Towards Achieving Stronger Isolation in Serverless Computing

Saurav Chhatrapati

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-141
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-141.html

May 18, 2021

Copyright © 2021, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Towards Achieving Stronger Isolation in Serverless Computing

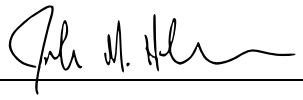
by Saurav Chhatrapati

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

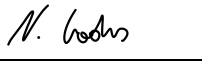


Professor Joseph M. Hellerstein
Research Advisor

May 16, 2021

(Date)

* * * * * * *



Professor Natacha Crooks
Second Reader

May 16th, 2021

(Date)

Towards Achieving Stronger Isolation in Serverless Computing

by

Saurav Chhatrapati

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Masters of Science

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Joseph M. Hellerstein, Chair
Professor Natacha Crooks

Spring 2021

Towards Achieving Stronger Isolation in Serverless Computing

Copyright 2021
by
Saurav Chhatrapati

Abstract

Towards Achieving Stronger Isolation in Serverless Computing

by

Saurav Chhatrapati

Masters of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Joseph M. Hellerstein, Chair

The recent rise in popularity of serverless computing has brought forth new challenges in determining the right consistency model for applications. Previous work has studied how to bring scalable, transactional isolation to serverless computing, but has been insufficient at preventing several consistency anomalies. To address this problem, we would like to guarantee stronger transactional isolation for serverless computing in a scalable manner.

In this thesis we present TASC, a transactional shim for serverless applications. TASC offers flexibility in interposing between most commodity FaaS platforms and cloud storage engines. It providers stronger transactional isolation by guaranteeing snapshot isolation. TASC decouples transaction management from data management and uses modified read and commit protocols to provide scalable snapshot isolation. We demonstrate that TASC has comparable overhead to other serverless consistency shims, and manages to scale to thousands of transactions per second.

To my parents, and my sister.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank everyone who has made this thesis possible through their support and guidance.

First, I would like to thank my advisor, Professor Joe Hellerstein, for providing me with an opportunity to join his research group at the RISE Lab when I was just a first year undergraduate exploring my interests in computer science. He provided constant advice and feedback throughout my time at Berkeley. Through his insightful questions and passion for research he was able to steer my work in the right direction. I would also like to thank Professor Natacha Crooks for her feedback on this thesis.

Vikram Sreekanti was my mentor from the very beginning of my time as a researcher and I am grateful that he took a chance on me. He provided constant guidance and much of this thesis is a result of discussions with him.

Taj Shaik was my partner for my research work over the past year and I would like to thank him for all of the time and effort he has put into this thesis.

Finally, I would like to thank my friends and family for their constant encouragement. I am especially grateful for the support of my parents and my sister, Suhani.

Chapter 1

Introduction

Today, cloud providers offer serverless computing through Functions-as-a-Service (FaaS) platforms. Serverless applications are decomposed into modular components that are chained together through function composition. Functions rely on shared storage for communication, but as the number of chained functions increases, so does the likelihood of exposing partial updates to shared storage due to failure. AWS Lambda and Azure Functions address this problem by retrying the entire function composition on failure, and also allow clients to send retry requests when they experience a timeout [2, 15]. Since functions might be executed multiple times, developers must proactively address this *at-least-once* execution guarantee by writing idempotent functions [16]. However, idempotence is not sufficient for correct fault tolerance in distributed systems due to partial updates that can be leaked.

In order to understand the insufficiency, consider a function f that updates two keys, k and j. If function f fails after writing a new version of k, the entire function will be retried and eventually both k and j will get updated. However, a concurrent function g could have issued a read request right after the failure, and would have seen the new version of k and the old version of j. This scenario is defined as a *fractured read*. Therefore, idempotent programs are not sufficient and atomicity becomes necessary for FaaS execution in order to prevent partial updates from being visible.

