Multi-Dispatch Regular Sequential Serializability

Aaron Wu

Advisor: Professor Wyatt Lloyd

Abstract

In this work, we introduce multi-dispatch, a perspective on consistency aimed at capturing correctness conditions in the face of asynchronous client interaction. In doing so, we will introduce examples of how protocols providing classical strong consistency can fail in asynchronous use cases, leading to unexpected behavior. Guided by these examples, we describe the design of RepQueue-RSS, a system that guarantees Helt et al.'s regular sequential serializability augmented with multi-dispatch. As a point of comparison, we describe an API shim for client applications using Spanner-RSS that achieves the same consistency. We find that RepQueue-RSS provides lower end-to-end latency for clients, especially under high levels of concurrency. A proof of correctness for RepQueue-RSS is also provided.

1 Introduction

Services often provide well-defined guarantees for application programmers in the form of consistency models. These range in strength from essentially no guarantees at all – eventual consistency, to a very strong set of guarantees – linearizability and strict serializability. In practice, linearizability and strict serializability are among the strongest consistency models in use. When provided, they allow application programmers to focus on implementing the actual application logic. More specifically, linearizability and strict serializability eliminate the need to handle complicated service interaction scenarios that may arise with weaker consistency models. While the landscape of system designs has grown and evolved significantly in the past few decades, the notion of "strong" consistency has mostly remained fixed to these two models.

Recent work has extended linearizability and strict serializability to better meet the demands of modern client applications. One of these extensions is the multi-dispatch perspective, which addresses a correctness gap in concurrent client invocation scenarios. The other extension has shown that it is possible to weaken strict serializability and linearizability, without increasing complexity from an application programmer's perspective. In particular, the corresponding models, regular sequential serializability and regular sequential consistency provably maintain all application invariants that strict serializability and linearizability provide

[8]. On the other hand, the weaker models allow more flexibility in system design, resulting in decreased tail-latencies.

This work synthesizes the two extensions into the design and evaluation of a multi-dispatch regular sequential serializable (MD-RSS) system. In the following, we will first provide a detailed description of multi-dispatch and regular sequential serializability. Then, we introduce examples further motivating the significance of multi-dispatch. The last of these examples will segue directly into our main contributions:

- We show how client applications can modify their usage of the Spanner/Spanner-RSS API to ensure multi-dispatch
- We introduce the design and prove the correctness of RepQueue-RSS, a system specifically designed to achieve multi-dispatch regular sequential serializability
- Evaluating RepQueue-RSS against this baseline in the single data center setting, we find that our system achieves significantly lower client-perceived end-to-end latency under highly concurrent client usage.

2 Background

2.1 Multi-Dispatch

Traditional consistency models, such as linearizability and strict serializability, center their definitions around synchronous client-service interaction. That is, each client has at most one outstanding invocation at any given time. However, this is becoming an increasingly unrealistic model to impose upon client applications.

Starting from the early 2000s, the turn towards multicore processor architecture has pushed software to adopt concurrency [13]. We can see this trend reflected in widely-used modern application programming languages such as Go, Java, and C++11, which have extensive facilities for concurrent programming either built-in or as a part of their standard library. Structuring programs to divide work across multiple threads can result in great performance gains but can also easily lead to asynchronous service usage. In the same way one tries to avoid lock contention amongst threads, it is natural to allow multiple threads to issue service requests independently of one another. This can lead to situations where a client has more than one outstanding request. Note that multi-threading is not a necessary pre-condition for asynchronous clients. For example, by using non-blocking system calls, libraries such as ASIO for C++ enable single-threaded applications to issue multiple requests without waiting for a network response.

When interacting with services guaranteeing traditional consistency models, asynchronous clients must

choose between bridging with synchronous APIs or simply accepting undefined behavior between asynchronous invocations. The former can be prohibitively expensive for some applications. For example, an application for fetching a particular user's news-feed might rely on concurrent queries to reduce page load times. The latter is generally not a good idea (in the same vein as indexing out of bounds). For instance, consider the concurrent interaction with a linearizable service shown in figure 1. Client 1 asynchronously invokes writes $x = 1$ then $y = 2$. Simultaneously, client 2 issues reads of y then x. However, the results of the reads are $x = \perp$ and $y = 2$, respectively.

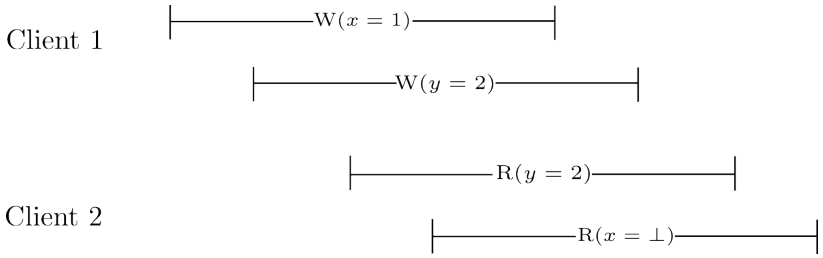

Figure 1: A legal execution under traditional linearizability

The example runs against what one would expect. The service sees the write to x before the write to y, yet the other client can observe the effects of the write to y without seeing the write to x. This motivates our definition of multi-dispatch as an augmentation of traditional consistency models. We say an execution satisfies *multi-dispatch* if for two requests π_1, π_2 such that some client invokes π_1 before π_2, the any results or effects from π_1 must precede that of π_2. We will formalize this nebulous definition in section 2.3, where we specialize to the cases of multi-dispatch regular sequential serializability.

We remark that multi-dispatch inherently captures a notion of causality through invocation ordering. Thus, it only makes sense to consider underlying traditional consistency models that are at least as strong as casual consistency. Otherwise, the resulting guarantees would be rather odd. For example, multi-dispatch over eventual consistency would give stronger guarantees for asynchronous interaction than normal, synchronous interaction.

2.2 Regular Sequential Serializability

The key observation of regular sequential serializability is that casually unrelated reads should not be subject to real-time constraints. To expand upon this, consider an ongoing write to keys x, and y. Client 1 reads these keys and sees the effect of the ongoing writes. After client 1's reads have completed but before the write finishes, client 2 then reads the same keys. By real-time ordering, client 2 must read the same values that client 1 observed. Intuitively, this ordering is unnecessary; assuming no communication between clients 1 and 2, the read of client 1 is casually unrelated to that of client 2. Moreover, the write to keys x, y has

not completed, so it would not be unexpected for client 2's read to return the value of the previous commit.

Regular sequential serializability relaxes the read-time constraint of strict serializability to allow more flexible executions of casually unrelated reads. It turns out that regular sequential serializability is still strong enough to maintain application invariants the client observes under strict serializability. We now state an abbreviated definition of regular sequential serializability (RSS) closely following the presentation of the original paper [8].

2.2.1 Computation Model Preliminaries

The semantics of a service's behavior in the absence of concurrency can be captured by a set of invocation-response pair sequences $\mathfrak{G}$. We call this the *sequential specification* of a service. In the case of transactional databases, the sequential specification is the set of sequences such that any reads of k must observe the effects of the most recent transaction which writes to k. Each sequence S in a sequential specification has a natural order, which we call $<_S$

Definition 2.1: An *execution*, σ, is a sequence of *actions* from a set of clients $j \in \{1, 2, ...\}$. The invocation and response actions represent service interaction. An invocation and its corresponding response constitute an *operation*. There are two types of operations that we consider: read-write transactions and read-only transactions.

Definition 2.2: An execution σ is *well-formed* a client has at most one outstanding invocation and all responses are preceded by their corresponding invocations.

Definition 2.3: The *local history* of client j, denoted $\sigma|j$ consists of the subsequence of actions specific to client j. We say two well-formed executions σ and η are *equivalent* if $\sigma|j = \eta|j$ for all clients j. Denote $complete(\sigma)$ as the maximal subsequence of σ such that all invocations have a matching response.

Definition 2.4: For $\pi_1, \pi_2 \in \sigma$, we say that π_1 precedes π_2 in *real time*, $\pi_1 \rightarrow_\sigma \pi_2$, if π_1 is a response, π_2 is an invocation, and π_1 appears in σ before π_2.

Definition 2.5: For any two π_1, π_2 in a well-formed execution σ, we say π_1 precedes π_2 in *casual order*, denoted $\pi_1 \rightsquigarrow_\sigma \pi_2$, if any of the following conditions hold

1. π_1 appears before π_2 in any client's local history $\sigma|j$
2. π_1 is the response of some operation o_1, π_2 is the invocation of operation o_2, and o_2 reads a value

written by o_1

3. There exists π_3 such that $\pi_1 \rightsquigarrow_\sigma \pi_3$ and $\pi_3 \rightsquigarrow_\sigma \pi_2$.

Definition 2.6: We say that a transaction T', we say that T *conflicts* with T' if either of the following two conditions hold

1. T is a read-only transaction that reads keys written to by T'
2. T is a read-write transaction

We will denote the set of transactions that conflict with T as $C(T)$.

2.2.2 Regular Sequential Serializability Definition

We say that an well-formed execution σ satisfies *regular sequential serializability* (RSS) if it can be extended to σ_1 by adding zero or more responses such that there exists a sequence $S \in \mathfrak{G}$ satisfying

1. S is equivalent to $complete(\sigma_1)$
2. For all pairs of invocations and responses π_1, π_2, if $\pi_1 \rightsquigarrow_{complete(\sigma_1)} \pi_2$, then $\pi_1 <_S \pi_2$
3. For all responses π_1 corresponding to a read-write transaction T and invocation actions π_2 corresponding to $T' \in C(T)$, if $\pi_1 \rightarrow_{complete(\sigma_1)} \pi_2$, then $\pi_1 <_S \pi_2$.

2.3 Multi-Dispatch Regular Sequential Serializability

2.3.1 Multi-Dispatch Preliminaries

To augment RSS with multi-dispatch, we make modifications to the preliminary definitions. Since multi-dispatch clients can have multiple outstanding requests, we must relax the conditions of 2.2 and make appropriate adjustments in other definitions.

Definition 2.7: We say an execution σ is *asynchronously well-formed* if all responses are preceded by their corresponding invocation.

Definition 2.8: For an asynchronously well-formed execution σ and client j, let the *local invocation sequence* for j, denoted by $(\sigma|j)_{\text{seq}}$ be the well-formed synchronous execution derived from the invocation order of $\sigma|j$. We construct $(\sigma|j)_{\text{seq}}$ from $\sigma|j$ as follows

- (Preservation of invocation order) For any given invocation π_1, if a response π_2 exists, then π_2 is placed immediately after π_1.

Then, we say an asynchronously well-formed execution σ and synchronously well-formed execution η are equivalent if $(\sigma|j)_{\text{seq}} = \eta|j$. The definition of $complete(\sigma)$ remains as before

Definition 2.9: For any two π_1, π_2 in a well-formed asynchronous execution σ, we say π_1 precedes π_2 in *asynchronous casual order*, denoted $\pi_1 \rightsquigarrow^a_\sigma \pi_2$, if any of the following conditions hold

1. If π_2 is a response, and π_1 is its corresponding invocation in a client's local history $\sigma|j$.
2. If π_2 is an invocation, and π_1 closest preceding response in a client's local history $\sigma|j$.
3. π_1 is the response of some operation o_1, π_2 is the invocation of operation o_2, and o_2 reads a value written by o_1
4. There exists π_3 such that $\pi_1 \rightsquigarrow^a_\sigma \pi_3$ and $\pi_3 \rightsquigarrow^a_\sigma \pi_2$.

Here, the additional conditions 1-3 aim to capture the original notion of causality, but accounts for multiple outstanding invocations.

