.

[90] Paolo Viotti and Marko Vukolić. 2016. Consistency in Non-Transactional Distributed Storage Systems. *ACM Computing Surveys* 49, 1, Article 19 (June 2016), 34 pages.

[91] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. 2018. Carousel: Low-Latency Transaction Processing for Globally-Distributed Data. In *ACM International Conference on Management of Data*. ACM, Houston, TX, 231–243.

[92] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *USENIX Symposium on Operating Systems Design and Implementation*. USENIX, Virtual Event, 191–208.

[93] Haifeng Yu and Amin Vahdat. 2002. Design and Evaluation of a Conit-Based Continuous Consistency Model for Replicated Services. *ACM Transactions on Computer Systems* 20, 3 (Aug. 2002), 239–282.

[94] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building Consistent Transactions with Inconsistent Replication. In *ACM Symposium on Operating Systems Principles*. ACM, Monterey, CA, 263–278.

[95] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2018. Building Consistent Transactions with Inconsistent Replication. *ACM Transactions on Computer Systems* 35, 4, Article 12 (Dec. 2018), 37 pages.

[96] Hanyu Zhao, Quanlu Zhang, Zhi Yang, Ming Wu, and Yafei Dai. 2018. SDPaxos: Building Efficient Semi-Decentralized Geo-Replicated State Machines. In *ACM Symposium on Cloud Computing*. ACM, Carlsbad, CA, 68–81.