1.1 Consistency in Serverless Computing

Atomicity can be guaranteed through traditional database transactions that encapsulate serverless functions, but choosing a consistency level that provides high performance, scales up and down in response to a dynamic workload, and prevents critical consistency anomalies becomes challenging [12]. Serializability prevents many consistency anomalies, but is known to have bottlenecks that limit its scalability [5, 6]. Previous work [19] demonstrates that read atomic isolation, introduced in [3], is extremely performant and scalable. However, read atomic isolation still fails to prevent certain consistency anomalies that make it difficult for developers to reason about their applications. We outline these anomalies in §2.4. Thus, we

turn to snapshot isolation, introduced in [4], as it prevents many consistency anomalies in order to intuitively allow programmers to reason about their application and scales reasonably well [18]. We choose to compromise between read atomic isolation and serializability, because of snapshot isolation's widespread adoption in industry by database vendors, which has made it a familiar consistency model for programmers [17].

1.2 Towards Stronger Transactional Isolation

Our goal in this thesis is to provide a performant serverless shim that offers stronger transactional isolation for widely-used FaaS platforms and storage systems. We present TASC, a scalable shim for serverless computing that enforces snapshot isolation. TASC can be interposed between most commodity FaaS platforms (e.g. AWS Lambda, Azure Functions) and cloud storage systems (e.g. AWS DynamoDB, AWS S3, Google Cloud BigTable). We make no consistency assumptions about the underlying storage layer, but do require persistence. The contributions of this thesis are the following:

- The design of TASC, a low-overhead, scalable serverless shim that enforces snapshot isolation and is flexible enough to work with several commodity compute and storage offerings.
- A new combination of read atomic and snapshot isolation protocols to guarantee snapshot isolation for shared, distributed storage systems.
- A modified 2PC protocol that works in an environment with dynamic membership and guarantees atomicity and durability (with snapshot isolation).
- A detailed evaluation of TASC, demonstrating it incurs low latency penalties and scales to thousands of transactions per second.

1.3 Thesis Overview

The rest of this thesis is organized as follows. In chapter §2, we provide a detailed overview of read atomic isolation and snapshot isolation, and compare the consistency anomalies each isolation level proscribes. In chapter §3, we introduce TASC's API, system architecture, and protocols, formally outline its guarantees, and discuss TASC's process for scaling up and down. In chapter §4, we describe how TASC correctly recovers from all failures. Chapter §5 presents an in-depth evaluation of TASC's overhead, scalability, cost, and fault tolerance. Finally, chapter §6 provides a discussion about related and future work.

Chapter 2

Background

In this chapter, we describe prior work on providing atomic fault tolerance through transactions for FaaS systems (§2.1), compare consistency anomalies prevented by read atomic isolation and snapshot isolation (§2.4), and explain the technical challenges for providing snapshot isolation for serverless applications (§2.5).

2.1 Atomicity in Serverless Computing

Prior work in [19] introduced an Atomic Fault Tolerance shim, AFT, that encapsulates multiple function requests into a single transaction to provide fault tolerance for serverless applications without leaking partial updates. AFT has low overhead due to its guarantee of read atomicity, which is a coordination-free isolation level. The AFT system provides important foundations that TASC extends to provide stronger transactional isolation, while maintaining similar low overhead.

2.2 Read Atomic Isolation

Read atomic isolation was introduced by Bailis et al. in [3] and addresses the *fractured read* scenario discussed in §1 by ensuring that transactions do not see partial updates. Bailis et al. define read atomic isolation as: "[A system that] prevents fractured reads anomalies and also prevents transactions from reading uncommitted, aborted, or intermediate data." Bailis et al. continue that a *fractured read* occurs when, "... transaction T_i writes versions x_m and y_n (in any order, with x possibly but not necessarily equal to y), [and] T_j [later] reads version x_m and version y_k, and $k < n$." Read atomic isolation can be achieved without sacrificing scalability, because its enforcement of update atomicity does not require coordination. However, it fails to prevent a number of consistency anomalies, which makes it difficult for developers to understand the execution of their application. We defer discussion of these anomalies to §2.4.

2.3 Snapshot Isolation

Snapshot isolation [4] provides an isolation level that uses multi-versioned data storage to achieve semantics nearly as strong as serializability with increased concurrency. In snapshot isolation, transactions are assigned a begin timestamp when they begin and a commit timestamp when they commit. Snapshot isolation guarantees that when a transaction T_i with a begin timestamp, b_i, performs a read, it sees updates from all transactions T_j with a commit timestamp, c_j, such that $c_j < b_i$. In other words, each transaction reads from a snapshot of the *committed* data at the time of its begin timestamp. Snapshot isolation is typically implemented using multi-version concurrency control (MVCC), which involves creating new versions of each data item at commit time. Thus, reads involve determining the valid version of a data item to read based on the rule above. Additionally, snapshot isolation follows *first-committer-wins*, which means at commit time a transaction aborts if a concurrent transaction has written to any of the same keys. The differences between snapshot isolation and read atomic isolation can best be explained by comparing the consistency anomalies that they prevent. We outline them next in §2.4.