2.3.2 Multi-Dispatch Regular Sequential Serializability Definition

With these modified definitions, the multi-dispatch definition looks much the same as the original RSS case. We say that an asynchronously well-formed execution σ satisfies *multi-dispatch regular sequential serializability* (MD-RSS) if it can be extended to σ_1 by adding zero or more responses such that there exists a sequence $S \in \mathfrak{G}$ where

1. S is equivalent (in the sense of definition 2.8) with $complete(\sigma_1)$
2. For all pairs of invocations and responses π_1, π_2, if $\pi_1 \rightsquigarrow^a_{complete(\sigma_1)} \pi_2$, then $\pi_1 <_S \pi_2$
3. For all responses π_1 corresponding to a read-write transaction T and invocation actions π_2 corresponding to $T' \in C(T)$, if $\pi_1 \rightarrow_{complete(\sigma_1)} \pi_2$, then $\pi_1 <_S \pi_2$.

The most important point here is the different notion of equality in condition 1. Enforcing that the global serialization respect the invocation order of each client formally captures our intuition of per-client temporal precedence. The same modified definitions can be used to define multi-dispatch strict serializability. Since there is a substantial overlap with the above, we choose to omit the full, formal definition.

3 Motivating Examples

To show the utility of multi-dispatch as a distinct concept, we next provide examples of existing protocols with strong consistency guarantees and discuss their behavior in concurrent usage scenarios.

3.1 Raft and Zookeeper

Ongaro and Ousterhout's Raft protocol is a consensus algorithm based around the notion of a replicated append-only log [12]. In the paper, the authors also describe how one can build a linearizable key-value storage system, using the Raft log to dictate state changes. However, Raft does not guarantee multi-dispatch linearizability, and the storage system as described in the paper also cannot make this guarantee. In fact, nothing in the protocol itself prescribes constraints on the ordering of concurrent requests. Thus, the execution illustrated in Figure 2 is admissible.

In this run, a cluster of three servers, A, B, and C, begins with leader A. A client sends a sequence of two state changes $\langle 1\rangle, \langle 2\rangle$ to A asynchronously. A commits $\langle 1\rangle$ but sudden network congestion temporarily partitions it from B, C, and the client. The majority partition quickly elects a new leader, say C. Then, the client asynchronously sends another request $\langle 3\rangle$ to C, which promptly replicates and commits the entry. At this point, C temporarily fails, the network partition heals, and A is re-elected the leader. It receives the delayed request $\langle 2\rangle$, which it subsequently replicates and commits. The committed log reads $\{\langle 1\rangle, \langle 3\rangle, \langle 2\rangle\}$, yet the invocation sequence was $\{\langle 1\rangle, \langle 2\rangle, \langle 3\rangle\}$.

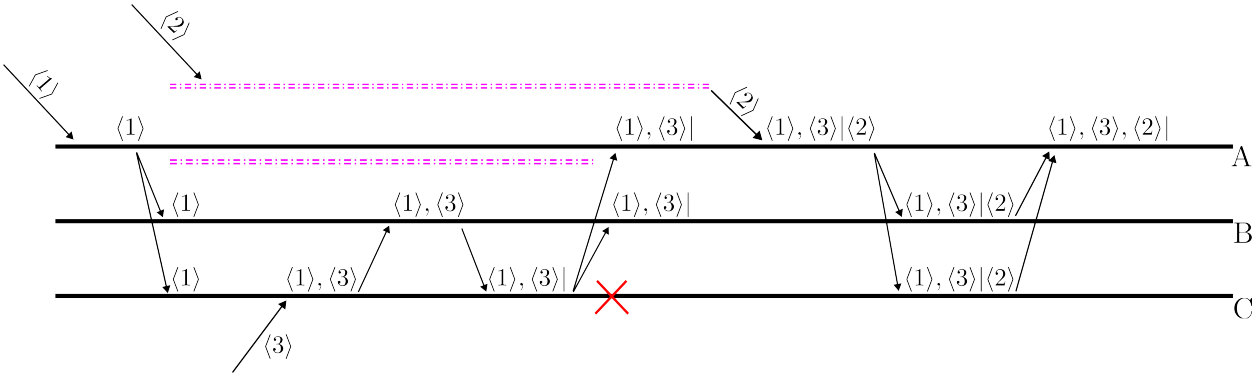

Figure 2: A Raft run which violates multi-dispatch linearizability. Temporal progression is from left to right. Entries to the left of the vertical bar are committed. The dotted magenta lines represent a transient network partition, and the red cross indicates a temporary node outage.

We found that several popular open-source implementations do not address this kind of violation and are also agnostic to the order in which concurrent requests are serviced. In particular, the implementations of etcd [1], HashiCorp [2], and TiKV [3], all admit the execution described above. For each, there are two major practical details which can lead to invocation order violations. First, cluster leaders have no information about the ordering on requests and unconditionally append to their log. This is exactly the issue that leads to the example execution. Second, incoming RPC requests are serviced in arbitrary order.[1] Even if all

[1]HashiCorp's RPC implementation spins off a new goroutine for every new request. TiKV and etcd both rely on gRPC, which also does something similar. Using RPC streams could solve reordering caused by this issue, but the first problem

client requests arrive at the leader without network reordering, there is no guarantee that those requests are processed in the same order.

Closely related to Raft is the Zookeeper Atomic Broadcast protocol (ZAB), which forms the core of Yahoo!'s Zookeeper distributed configuration management service. It features a strong leader and log consistency enforcement, but lacks an internal mechanism for detecting leader failure. Of note is how the Zookeeper system as a whole builds substantially upon the core protocol to guarantee what its authors call asynchronous linearizability (A-linearizability) [9, 10].

A-linearizability is an analogue of the multi-dispatch perspective specific to linearizability. The exact mechanism through which this guarantee is achieved is not made explicit in the paper nor documentation. However, this is likely accomplished through its *client session* construct. In this scheme, all asynchronous requests in a given client session would be assigned a number, and the ZAB leader would track the number it expects next. To ensure correctness across leader failures, these numbers would be replicated across the cluster. This sequence number based approach resembles the TCP protocol for enforcing in-order delivery, and as we will see later, it features prominently in our RepQueue-RSS protocol.

3.2 Zelos

Recent work from Meta (formerly Facebook), has also touched upon topic [4]. Building off of the Delos system, the authors propose the abstraction of a *log-structured protocol* as a primitive for storage system development. Each log-structured protocol comprises of an engine, which communicates with its counterpart engines on other hosts through reading and appending to the shared log. Instead of monolithic codebases, storage systems can then be expressed as a stack of modular and reusable log-structured protocols.

One of the storage systems that was implemented using this framework was Zelos, a Zookeeper clone providing the same asynchronous linearizability guarantee. To achieve this, an additional SessionOrderEngine was layered on top of an existing database stack. As in the discussion of Zookeeper, the SessionOrderEngine tags each request with a sequence number. On reordering, the engine simply throws away the reordered entry and send a retry back down the stack.

3.3 Spanner and Spanner-RSS

The preceding examples concern linearizability in the single cluster case. In the context of data stores, it is highly pertinent to consider transactional protocols that also incorporate sharding. Sharding enables horizontal scalability, which is a key priority for modern distributed applications, while transactional models

remains.

expose a more expressive interface to application programs.

Similar to our discussion of Raft, Spanner guarantees Strict-Serializability but admits executions that violate our multi-dispatch constraint. Figure 3 illustrates such an execution. Note that the same execution is valid for Spanner-RSS, as the set of Spanner-RSS executions is a strict superset of Spanner executions.

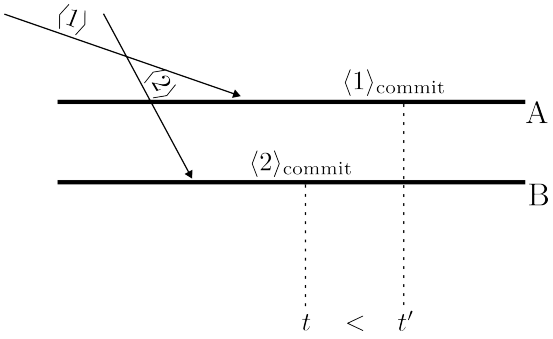

Figure 3: An example Spanner run that violates multi-dispatch strict-serializability and multi-dispatch RSS

Here, we only need consider two shards A and B. For clarity, we omit the depiction of the shard replicas. A client sends read-write transactions $\langle 1 \rangle, \langle 2 \rangle$, with $\langle 1 \rangle$ only affecting keys on shard A and $\langle 2 \rangle$ affecting keys on shard B. Then, $\langle 2 \rangle$ first arrives at shard B and is committed with some timestamp t. At some later time, $\langle 1 \rangle$ arrives at A and is committed at time $t' > t$. The invocation order is $\langle 2 \rangle, \langle 1 \rangle$, yet the commit order is the reverse. Again, we see how the lack of ordering information the coordinators receive and the inability of the client to control the timing of the state change cause ordering violations.

4 Multi-Dispatch Spanner/Spanner-RSS Client

As we have seen, asynchronous client interaction with Spanner-RSS directly through the API presented in the original paper can lead to multi-dispatch violations. A thin client library can be used to rectify this problem. Practically, this approach has the benefit of being easily integrated into existing codebases. Furthermore, our goal in this work is not to modify Spanner itself; rather, we aim to examine how a system with first class multi-dispatch support compares against modified client usage of an existing system. We give an overview of this library through a case-by-case analysis.

To enforce multi-dispatch between read-write transactions, we must ensure that commit times obey invocation order. Unfortunately, there is no facility to specify *read-writes* at a timestamp, so this must be done through the use of per-client sequence numbers stored in the database itself. Every the client tags each of its outgoing read-write transactions with a sequence number. This constitutes explicit invocation order information. The bodies of these read-write transactions are wrapped in an if-statement; this conditional

checks whether the sequence number stored in the database matches that of the transaction. If it is equal, the transaction is next in invocation order and can proceed. After the end of the original transaction body is executed, the client-specific sequence number is incremented in the database. If it is not equal, the transaction is *not* next in invocation order and must abort to be retried at a later point in time.

Modified Read-Write Transaction

```
1 curr_seq_no := GET(client_id)
2 if my_seq_no = curr_seq_no then
3 |   Transaction Body
4 |           ⋮
5 |   PUT(client_id, curr_seq_no + 1)
6 else
7 |   RETRY
8 end
```

For read-only transactions, we assume that clients either have access to the TrueTime service or have a reasonably small, bounded clock skew. In the case that the TrueTime service is unavailable to the client, most modern operating systems can be configured to poll a time server regularly, so we believe this is a realistic assumption. Under this assumption, it suffices construct monotonically increasing timestamps that (1) are greater than or equal to most recent read-write transaction (2) true current time. We can then leverage the read-at-timestamp RPC with these timestamps to guarantee correctness.

Note that for (1), read-only transactions must block until Spanner sends a response to the most recent read-write, which contains the commit timestamp. In the absence of any read-write transactions from the client, meeting constraint (2) suffices.

The construction of these timestamps is as follows. To meet constraint (2), we take the max of *current_time.latest* and the previous read-only timestamp. This is necessary, as clock synchronizations can lower the upper bound on clock skew i.e. *current_time.latest* can go backwards. Then if there any read-write transactions, we take the max of the result and timestamp of the most recent commit time to meet constraint (1).

Procedure ReadOnlyTransaction

```
1 read_ts := max{curr_time.latest(), prev_read_ts}
2 if sent a read-write transaction then
3 |   wait until latest read-write commits with timestamp t
4 |   read_ts := max{t, read_ts}
5 |   Send(ReadAtTimestamp, {read_ts})
6 else
7 |   Send(ReadAtTimestamp, {read_ts})
8 end
```

To ensure that read-only transactions do not execute after a subsequent read-write transaction, every

read-write following a read-only transaction must wait until its associated *current_time.latest* has passed.

Observe that the library exhibits anti-patterns of Spanner usage, especially with regards to read-write transactions. All asynchronous read-writes from a single client will experience high contention due to the read and subsequent write of the client-specific sequence number. At best, this contention serializes read-write transactions. At worst, every concurrent read-write will cause each other to partially complete their transaction, abort due to failed lock acquisition, and then wound wait before retrying, resulting in wasted system capacity.[2]

5 RepQueue-RSS

RepQueue-RSS is a distributed transactional storage system purpose built to guarantee Multi-Dispatch RSS. Like Spanner, it supports read-write and read-only transactions, and optimizes for read-heavy workloads.

The apparent issue with the multi-dispatch Spanner client is the tracking of client's sequence as a key in the store itself and the resulting contention. We avoid these shortcomings by avoiding contention through deterministic ordering and embedding ordering logic into the protocol itself.

Introduced by Calvin, the deterministic approach to concurrency control fixes an order of transactions through a sequencing layer before execution [15]. This elides the need for locking or optimistic concurrency control, and also allows explicit tracking and enforcement of invocation order at points of potential reordering i.e. receiving and sending RPCs over the asynchronous network. As we will see in §5.4, an integral part of the read-only transaction protocol are the queues $Q_i|j$ on each manager node, hence the name **Rep**licated **Queue**-RSS. Proof of the core protocol's correctness can be found in section §7.

5.1 System Assumptions and Design Overview

All transactions are handled by a transaction manager, which is a collection of servers ordered to form a chain. This manager communicates with shard groups to durably store data. Each shard group is a multi-versioned, linearizable, storage service; we elect to use Raft for data replication, but any other system that meets these requirements would also be fine. Only read-write transactions are replicated in the manager chain, and clients can open a session with any manager server except for the tail. Read-only transactions pass through a transaction manager node before being directly forwarded to the appropriate shard groups for execution. Figure 4 illustrates the architecture of our system and figure 5 outlines protocol data structures.

[2]According to the most recent documentation on Cloud Spanner, SQL queries support a LOCK_SCANNED_RANGES hint, allowing transactions to acquire an exclusive lock on read keys. This mitigates the possibility of the described worst-case scenario.

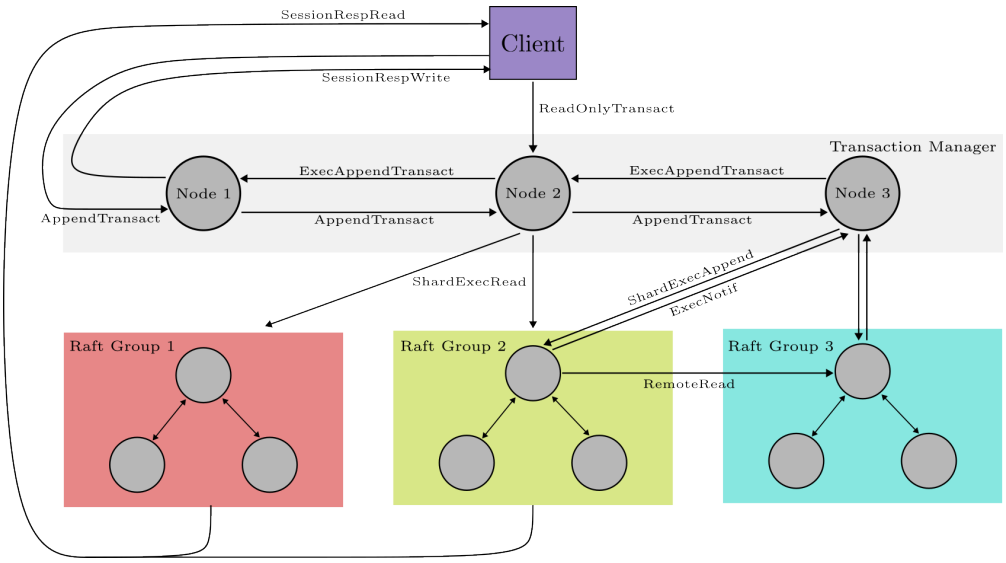

Figure 4: Overview of RepQueue-RSS architecture and RPCs.

In the subsequent, assume the transaction manager consists of n servers and data is split among m shard groups. We uniquely identify each node in the chain with an index $i \in [1, n]$ based on their location. That is, the head of chain is 1; the tail is n; and the successor of any other node i in the chain is $i + 1$. We also label each shard group with some unique $j \in [1, m]$.

Read-write transactions start as in-progress (INPROG) at the head of the chain and are replicated at successive nodes. Once the tail appends a transaction to its log, the transaction is considered committed, and its constituent sub-transactions sent to the relevant shard groups for execution. Completion of sub-transactions are sent to the chain tail. After the tail learns that all of these are complete, a command to transition the transaction to SUCCESS status is sent backwards through the chain. A transaction whose status changes to SUCCESS on the head node subsequently has its completion returned to the client.

Like Chain Replication, we assume the transaction manager servers are fail-stop. We also assume the presence of a failure detector for the chain and a coordination service, which maintains the mapping from shards to keyspaces. We will refer to this mapping as $sh(\cdot)$. Our network model is asynchronous: messages can be dropped, delivered out of order, delivered with unbounded delay, or all three.

As a notational convention, we will herein take take superscript "write" to denote the write key-set and superscript "read" to denote the read key-set of a transaction. For maps, superscript "key" represents the domain of the map and "value" to be its range.

On **transaction manager node** i

1. (RW) Write-ahead log of read-write transactions, $\mathcal{L}_i$: All transactions on the log have a sequence number, which doubles as an index into the log. Log entries contain both the transaction and associated CWSN.
2. FIFO queue for each shard group j, $Q_i|j$: Each queue j is maintained so that its contents and order are exactly the log sequence numbers of in-progress transactions affecting keys on shard group j. Additionally, each queue tracks the log sequence number of the most recently dequeued entry, which we will write as $exec(Q_i|j)$.
3. (RW) Map from in-progress transactions to shards, C_i: Any given transaction $T \in \mathcal{L}$ with state IN-PROG is viewed as a collection of disjoint sub-transactions $\{T|j_p\}_{p=1}^{k}$, where $k \leq m$ and $T|j_p$ is the subset of operations in T that only concern the state on shard j_p. This map stores bindings of the form $T.sn \mapsto (shds, cwsn)$, where $shds := \{j_1, ..., j_k\}$ and $cwsn$ is the associated CWSN. For a given in-progress transaction, $shds$ informs us which participating shard groups have not yet executed the necessary operations, and the $cwsn$ enables the update of transaction status in item 5.
4. (RW) Map from shards to sequence number, SSN_i: In the same way that the client library appends each outgoing transaction with either a CWSN or CRSN, the manager nodes increment SSN on every insert into $Q_i|j$.
5. (RW) Ordered Map from CWSNs to transaction metadata, $Ongoing_i$: This associates a CWSN with its corresponding meta or response that has not yet been acknowledged. Values take on the form $(status, data, logIndex)$. The additional per-client bookkeeping is essential to guaranteeing exactly-once semantics in the presence of server failures and message loss.
6. (RW) Map from cids to max observed CWSN sequence number, W_i: This is used to prevent invocation reordering on chain replication.
7. (RO) Map from cids to $(maxNum, lsn)$ tuple, R_i: The component $maxNum$ is the highest CSRN num seen from client cid, and lsn is a log sequence number. As we will see in §5.4, this other form of per-client metadata is used for preserving the invocation-order for reads when messages arrive out of order.

On **shard group** j

1. Log sequence number of the most recently completed sub-transaction, $shExec_j$: This is analogous to the queues and $exec(\cdot)$ in the transaction manager nodes.
2. Local sequence number, ssn_j: After the completion of any sub-transaction $T|j$ that involves a state mutation, the shard group increments ssn_j.

On **client** cid

1. Next CWSN and CRSN in sequence, $cwsn_{\text{curr}}$ and $crsn_{\text{curr}}$: These are attached to client transactions and incremented accordingly.
2. Ordered set of CWSNs assigned to outstanding transactions, $InProg_{cid}$: This allows computation of the largest CWSN such that all transactions with lower CWSNs have received responses. Set elements are removed on receiving responses.
3. (RO) Ordered map from CRSNs to log sequence numbers, UB_{cid}: This is a set of upper bounds that ensure the results of a retried read-only transaction do not violate invocation ordering. As above, the client library prunes this map appropriately as it receives responses from the service.
4. (RO) Map from CRSNs to response data, $ReadResult_{cid}$: This map is of the form $\{crsn_T \mapsto (num, data, keys, lsn)\}$, where num is the number of participating shards, $data$ contains response data, $keys$ are the keys to be read, and lsn is the read's log sequence number constraint.

Figure 5: Data structures used by RepQueue-RSS servers. RW denotes structures specific to read-write transactions §5.3, and RO denotes those specific to read-only transactions §5.4.

For the rest of this section, we first detail the protocol for simple read-write and read-only transactions. Then, we describe how read-only and dependent read-write transactions are reduced to the case of simple read-write transactions. Finally, we touch on extensions to the core protocol, such as metadata garbage collection and optimizations for the wide-area.

5.2 Client-Service Interaction

Clients establish a session using a local library, through which all interactions with the service occur. The client-side library transparently maintains this session without the need for user intervention. Clients may have multiple outstanding transaction requests and can expect that the resulting execution preserves invocation order. In the case of connection loss or node failures, the library will automatically attempt to reconnect and retry requests in a multi-dispatch safe manner. We discuss this safe retry logic in sections 5.3 and 5.4.

Read-only transactions are automatically assigned a sequence number respecting real-time invocation order, which we call a CRSN (client read-only sequence number). Similarly, read-write transactions and write-only transactions are stamped with a CWSN (client write sequence number). Both CRSNs and CWSNs take the form (cid, num), where cid is a unique client identifier and $num \in \mathbb{N}$ is a logical timestamp representing invocation order. The ordering on CRSNs and CWSNs is derived from the ordering on $\mathbb{N}$. Concretely, we have $(cid, 0) < (cid, 1) < (cid, 2) < ....$

5.3 Simple Read-Write Transactions

Since our system expresses transactions as a stored procedure on a log, we must distinguish between two types or read-write transactions: *simple* read-write transactions and *dependent* read-write transactions. Simple read-write transactions do not rely on the result of any constituent operation to determine the full read/write key-set. For example, there are no dependencies amongst the operations in $T := \{w(x) = 5, r(y), r(z)\}$, so this is of simple type. Another example is a conditional write such as $T := \{\text{if } r(z) \geq 100 \text{ then } w(x) = x - 100\}$. While the value written to x depends on z, all keys affected by the transaction are known statically. On the other hand, dependent read-write transactions require the results of one or more constituent operations to determine the full read/write key-set. A common scenario where this can arise is secondary index lookups. We first focus on specifying our protocol in the simple read-write case. We will extend this to cover the dependent case after introducing the protocol for read-only transactions.

To submit a simple read-write transaction to the system, the client invokes SessionRuntimeAppend through a wrapper, which handles triggering retry timeouts. The procedure tags the transaction with the current CWSN, increments the client CWSN, and then sends the transaction to the head of the transaction

manager chain.

Procedure SessionRuntimeAppend(T, $cwsn_T$ | None)

```
   // SessionRuntimeAppend on client cid
1  if cwsn_T = None then
2  |  cwsn_T := cwsn_curr
3  |  cwsn_curr := cwsn_curr + 1
4  ackBound := min(InProg_cid)
5  Send_head(AppendTransact, {T, cwsn_T, None, ackBound}) // all writes go to head of chain
6  return cwsn_T
```

Note the arguments $cwsn_T$ and $ackBound$ in the initial AppendTransact RPC to the head. The first of the two prevents the transaction manager from retrying an already committed transaction. Any retry must provide the $cwsn_T$ associated to that transaction. The second enables garbage collection of no-longer-needed metadata.