2.4 Comparing Read Atomic Isolation and Snapshot Isolation

Bailis et al. formally state that "A system provides [read atomic isolation] if it prevents fractured reads phenomena and also proscribes phenomena G0, G1a, G1b, G1c (i.e., prevents transactions from reading uncommitted, aborted, or intermediate versions)." The phenomena Bailis et al. refer to are defined by Adya in [1]. In order to prove that snapshot isolation is at least as strong of a consistency level as read atomic isolation, we rely on Adya's formal model:

Definition 1 (Read-Dependency) *Transaction T_j directly read-depends on T_i if transaction T_i writes some version x_i and T_j reads x_i.*

Definition 2 (Antidepends) *Transaction T_j directly antidepends on T_i if transaction T_i reads some version x_k and T_j writes x's next version (after x_k) in the version order.*

Definition 3 (Write-Depends) *Transaction T_j directly write-depends on T_i if T_i writes a version x_i and T_j writes x's next version (after x_i) in the version order.*

Definition 4 (Direct Serialization Graph) *We define the Direct Serialization Graph (DSG) arising from a history H, denoted by $DSG(H)$ as follows. Each node in the graph corresponds to a committed transaction and directed edges correspond to different types of direct conflicts. There is a read dependency edge, write dependency edge, or antidependency edge from transaction T_i to transaction T_j if T_j reads/writes/directly antidepends on T_i.*

The undesirable isolation phenomena are derived from Adya's dependency definitions as follows:

Definition 5 (G0: Write Cycles) *A history H exhibits phenomenon $G0$ if $DSG(H)$ contains a directed cycle consisting entirely of write-dependency edges.*

Definition 6 (G1a: Aborted Reads) *A history H exhibits phenomenon $G1a$ if H contains an aborted transaction T_a and a committed transaction T_c such that T_c reads a version written by T_a.*

Definition 7 (G1b: Intermediate Reads) *A history H exhibits phenomenon $G1b$ if H contains a committed transaction T_i that reads a version of an object x_j written by transaction T_f, and T_f also wrote a version x_k such that $j < k$.*

Definition 8 (G1c: Circular Information Flow) *A history H exhibits phenomenon $G1c$ if $DSG(H)$ contains a directed cycle that consists entirely of read-dependency and write-dependency edges.*

With the formal definitions above, we can continue with our proof.

Theorem 1 *Snapshot isolation provides read atomic isolation.*

Proof. We prove by showing that snapshot isolation prevents each of the phenomena that read atomic isolation also prevents.

G0. Assume that for a history H, its $DSG(H)$ contains a directed cycle consisting entirely of write-dependency edges. This implies that there must exist two transactions T_i and T_j, such that T_i write-depends on T_j and T_j write-depends on T_i. These transactions must have been running concurrently, but snapshot isolation checks for write-write conflicts at commit time and will only allow one of the transactions to commit, typically the one to commit first. Therefore, this is a contradiction and snapshot isolation prevents phenomenon $G0$.

G1a. In snapshot isolation, a transaction T_i can only read versions written by a committed transaction T_j, such that the *commit-timestamp* of T_j is less than the *begin-timestamp* of T_i. Therefore, snapshot isolation prevents phenomenon $G1a$.

G1b. Under snapshot isolation, a transaction T_i can overwrite itself for an object x. Once T_i commits, only its most recent write to x will be made visible to other transactions. This prevents another transaction T_j from reading T_i's intermediate write to x. Therefore, snapshot isolation prevents phenomenon $G1b$.