The replication of the transaction through the chain is straightforward. At each node, each request blocks until both its CWSN and log index match the next expected value. This ensures preservation of invocation order as well as consistency of the log across transaction manager nodes. After satisfying these constraints, data structures defined in figure 5 are updated accordingly. Note that the head can ignore the log index constraint, as it determines the log index itself. Thus, the client passes a placeholder value of None for that argument. Once the replication is finished at the tail, the transaction is decomposed into sub-transactions corresponding to each shard's keyspace and sent for execution.

In order to support more expressive types of transactions, such as conditional writes, the tail must also compute dependencies between shards. These dependencies are also forwarded along with the sub-transactions, and are resolved dynamically by the shards. For example, recall the transaction $T := \{\text{if } r(z) \geq 100 \text{ then } w(x) = x - 100\}$. Suppose that key z and x reside on different shards; then, the write of x takes a remote dependency on z. Using the additional dependency data from the tail, the shard responsible for z shard forwards the read value to the shard responsible for x. Transaction manager driven retries are piggybacked on subsequent invocations or sent after a preset timeout, whichever happens first.

Procedure AppendTransact(T, $cwsn_T$, $logInd$, $ackBound$)

```
   // AppendTransact on transaction manager node i ∈ [1, n]
 1 if cwsn_T.num ≤ W_i[cwsn_T.cid] then
 2 |   if i = 1 ∧ Ongoing_i[cwsn_T] = resp(SUCCESS, data, logIndex) then
 3 |   |   Send_cid(SessionRespWrite, {cwsn_T, resp})// immediately return committed result
 4 |   return
 5 else
   |   // prevent reordering
 6 |   if i = 1 then
 7 |   |   wait until W_i[cwsn_T.cid] + 1 = cwsn_T.num
 8 |   else
 9 |   |   wait until W_i[cwsn_T.cid] + 1 = cwsn_T.num ∧ ind = L_i.len
10 |   end
11 end
12 logIndex := L_i.len
13 L_i ← (T, cwsn_T)
14 W_i[cwsn_T.cid] := W_i[cwsn_T.cid] + 1
15 Ongoing_i := Ongoing_i ∪ {cwsn_T ↦ (INPROG, {}, logIndex)}
16 Ongoing_i := Ongoing_i \ {(k ↦ W_i[k]) ∈ W_i : k < ackBound} // remove unneeded metadata
17 C_i := C_i ∪ {logIndex ↦ (cwsn_T, {})}
18 foreach j ∈ [1, m] do
19 |   particip := T^write ∩ sh(j)
20 |   if particip ≠ ∅ then
21 |   |   Q_i|j ← logIndex
22 |   |   SSN[j] := SSN[j] + 1
23 |   |   C_i[logIndex].shds := C_i[logIndex].shds ∪ {j}
24 end
25 if i = n then
26 |   ShDeps := DepAnalysis(T^key) // local and remote dependencies for each participant j
27 |   foreach j ∈ C_i[logIndex] do
28 |   |   Send_j(ShardExecAppend, {T|j, logIndex, SSN[j], shDeps[j]})
29 |   end
30 else
31 |   Send_succ(AppendTransact, {T, cwsn_T, logIndex, ackBound})
32 end
33 return
```

Just as before, upon receiving a sub-transaction from the tail, the request must wait until its sequence number matches that of the shard group. This guarantees that any processed sub-transaction has sequence number less than or equal to the current.

An important detail that is <u>not</u> shown in the ShardExecAppend procedure below is the case of retries. More specifically, there is a danger that a retried request with a lower sequence number is applied on top one with a higher sequence number. Generally, the way this is handled is protocol specific. In our case of Raft, we replicate sequence numbers alongside their requests. When applying entries to the state machine, we simply compare the requests sequence number against the current. If it is greater, then we apply the changes. If it is less than or equal, we know the request was a retry and ignore its operations.

Procedure ShardExecAppend($T|j$, ind, sn, $deps$)

```
   // ShardExec on shard group j ∈ [1, m]
 1 if sn ≤ ssn_j then
 2 |  return
 3 else
 4 |  wait until ssn_j + 1 = sn // handle manager to shard message reordering
 5 end
 6 recv all deps^local
   // Once dependencies resolved, can execute local operations and serve reads
 7 respData := ∅
 8 foreach op ∈ T|j do
 9 |  match op with:
10 |  |  put(key, value) ⟹ put(key, value, ind) // versioned put exposed by storage service
11 |  |  get(key) ⟹ respData := respData ∪ {key ↦ get(key, ind)}
12 |  end
13 end
14 foreach op ∈ deps^remote do
15 |  Send_{op.shard}(RemoteRead, {op.key ↦ get(key)})
16 end
17 shExec_j := ind
18 ssn_j := sn
19 Send_tail(ExecNotif, {j, ind, respData})
20 return
```

The tail node listens to shard group replies through the ExecNotif RPC and updates C_n. After all sub-transactions return, the tail propagates transaction completion backwards through the chain using ExecAppendTransact. For each participating shard queue, indices are dequeued up to and including the completed transaction index. Since results are only returned to the client once the backwards pass has reached the head, this ensures the manager node always has a non-stale view of all transaction statuses.

An interesting observation here is that the actual order of transaction completion (not committment) may not reflect the original invocation ordering, while the logical ordering of transactions based on log committment will remain consistent.

Procedure ExecNotif(j, ind, $respData$)

```
   // ExecNotif on tail node n
 1 C_n[ind].shds := C_n[ind].shds \ {j}
 2 cwsn := C_n[ind].cwsn
 3 W_n[cwsn].data := W_n[cwsn].data ∪ respData
 4 if C_n[ind].shds = ∅ then
 5 |  ExecAppendTransact({ind, W_n[cwsn].data})
 6 return
```

Procedure ExecAppendTransact(ind, $data$)

```
   // ExecAppendTransact on node i ∈ [1, n]
 1 cwsn := C_i[ind].cwsn
 2 foreach j ∈ C_i[ind].shds do
 3 |   while exec(Q_i|j) < ind do
 4 |   |   exec(Q_i|j) ← Q_i|j
 5 |   end
 6 end
 7 C_i := C_i \ {ind ↦ C_i[ind]}
 8 W_i[cwsn].status := SUCCESS
 9 W_i[cwsn].data := data
10 if i = 1 then
11 |   Send_{cwsn.cid}(SessionRespWrite, {cwsn, W_i[cwsn]})
12 else
13 |   Send_{pred}(ExecAppendTransact, {ind, data})
14 end
15 return
```

Procedure SessionRespWrite($cwsn$, $resp$)

```
   // SessionRespWrite on client cid
 1 InProg_cid := InProg_cid \ {cwsn}
 2 return resp
```

5.4 Read-Only Transactions

Clients invoke SessionRuntimeRead for every read-only transaction. Retries are handled separately in SessionRuntimeReadRetry. As with read-write transactions, the runtime tags each read-only transaction with a CRSN. For read-only transactions, clients must track two additional types of metadata: read-write dependencies and a set of freshness upper bounds for retries.

Procedure SessionRuntimeRead(T)

```
   // SessionRuntimeRead on client cid, connected to manager node i
 1 writeDep := None
 2 if InProg_cid ≠ ∅ then
 3 |   writeDep := max(InProg_cid) // read-write dependency
 4 crsn_T := crsn_curr
 5 crsn_curr := crsn_curr + 1
 6 ReadResult_cid := ReadResult_cid ∪ {crsn_T ↦ (None, {}, T, lsnConst)}
 7 UB_cid := UB_cid ∪ {crsn_T ↦ None} // create entry in map of upper bounds
 8 Send_i(ReadOnlyTransact, {T, crsn_T, writeDep, None})
 9 return crsn_T
```

Recall that all read-write transactions must travel through the chain before being sent to the shards. In contrast, we impose no such restriction upon read-only transactions; read-only transactions must only pass through a singular transanction manager node. Without any additional constraints, this can lead to situations where a client's asynchronous read-only transaction does not see the effects of the same client's casually

preceding (in invocation order) read-write transaction. Thus, clients also tag each read-only transaction with the CWSN of the most recent read-write transaction for which it has not heard a response, *writeDep*.

For retries, the client session must also explicitly provide an upper bound on freshness, $lsnConst$, to the transaction manager node. Otherwise, a retry of an earlier read-only transaction could return information newer than that of later read-only transactions, with respect invocation order. Consider the set of completed read-only transactions with CRSN greater than $crsn_T$. We take $lsnConst$ as the log sequence number contained in the response to the minimal element of this set. That is, $lsnConst$ can be though of as the least upper bound on freshness. In practice, we do not track all completed read-only transactions. The map UB_{cid} only contains least upper bounds that correspond to transactions whose immediate predecessors have not yet received a response.

Once a read is retried, responses matching the original invocation must be discarded. Since the $lsnConst$ of retries might result in newer data than the original read, we wish to prevent responses from the previous invocation matching with the retry. This serves to ensure a consistent snapshot of the database state.

Procedure SessionRuntimeReadRetry($crsn_T$)

1 $lubCrsn := \min\{x \in UB_{cid}^{\text{key}} : crsn_T \leq x \wedge UB_{cid}[x] \neq \text{None}\}$
2 $lsnConst := UB_{cid}[lubCrsn]$
`// Retry all read-only transactions up to least upper bound`
3 **foreach** $sn \in [crsn_T, lubCrsn]$ **do**
4 $\quad$ __, __, T, __ := $ReadResult_{cid}[sn]$
5 $\quad$ $ReadResult_{cid}[sn] := (\text{None}, \{\}, T, lsnConst)$ `// discard partially completed read`
6 $\quad$ $\text{Send}_i(\text{ReadOnlyTransact}, \{T, sn, \text{None}, lsnConst\})$
7 **end**

Observe that any one retry automatically forces retries for all later reads that have not yet returned. In the worst case, this can potentially lead to a large amount of cascading retries. For example, responses for a large set of read-only transactions may arrive just after all retries are sent out, thereby invalidating their content. However, we expect that manifestation of this behavior is highly unlikely, assuming a reasonable timeout scheme.

The client library exposes the SessionRespRead RPC for shards to return read-only transaction results. The procedure is essentially the same as ExecNotif on the transaction manager tail. Once all shards have returned data, the transaction is considered complete and the result returned to the client. In the process, UB_{cid} is updated to reflect the log sequence number associated with the result.

Procedure SessionRespRead($data$, $crsn_T$, lsn, $numShards$)

```
   // SessionRespRead on client cid
 1 (currNum, currData, currLsn) := ReadResult_cid[crsn_T]
 2 if lsn = currLsn then
 3 |   if currNum = None then
 4 |   |   currNum := numShards
 5 |   currNum := currNum − 1
 6 |   currData := currData ∪ {data}
   |   // Received data from all shards, so can return to client
 7 |   if currNum = 0 then
 8 |   |   ReadResult_cid := ReadResult_cid \ {crsn_T ↦ ReadResult_cid[crsn_T]}
   |   |   // If crsn_T is not present, preceding result has already returned
 9 |   |   if crsn_T ∈ UB_cid^key then
10 |   |   |   UB_cid[crsn_T] := lsn
11 |   |   |   if crsn_T ≠ crsn_curr − 1 then
   |   |   |   |   // No longer need successor's log sequence number
12 |   |   |   |   UB_cid := UB_cid \ {(crsn_T + 1) ↦ UB_cid[crsn_T + 1]}
13 |   |   return currData
14 |   else
15 |   |   ReadResult_cid[crsn_T] := (currNum, currData, currLsn)
16 |   end
```

Upon receiving a read-only transaction, the manager node connected to the client session will first wait for any read-write dependencies. Then, it computes a consistent read across shards by finding an appropriate log sequence number to associate with the request. This log sequence number constitutes the logical timestamp of the database snapshot, and its computation must satisfy three types of constraints. First, the timestamp should reside in a range derived from *writeDep* and *readBound*. Second, the timestamp should conform to the client library retry timestamp *lsnConst*, if provided. Finally, the manager node must account for read-write transaction completion it learns through ExecAppendTransact, which imposes a staleness constraint.

If the client provides *lsnConst*, then the manager node defers to this value as the log sequence number. Since the client library is the only portion of the protocol responsible for retry correctness, it is safe for the manager node to do this. If the *lsnConst* is not provided, then the manager node must find a log sequence number on the client's behalf. To ensure preservation of invocation order, the request's CRSN is checked against the largest seen CRSN for that client. If it is less, the log sequence number is set to be equal to that of the largest CRSN. If it is greater, we take the maximum of the log index associated with the read-write dependency and $exec(Q_i|j)$, over each shard group j involved in the transaction. Then, we set the log sequence number equal to the minimum of this value and *logIndex* of the client's next read-write transaction, if it exists. Since manager notes cannot discern whether or not any given read-write transaction's response has been delivered, taking the maximum over $exec(Q_i|j)$ is necessary to prevent stale reads. Taking the minimum with the *logIndex* associated to the client's next read-write transaction ensures reads never push past subsequent (in invocation order) read-writes. After all this, the entry in R_i is updated.

We remark that the log sequence number computation can be visualized as a "fence" that cuts across the manager's queues. In fact, this fence is precisely a lower bound on the maximum staleness of reads allowed by regular sequential serializability. If the manager node instead pushed the fence to be the maximum over $tail(Q_i|j)$, we would obtain a strictly serializable system.

Procedure ReadOnlyTransact(T, $crsn_T$, $writeDep$, $lsnConst$)

```
   // ReadOnlyTransact on manager node i ∈ [1, n]
 1 if writeDep > W_i[crsn_T.cid] then
 2 |   wait until writeDep ∈ Ongoing_i^key or ABORT(writeDep)
 3 fence := Ongoing_i[writeDep].logIndex or L.find(writeDep)
 4 upperFence := Ongoing_i[writeDep + 1].logIndex or L.find(writeDep + 1) or None
 5 shardSet := ∅
   // Participating shards and preliminary fence computation
 6 foreach j ∈ [1, m] do
 7 |   particip := T^read ∩ sh(j)
 8 |   if particip ≠ ∅ then
 9 |   |   shardSet := shardSet ∪ {j}
10 |   |   fence := max{fence, exec(Q_i|j)}
11 end
   // Modify constraints based on retry flag and R_i
12 if lsnConst ≠ None then
13 |   fence := min{lsnConst, upperFence}
14 if crsn_T.cid ∈ R_i^key then
15 |   if crsn_T.num > R_i[crsn_T.cid].maxNum then
16 |   |   R_i[crsn_T.cid] := (crsn_T.num, fence)
17 |   else if lsnConst = None then
18 |   |   fence := min{R_i[crsn_T.cid].lsn, upperFence}
19 else
20 |   R_i := R_i ∪ {crsn_T.cid ↦ min{fence, upperFence}}
21 end
   // Send reads and constraints to shards
22 numShards := |shardSet|
23 foreach j ∈ shardSet do
24 |   Send_j(ShardExecRead, {T|j, crsn_T, fence, numShards})
25 end
26 return
```

Due to the check against R_i (lines 14-21), it is not strictly necessary for the client library to declare a dependency on the most recent read-write transaction. If the write key-set of all ongoing transactions are known at time of invocation, a dependency need only be declared on the latest transaction X such that $X^{\text{write}} \cap T^{\text{read}} \neq \emptyset$. Unfortunately, this optimization is not always possible due to the way we handle read-write and dependent transactions (§5.5). Also note that $writeDep$ entry of $Ongoing_i$ may have already been garbage collected before the arrival of a read. In this case, we must scan the log directly to ascertain the associated log index. This also applies to $writeDep + 1$. In practice, a staleness check can be used to simply reject reads that depend on read-writes that are too old.

A shard group simply waits until the manager-determined constraint is satisfied before executing its

portion of the read. Shards return directly to the client, $crsn_T.cid$, which originated the request. Because correctness depends only on execution up to a specified log sequence number, writes on the shard can continue uninterrupted in parallel.

Procedure ShardExecRead($T|j, crsn_T, fence, numShards$)

```
   // ShardExecRead on shard group j
1  wait until shExec_j ≥ fence
2  data := ∅
3  foreach op ∈ T|j do
   |  // multi-versioned get exposed by storage service, returns value with greatest
   |     version less than fence
4  |  data := data ∪ {op.key ↦ get(op.key, fence)}
5  end
6  Send_{crsn_T.cid}(SessionRespRead, {data, crsn_T, fence, numShards})
7  return
```

While this method of handling reads is safe (see §7), progress is not guaranteed. The issue is the wait in line 1 of ShardExecRead. It is possible that a read-only transaction will have a $fence$ that is larger than $shExec_j$. If there are no more subsequent writes to that shard, the read will block forever. Thus, the transaction manager must periodically send *flush* messages to the shards. These flush messages contain the index of the last entry on the transaction manager log $\mathcal{L}$. Shard groups can then safely serve all reads with $fence$ less than this value. These flush messages may be periodically sent by any node in the chain. To reduce read latency, flushes may also be shard driven. If a shard notices it has buffered a read for a specified amount of time, it can contact any manager node to send an early flush message.

5.5 Dependent Read-Write Transactions

We now extend the protocol to cover the case of dependent read-write transactions. These types of transactions are buffered at the head node, which fully reduces the transaction into a simple read-write transaction. More specifically, the head node makes the necessary read requests to compute the full read/write key set of the transaction. For deduplication, the head maintains a set BF of all buffered transactions, in addition to the state shown in figure 5. Buffering these transactions on the client instead of the head would enable the *writeDep* optimization mentioned above, but the tradeoff would be an increase in the number of client-manager node round trips. In view of this, we choose to keep reduction of dependent read-write transactions as an internal function of the transaction manager.

Unfortunately, contention and aborts are possible when clients elect to use these types of transaction. Since the reduction process involves one or more reads, any write that affects the read keys must trigger a transaction abort. Otherwise, we would introduce safety violations. To prevent large volumes of interfering simple read-write transactions from starving a dependent transaction, one could wound-wait. We omit

this from our procedure pseudo-code for brevity. When a transaction aborts, the head sends an ABORT notification down the chain, enabling constraint removal on affected reads (line 2 of ReadOnlyTransact).

To submit a dependent read-write transaction, the client uses a slightly modified SessionRuntimeAppend, with ReduceTransact replacing AppendTransact in the call on line 5.

Procedure ReduceTransact(T, $cwsn_T$, $ackBound$)

```
   // ReduceTransact on head i = 1
 1 if cwsn_T ∈ BF then
 2 |  return
 3 BF := BF ∪ cwsn_T
 4 toRead := DepAnalysis(T^keys) // get read dependencies
 5 foreach j, ops ∈ toRead do
 6 |  Send_j(ShardExecHeadRead, {ops, tail(Q_1|j)})
 7 end
 8 if ABORT(cwsn_T) then
 9 |  BF := BF \ cwsn_T
10 |  Send_{cwsn_T.cid}(SessionRespWrite, {cwsn_T, (ABORT, ∅, None)})
11 |  Send_succ(ABORT, {cwsn_T})
12 |  return
13 else
14 |  collect all read results into readResults
15 end
16 T_reduce := ReduceDeps(T, readResults)
17 AppendTransact(T_reduce, cwsn_T, ackBound)
```

Here, ShardExecHeadRead is the same as ShardExecRead, except that results are returned to the head and neither $crsn_T$ nor $numShards$ need to be passed as arguments. Further notice that the $fence$ argument is always set to the tail of the corresponding queue, resulting in strictly serializable reads. This is because any buffered transactions will be logically ordered after all currently executing ones. Thus, dependency resolution must observe the effects of all interfering read-write transactions that are already on the log and queues.

5.6 Failure Recovery

Failure recovery on the shard groups is straightforward, as Raft handles this automatically. Failures in the chain require a bit more attention. Either a middle node, head, or tail can fail in the transaction manager chain.

Middle node failures are the simplest case. Upon learning of a failure, the failed node's predecessor and successor update their network connections and continue as normal. On the failure of the tail, the new tail must retry all INPROG transactions contained within W_i. This is since the old tail may have already received partial results from shards. A minor detail is that manager nodes do not replicate the shard sequence

numbers nor read dependencies associated to each sub-transaction. However, this is easily reconstructed by starting from the earliest INPROG transaction and iterating to the end of the log, running DepAnalysis on each entry. Similarly, on failure of the head, the new head must resend responses for every transaction with status SUCCESS in W_i.

5.7 Extensions

This section touches upon some extensions and optimizations of the core RepQueue-RSS protocol.

5.7.1 Checkpointing

Checkpointing on the Raft groups can be handled using the modified Zig-Zag algorithm presented in Calvin [5, 15]. For checkpointing on the transaction manager nodes, it suffices to specialize this algorithm to just one key: the log.

For completeness, we give a brief summary of the algorithm. Cao et al's Zig-Zag algorithm maintains two copies of the data in memory, $AS[K]_0$ and $AS[K]_1$, in addition to two additional bitmaps $MR[K]$ and $MW[K]$. The bit of $MR[K]$ specifies which copy of the data to read and $MW[K]$ specifies which copy of the data to write. Thus, updates or new values are always written to $AS[K]_{MW[K]}$ and any new updates result in $MR[K]$ being set to $MW[K]$. Then, on every checkpoint, $MW[k]$ is set to $\neg MR[K]$ and a separate thread can take a snapshot by reading $AS[K]_{\neg MW[k]}$.

Notice that the original version of Zig-Zag calls for the start of each checkpoint period to wait until the database state is entirely quiesced. This ensures that $AS[K]_{\neg MW[k]}$ provides a consistent view of database state. However, in Calvin and our system, the log sequence numbers provide a global serialization order. Thus, simply choosing an log sequence number implicitly fixes a consistent snapshot. The Calvin authors call this a *virtual* point of consistency. When the nodes observe log sequence numbers approaching a predetermined virtual consistency point, they maintain two copies of the data – one "before" copy and one "after" copy. The "before" and "after" play roles analogous to $AS_0[K]$ and $AS_1[K]$. After the log sequence numbers grow larger than the virtual consistency point, no more updates are applied to the "before" copy, and a checkpoint thread can safely save it to disk. Once complete, the "after" copy can delete all data before the virtual consistency point and the "before" copy is garbage collected.

5.7.2 Multicast

As with Terrace and Freedman's CRAQ, we can leverage multicast protocols to decrease read-write latencies, especially for workloads with large updates or when transaction replication factor is high. We do not impose

any ordering or reliable delivery guarantees on the multicast [14]. Both network or application-level multicast protocols can be used, but note that unlike CRAQ, the head must sequence read-writes before a multicast in order to preserve invocation order. That is, the chain minus head forms the multicast group instead of the entire chain.

The chain head can then multicast full log entries, and then only a small commitment message needs to travel down the rest of the chain. This takes expensive network serialization and transmission off the critical path of transaction replication. If any manager node fails to receive a multicasted entry, it can initiate a "pull" from its predecessor before forwarding the commit message. The backwards direction admits the same optimization, with the multicast group in this case being the chain minus the tail.

5.7.3 Optimizations for the Wide-area

In the wide area, transaction manager nodes and replicas are placed in different geographic regions. The logical chain topology has a marked disadvantage in this case, as each additional node in the chain incurs another wide-area round trip. As a result, one would expect the write latency experienced by clients to grow proportionally with respect to number of transaction manager nodes. While we cannot completely resolve this problem without significant changes or a system redesign, we can make two helpful adjustments, both involving reorganization of chain nodes and shard replicas.

Observe that the shard procedure in read-only transactions does not require the leader to service the request. Indeed, any of the replicas, including the one geographically closest to the client, can safely serve read results if both freshness and ordering constraints are met.

Further note that replication happens twice: once to log the transaction, and another to replicate data. We can generalize our system model to distribute shard replicas amongst nodes in the transaction manager chain. Each node can have just one shard replica, a Raft group of replicas, or none at all. Each node communicates directly with its shard replicas to serve read-only transactions and replicate read-writes. As before, however, the tail of the chain determines commitment, and read-write responses are only returned after a full forwards-backwards pass through the chain. Here, we trade off increased intra-region communication for at least one less wide-area round-trip.

6 Evaluation

6.1 Implementation

The Spanner-RSS with multi-dispatch client baseline builds on top of the original Spanner-RSS C++ codebase, which itself is based of TAPIR's experimental framework. The implementation uses view-stamped replication instead of Multi-Paxos but is an otherwise faithful reproduction of the protocol as described in [7]. Both shards and clients are single-threaded, using libevent to avoid blocking on network I/O.

For our evaluation, we only implement the read-write wrapper and read-only dependency declaration portions of the client. We did not add read-at-timestamp support to Spanner-RSS, which is problematic for asynchronous read-only transactions due to the presence of clock skew. As a result, our implmentation can guarantee invocation ordering on writes but not on reads. Moreover, removing the client specified timestamp allows shards to return earlier in some cases, rather than waiting for t_{safe} to become sufficiently large. We also did not enforce the rule that read-write transactions following read-only transactions wait until $current\_time.latest$ passes. This could lead to scenarios where read-only transactions are reordered after subsequent read-write invocations. Similarly to the previous omission, this actually helps our baseline in terms of performance. Nonetheless, we will see in the evaluation results that the potential performance gain due to the lack of these components has no impact on our conclusions. To prevent the worst-case scenario of multiple aborts on write lock upgrade, we use the GetForUpdate facility, which acquires an exclusive write lock on reading specified key.

The ReqQueue-RSS database and client library are both written in Rust using the multithreaded Tokio asynchronous runtime and Tonic gRPC over HTTP/2 for transport. We rely on the OpenRaft[3] library for the implementation of Raft. Observe that at no point in our protocol did we assume usage HTTP nor TCP. As such, we could have opted to use a much more barebones method of inter-node communication, such as simple messaging over UDP. Nonetheless, the usage of a well-maintained RPC and serialization ecosystem significantly eased the implementation burden. Future work could explore more performant network communication methods.

A completely faithful comparison between the baseline and our system would involve (re)implementing everything in one framework. Again, we believe that any performance differences induced by the choice of language and implementation strategy do not affect outcomes in any meaningful way. The source code of both ReqQueue-RSS and Spanner-RSS with MD client are freely available.[4]

Due to high resource contention and time constraints, we were not able to run experiments on CloudLab.

[3] https://github.com/datafuselabs/openraft

[4] https://github.com/aaronwu667/RepQueue-RSS, https://github.com/aaronwu667/spanner-rss-md

We remark that both codebases and experiments are equipped to run remotely, so this would be a first priority for future work. We ran each experiment with three shards, each with three replicas. Keys are drawn from a Zipf distribution with skew 0.7 and benchmark clients are closed loop. Specific to RepQueue, we set the chain to be of length 3, and clients communicate with the service with the non-tail nodes. Specific to Spanner, we set the TrueTime error to be the figure given in the original work, 7ms.

6.2 Experiment Results

Our single client latency experiments aim to quantify the end-to-end latency. That is, given a set of asynchronous invocations, this measures how long the client must wait for the all the requests to complete. To get a better idea of how the system behaves, we plot the latencies of each individual request against their corresponding number in the invocation sequence. The maximum end-to-end latency a client experiences is simply the largest data point, but the resulting graph gives us better insight into how the overall latency accumulates over each request. While we tag each request with a sequence number, our benchmark clients do not introduce any delay between sending requests. Thus, our experiments aim to test Repqueue-RSS and our Spanner baseline against the highest possible level of client concurrency.

For the write-only workload, each request consists of writes to a random number of keys between 1 and 10. For the mixed workload, read-only transactions read a random number of keys between 1 and 10, and writes of the aforementioned form are interspersed every 10 read-only transactions. Figure 6 shows the results for the write-only workload.

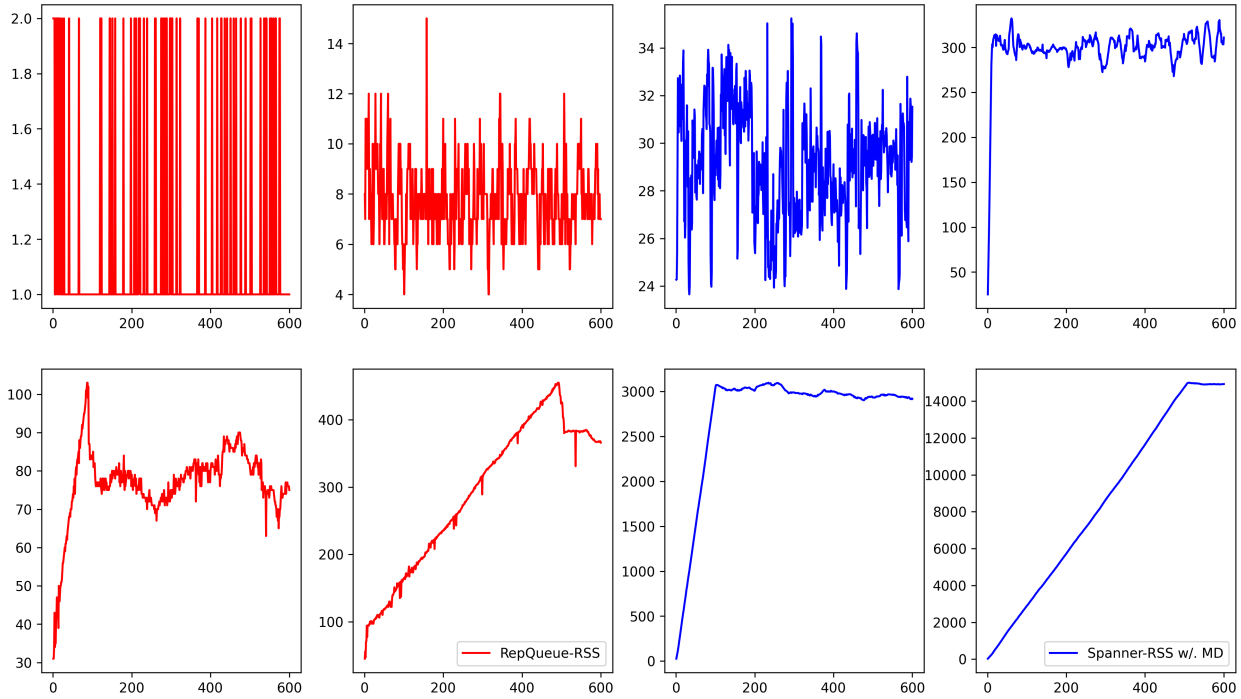

Figure 6: Write-only latency results for RepQueue-RSS and baseline. Latencies measured in milliseconds. We varied the number of maximum allowed outstanding invocations as an independent parameter. Starting from the top left and going clockwise the values are 1 (no asynchronous invocation at all), 10, 100, and 500.

In the cases of 100 and 500 concurrent requests, we can see that for both our baseline and Repqueue-RSS, the initial burst of concurrent requests causes a linear increase in latency. In the case of Spanner-RSS, this is due to the effective serialization of all requests around shared access to the client's sequence number. For Repqueue-RSS, this is due to the sequencing by transaction manager nodes. While the initial linear trend is the same, one notes the difference in slope between the left and the right. The cost of lock contention and TrueTime commit wait for each transaction outweighs the forwards-backwards pass in RepQueue-RSS.

Another feature of note is the latency behavior after the initial batch of concurrent requests. In both RepQueue-RSS and the baseline, the linear increase stops after the request number reaches the maximum allowed number of concurrent transactions. For the baseline, we see this happen in all experiments with a concurrent client. After the increase stops, the latencies remain around the largest point on the line. This is likely due to continued contention around the client's sequence number. At any time, there are the maximum allowed asynchronous transactions vying for access to the one key. Since the lock implementation of Spanner-RSS uses a fairness heuristic (wound-wait), all transactions block for approximately the same amount of time before being able to proceed.

This phenomenon manifests on RepQueue-RSS in the experiments with number of allowed outstanding

invocations equal to 100 and 500. In contrast to Spanner-RSS with multi-dispatch, the latencies decrease after the initial linear growth. We can reason about this by thinking of the entire system as a funnel. At the outset of the experiment, all client requests more or less arrive simultaneously. They must then wait after each hop in the system, either to be serviced or to preserve ordering guarantees. Since requests are serviced in order, responses are also returned roughly in order, assuming load is evenly spread amongst shards. Furthermore, we use closed loop clients, so the order of responses then determines the order of subsequent requests. As a result, each concurrent request is therein implicitly assigned a "frequency", resembling time-division multiplexing. Within its "frequency," requests block for a much smaller subset of all outstanding invocations. This leads to the decreased latency we see above.

For a clearer visualization of how the two compare, we overlay the two graphs on a log scale in figure 7.

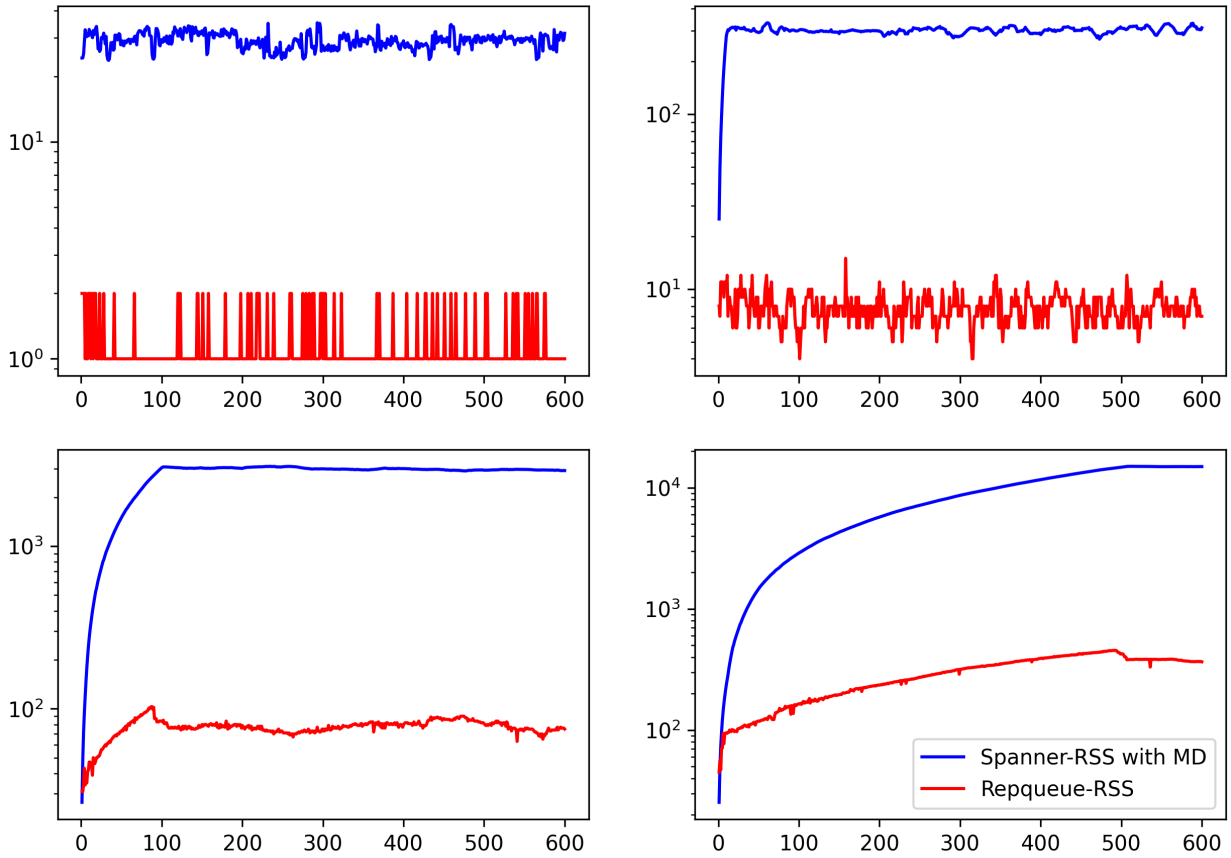

Figure 7: Write-only latency results on log scale

The mixed workload results mostly reflect the conclusions of the write-only workload. Both RepQueue-RSS and Spanner-RSS with MD allow reads to be executed in parallel. For Spanner-RSS, we see step function-like behavior. Each batch of 10 reads are executed in parallel, and thus, we do not see much increase in latency between each read-only transaction. On the other hand, we have established previously

that read-write transactions are effectively serialized. Thus, on every read-write transaction, we observe a jump in latency.

The results for RepQueue-RSS look a bit different, but the general concept is the same. We can still see this step function-like behavior, but the latency on each read-write transaction spikes well above the "step". The reason for this is the necessary backwards pass through the chain before return to the client. Notice that in our ReadOnlyTransact RPC, read-only transactions can be sent to the shard groups immediately after its write dependency is replicated on that manager node. Thus, read-only transactions need only buffer for at most one forwards pass down the chain.

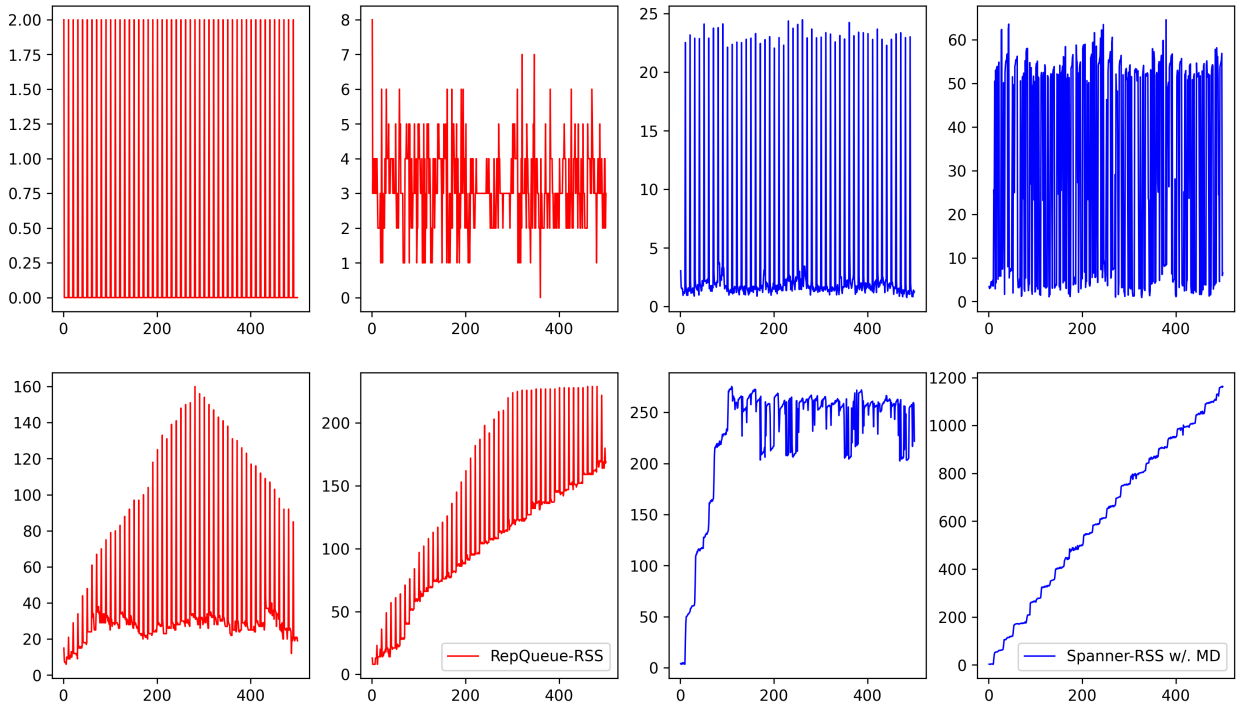

Figure 8: Mixed workload latency results for baseline and RepQueue-RSS. Latencies are measured in milliseconds. As before, the number of maximum allowed outstanding transactions are 1, 10, 100, and 500 going clockwise.

As before, we provide the same results merged onto one plot with a log-scaled y-axis.

In general, we see that RepQueue-RSS provides a single, concurrent client with significantly lower latencies. However, we anticipate that RepQueue-RSS loses its advantage over Spanner-RSS with MD when TrueTime ϵ is close to 0. Recent work has shown that this is not as unrealistic as one might expect [11].

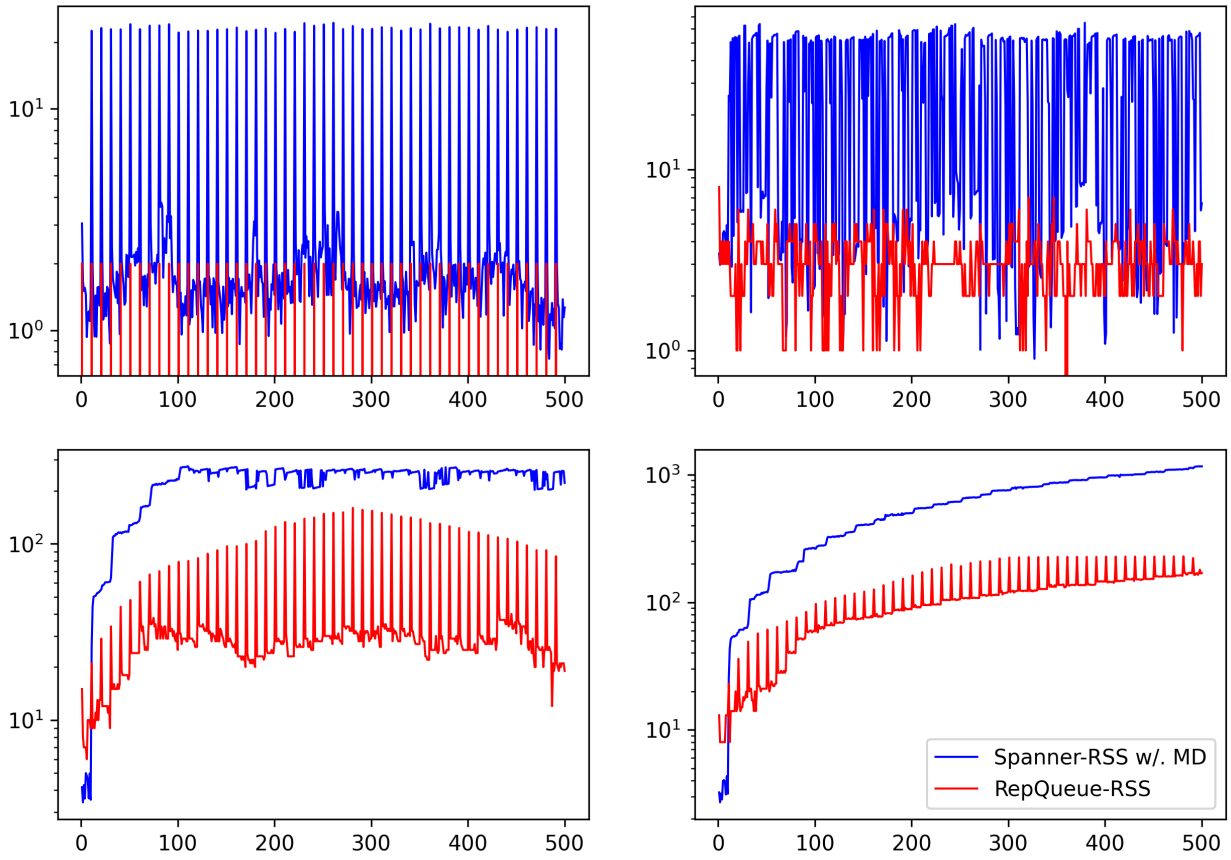

Figure 9: Mixed workload latency results on log scale

7 Proof of Correctness

In this section, we turn to proving that RepQueue-RSS guarantees MD-RSS. The main correctness result is as follows.

Theorem (Safety): Any execution, σ, of the ReqQueue-RSS satisfies MD-RSS.

First, we introduce a collection of helpful lemmas.

Lemma 1: The transaction manager log order is consistent across all nodes and respects invocation order of state updates across all clients.

Proof: The head checks to ensure no reordering of client requests in line 7 of AppendTransact. It then decides the transaction's position in the log. Each successive node in the chain then checks against log reordering and invocation reordering in line 9 of AppendTransact. □

Lemma 2: Log sequence numbers associated with read-write transactions are strictly increasing in CWSNs.

Proof: This follows from the fact that the transaction manager log is append-only and that new read-write transactions are always appended to the log. See lines 12-13 of AppendTransact. □

Lemma 3: Log sequence numbers associated with read-only transactions are monotonically increasing in CRSNs.

Proof: If the read-only transaction is not a retry, the check against the entry stored in R_i on lines 14-21 of ReadOnlyTransact ensures monotonicity. In the case that the read-only transaction is a retry, SessionRuntimeReadRetry ensures that all outstanding read-only transactions with greater CRSN are retried up to some read that has already returned. Additionally, all retried reads use the log sequence number of the already returned read, which is stored in UB_{cid}. This prevents the log sequence number from ever decreasing. See line 7 of SessionRuntimeRead, SessionRuntimeReadRetry, and lines 9-10 of SessionRespRead. □

Lemma 4: For any given subsequence of the transaction manager log that affects shard group j. Shard group j will commit its corresponding sub-transactions respecting log order and invocation order.

Proof: From lemma 1, we know that the tail will process transactions in log order. Since the shard sequence number for j is incremented by one for every transaction in which it participates, the shard sequence numbers must preserve both log ordering and invocation ordering. Once sent over the network, the shard groups check against reordering using the sequence numbers it receives. See lines 18-30 of AppendTransact and 1-5 of ShardExecAppend. Specific to Raft, shards groups guard against incorrect retries by checking sequence numbers again before applying committed log entries. □

Lemma 5: CRSNs and CWSNs preserve invocation order on read-only transactions and read-write transactions, respectively.

Proof: Immediate by lines 4-5 of SessionRuntimeRead and lines 2-3 of SessionRuntimeAppend. □

We are now ready to prove the main theorem. Our strategy will be to first define a total ordering on σ, using log indices as logical timestamps. We will then show that the resulting sequential execution σ_{ord} is in our desired sequential specification. Finally, we will verify that σ_{ord} is a sequence that satisfies both multi-dispatch and regular sequential serializability. As in our definitions, we assume that clients do not communicate with each other for simplicity. This allows us to focus on executions that only comprise of service invocations and responses.

Proof of Theorem: Let W be the set of read-write invocation and response pairs in σ, and let R be the

set of read-only invocation and response pairs. Observe that pair in W corresponds to a unique index in the transaction manager log $\mathcal{L}$. Similarly each pair in R has an associated $fence$, which is also an index into the transaction manager log $\mathcal{L}$. For read-write pair w, denote its associated index $\text{ind}(w)$. For read pair r associated with client j, we set $\text{ind}(r)$ to be its $fence$. Here, we will always take the last computed $fence$ for read-only transactions. That is, if a read-only transaction experiences a retry, we associate $fence$ with the log sequence number of the retry and not the original invocation.

The definition of $\text{ind}(\cdot)$ is well-defined due to lemma 1. In particular, there is an unambiguous choice of what the index should be, as all nodes have a consistent log. Notice we can treat these $\text{ind}(\cdot)$ as logical timestamps to define an strict total order over σ. Concretely, let $\sqsubset$ be the binary relation over $\sigma \times \sigma$, such that for $a, b \in \sigma$ one has $a \sqsubset b$ if and only if one of the following hold

i.) $\text{ind}(a) < \text{ind}(b)$

ii.) $\text{ind}(a) = \text{ind}(b)$ where $a \in W$, $b \in R$, and both are from the same client.

iii.) $\text{ind}(a) = \text{ind}(b)$ where $a, b \in R$, both are from the same client, and $\text{CRSN}(a) < \text{CRSN}(b)$.

iv.) $\text{ind}(a) = \text{ind}(b)$ where a, b are from different clients, and the *cid* of the client who issued a is strictly less than that of b i.e. we arbitrarily break ties.

We show that this is a strict total order over transactions.

Irreflexivity: Let a be a read-write transaction. We have that $\text{ind}(a) = \text{ind}(a)$, so we check that rules (ii)-(iv) do not apply. Clearly a cannot simultaneously be a read-only and a read-write transaction, so we can rule out (ii). Since a is a read-write transaction, it does not have a CRSN, which covers (iii). Each transaction is from one client, so rule (iv) does not apply. The reasoning for read-only transactions is the same.

Transitivity: Transivity through rules (i), (iii), and (iv) follow from the usual ordering on natural numbers. Suppose that $a \sqsubset b$ through (ii) and $b \sqsubset c$. If $b \sqsubset c$ through (i), then we are immediately done. Since $b \in R$ we cannot have $b \sqsubset c$ through (ii). If $b \sqsubset c$ through either (iii) or (iv), then we can apply rule (ii) again to get $a \sqsubset c$. Now instead suppose that $a \sqsubset b$ and $b \sqsubset c$ through (ii). Then b must be a read-write transaction. Thus, we can only have $a \sqsubset b$ through (i), and we are immediately done.

Connectedness: Let $a \neq b$. If both are read-only transactions, rules (iii) and (iv) give an ordering. If one is read-write and the other is read-only, then rules (i) and (ii) apply. Finally, if both are read-write, then (i) combined with the fact that the transaction manager log $\mathcal{L}$ is append-only gives an ordering.

The total order $\sqsubset$ induces the invocation-response pair sequence σ_{ord}. Claim that this is in our desired sequential specification. Consider a read of a key and suppose that this key lives on shard α. We first consider the case that the read belongs to a read-only transaction T. We know by lemma 4 that shards always apply state changes respecting log order. Furthermore, the multi-versioned get exposed by the storage service combined with the wait rule in line 1 of ShardExecRead will always yield a value with the greatest version less than or equal to $fence$. Thus, the read will return the result of the latest write in this case.

Now consider the case where the read is part of a read-write transaction. All read-write transactions are appended to the end of the log, and thus its log index will be greater than all preceding writes. Thus, multi-versioned gets in lines 8-12 of ShardExecAppend will return the most recent writes to the key. In both cases, reads return the last value written as was to be shown.

Observe that σ_{ord} may contain responses that are not in σ. We thus extend σ by adding responses appropriately. Consider the case of a read-write invocation. For each key k read by this invocation, return the value written by the closest preceding read-write transaction in σ. We handle the case of read-only transactions in the same way. Denote the extension of σ as σ_1.

We finish by checking that σ_{ord} is a sequence satisfying multi-dispatch regular sequential serializability as defined in 2.3.2.

Equivalence: Here, we need to show $\sigma_{\text{ord}}|j = (\sigma_1|j)_{\text{seq}}$. By construction of σ_{ord} and σ_1, the two contains exactly the same invocations and responses. It suffices to demonstrate the preservation of ordering between transactions. For clarity, we will use the notation $\rhd$ to denote invocation order. There are four cases

- (RO $\rhd$ RO) Let $T_1 \rhd T_2$ and both be read-only transactions. By lemma 3, $\text{ind}(T_1) \leq \text{ind}(T_2)$. Lemma 5 tells us that the CRSN of T_1 is strictly less than that of T_2. Thus, by rule (iii), we must have that T_1 precedes T_2 in $\sigma_{\text{ord}}|j$.

- (RW $\rhd$ RO) Let $T_1 \rhd T_2$, with T_1 being a read-write transaction and T_2 being a read-only transaction. The client library declares a dependency $writeDep$ on the most recent read-write from the same client, which ReadOnlyTransact subsequently translates a log index in lines 1-3. The computation of $fence$ thus guarantees that $\text{ind}(T_2)$ will be at least $\text{ind}(T_1)$. From rule (ii), we get $T_1 \sqsubset T_2$ in $\sigma_{\text{ord}}|j$.

- (RO $\rhd$ RW) Let $T_1 \rhd T_2$ be such that T_1 is a read-only transaction and T_2 is a read-write transaction. On lines 13 and 18, ReadOnlyTransact ensures that the $fence$ of T_1 is no larger than that of T_2.

- (RW $\rhd$ RW) For read-write transactions T_1 and T_2 such that $T_1 \rhd T_2$, lemmas 2 and 5 give $\text{ind}(T_1) < \text{ind}(T_2)$. Therefore $\text{ind}(T_1) \sqsubset \text{ind}(T_2)$. By rule (i), we are done.

Obeys Asynchronous Casual Precedence: As above, we do a case-by-case analysis.

- (RO $\rightsquigarrow$ RO) If T_1 and T_2 are both read-only, the only possible causality in our model of computation is through process order. By lemma 3 and our tiebreaking rule (iii), we must have $T_1 \sqsubset T_2$.
- (RO $\rightsquigarrow$ RW) If T_1 is a read-only transaction and T_2 is a read-write, then as before, the only possible causality is through process order. Since the read-only receives a response before the invocation of the read-write, we are guaranteed that the associated log index of the read-only transaction is at least one less than the read-write. We conclude that $T_1 \sqsubset T_2$ in this case.
- (RW $\rightsquigarrow$ RO) If T_1 is a read-write transaction and T_2 is a read only transaction, our construction of ind(T_2) and property (ii) again give us $T_1 \sqsubset T_2$.
- (RW $\rightsquigarrow$ RW) If T_1 and T_2 are read-write transactions related by process order, lemma 2 yields the desired result. If T_2 reads from T_1, then by lemma 4, the writes of T_1 must be associated with a smaller log sequence number. In either case, $T_1 \sqsubset T_2$

Since in all cases we have $T_1 \sqsubset T_2$, we are done (no need to consider transitivity).

Conflicting Transactions: Suppose T_1 and T_2 are read-write transactions. Since all read-write transaction are appended to the log and read-write responses are only returned from the head after a full forwards-backwards pass through the chain, if $T_1 \rightarrow T_2$, then T_2 will have strictly greater log index than T_1. Now let be T_1 a read-write transaction and T_2 a conflicting read-only transaction. Suppose the conflict involves keys on shard α. By lemma 4 and construction of the queues $Q_i|\alpha$, we know that $exec(Q_i|\alpha)$ will always contain the log sequence number of the most recently executed transaction on shard α. Then, by the $fence$ computation in ReadOnlyTransact, ind(T_2) must be at least ind(T_1). If strict equality holds, then rule (i) gives $T_1 \sqsubset T_2$. In the second, rule (ii) gives the same result.

□

8 Conclusion and Future Work

In a software landscape that has increasingly adopted concurrency and asynchrony to drive performance gains, consistency models should also evolve to meet the needs of client applications adopting these programming paradigms. In this thesis, we have demonstrated through various examples that existing protocols and consistency models are not adequate to provide the sort of strong guarantees a purely single-threaded synchronous client can expect. To rectify this gap in safety, we introduced a framework, multi-dispatch, for augmenting traditional consistency models, allowing them to extend beyond purely synchronous executions. This framework was applied to regular sequential serializability, a weakened version of strict serializability

that maintains application invariants. Using the resulting consistency model as a guide, we examined how clients using an existing system, Spanner/Spanner-RSS, can modify their usage of the API to acheive safety with multiple outstanding invocations. From this, we then presented the design of a system built specifically to acheive multi-dispatch regular sequential serializability, RepQueue-RSS. In benchmarks against Spanner-RSS with modified client usage, RepQueue-RSS acheives significantly lower latency under concurrent (and non-concurrent) usage in the single datacenter setting.

A recurring theme in the protocol of RepQueue-RSS is the notion of sequencing to ensure multi-dispatch. RepQueue-RSS accomplishes through two principle means. First, it explicitly tracks sequence numbers at each network hop as in TCP, blocking transactions from proceeding if their sequence number is out of order. The second is the transaction manager sequencing layer. Both of these have downsides. The blocking on sequence numbers is inherently pessimistic and drastically reduces the level of parallelism possible at each node in the system. Future work in this area could explore more optimistic ordering schemes, perhaps leveraging more fine-grained data about dependencies. The usage of a centralized sequencer imposes a throughput bottleneck. From figure 4, it is clear that the head and especially tail must deal with much more traffic than the rest of the system. Future work in this direction could involve investigating the usage of multi-sequencers such as Hydra [6] or a using a completely different system design.

9 Acknowledgements

I would like to thank Professor Lloyd and Jeffrey Helt for their patient guidance. Our weekly discussions were enlightening and always provided me with unexpected insights. I would also like to thank my family for the constant support throughout my senior year at Princeton.

References

[1] etcd implmentation of raft. https://github.com/etcd-io/raft. Accessed: 2023-04-18.

[2] Hashicorp implmentation of raft. https://github.com/hashicorp/raft. Accessed: 2023-04-18.

[3] Tikv implmentation of raft. https://github.com/tikv/raft-rs. Accessed: 2023-04-18.

[4] Mahesh Balakrishnan, Chen Shen, Ahmed Jafri, Suyog Mapara, David Geraghty, Jason Flinn, Vidhya Venkat, Ivailo Nedelchev, Santosh Ghosh, Mihir Dharamshi, Jingming Liu, Filip Gruszczynski, Jun Li, Rounak Tibrewal, Ali Zaveri, Rajeev Nagar, Ahmed Yossef, Francois Richard, and Yee Jiun Song. Log-structured protocols in delos. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating*

Systems Principles, SOSP '21, page 538–552, New York, NY, USA, 2021. Association for Computing Machinery.

[5] Tuan Cao, Marcos Vaz Salles, Benjamin Sowell, Yao Yue, Alan Demers, Johannes Gehrke, and Walker White. Fast checkpoint recovery algorithms for frequently consistent applications. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 265–276, New York, NY, USA, 2011. Association for Computing Machinery.

[6] Inho Choi, Ellis Michael, Yunfan Li, Dan R. K. Ports, and Jialin Li. Hydra: Serialization-Free network ordering for strongly consistent distributed applications. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 293–320, Boston, MA, April 2023. USENIX Association.

[7] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 261–264, Hollywood, CA, October 2012. USENIX Association.

[8] Jeffrey Helt, Matthew Burke, Amit Levy, and Wyatt Lloyd. Regular sequential serializability and regular sequential consistency. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 163–179, New York, NY, USA, 2021. Association for Computing Machinery.

[9] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*. USENIX Association, June 2010.

[10] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 245–256, 2011.

[11] Ali Najafi and Michael Wei. Graham: Synchronizing clocks by leveraging local clock properties. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 453–466, Renton, WA, April 2022. USENIX Association.

[12] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[13] Herb Sutter et al. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

[14] Jeff Terrace and Michael J. Freedman. Object storage on CRAQ: High-Throughput chain replication for Read-Mostly workloads. In *2009 USENIX Annual Technical Conference (USENIX ATC 09)*, San Diego, CA, June 2009. USENIX Association.

[15] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Fast distributed transactions and strongly consistent replication for oltp database systems. *ACM Trans. Database Syst.*, 39(2), may 2014.