G1c. Assume that for a history H, its $DSG(H)$ contains a directed cycle consisting entirely of read-dependency and write-dependency edges. Therefore, there must exist two

transactions, T_i and T_j, such that T_i read-depends on T_j and T_j write-depends on T_i. If T_i read-depends on T_j, then T_i must have begun after T_j had committed under the definition of snapshot isolation. However, if T_j write-depends on T_i, T_j must have performed a write of x that was the next version after x_i and thus T_j must have begun after T_i had committed. This is a contradiction, which implies that snapshot isolation prevents phenomena $G1c$.

Since snapshot isolation proscribes all of the phenomena that read atomic isolation proscribes, snapshot isolation guarantees read atomic isolation. □

Not only does snapshot isolation prevent the same consistency anomalies as read atomic isolation, snapshot isolation also prevents certain anomalies that read atomic isolation fails to prevent. The following sections discuss these anomalies (§2.4.1-§2.4.3).

2.4.1 Lost Updates

Consider the following transaction history H_1:

$$\begin{aligned} T_1&\text{: } \mathrm{r}(x_0); \mathrm{w}(x_1) \\ T_2&\text{: } \mathrm{r}(x_0); \mathrm{w}(x_2) \end{aligned}$$

This is considered a lost update, because T_1 and T_2 concurrently modify x, which causes the second committer to overwrite the changes made by the first committer.

Theorem 2 *Read atomic isolation does not proscribe lost updates.*

Proof. Read atomic isolation does not prevent lost updates, because there is no conflict checking that occurs at commit time. □

On the other hand, snapshot isolation prevents lost updates, because at commit time it checks for transactions that concurrently write to the same object, and would abort all except one of the transactions.

2.4.2 Missing Dependencies

Consider the following transaction history H_2:

$$\begin{aligned} T_3&\text{: } \mathrm{r}(x_1)\ \mathrm{w}(y_3) \\ T_4&\text{: } \mathrm{r}(x_0)\qquad\quad \mathrm{r}(y_3) \end{aligned}$$

This is an example of the missing dependencies anomaly. T_3's write of y can be influenced by its read of x. T_4's read of y depends on the value that T_3 wrote for y. Since T_4 read y_3, it misses the dependency of x_1 influencing y_3, since it also read x_0.

Theorem 3 *Read atomic isolation does not proscribe missing dependencies.*

Proof. Constructing the DSG of history H_2 shows that the only edge is a read-dependency edge from T_4 to T_3. Since there are no cycles in $DSG(H_2)$, clearly phenomena $G0$ and $G1c$ can not detect the missing dependencies anomaly. In addition, the lack of aborted reads ($G1a$) and intermediate reads ($G1b$), since T_4 reads y after T_3 has committed, implies that read atomic isolation does not prevent missing dependencies. □

On the other hand, snapshot isolation does protect against missing dependencies by ensuring that all reads come from the same consistent snapshot of the system as of the T.BEGIN-TS of the transaction. Therefore, it would not be possible for T_4 to read y_3 if T_3 had not committed before T_4 began.

2.4.3 Predicate-Many-Preceders

The predicate-many-preceders (PMP) anomaly occurs when a transaction observes different versions resulting from the same predicate read. Consider the following transaction history H_3 with a predicate P.

T_1 : r$\{P : x, y\}$ r$\{P : x, y, z\}$
T_2: w$(z|z \in P)$

When T_1 first performs a predicate read of P, it reads keys $\{x, y\}$, but a write by T_2 where z satisfies the predicate P causes T_1 to read $\{x, y, z\}$ the second time it performs the predicate read.

Theorem 4 *Read atomic isolation does not proscribe predicate-many-preceders.*

Proof. Constructing $DSG(H_3)$ shows that there is an antidependency edge from T_1 to $T2$ and a read-dependency edge from T_1 to T_2. There is no cycle in the DSG, therefore $G0$ and $G1c$ can not detect the predicate-many-preceders anomaly. There are also no aborted reads ($G1a$) or intermediate reads ($G1b$) in H_3, since T_2 is committed before T_1 performs its second read. Therefore, read atomic isolation does not prevent predicate-many-preceders. □

On the other hand, snapshot isolation prevents it by enforcing which versions of a key are visible to each transaction based on its T.BEGIN-TS. This would prevent T_1 from getting two different views of the database when it performs the same predicate read twice.

2.5 Challenges

Committing transaction T under snapshot isolation's involves checking for write-write conflicts with concurrent transactions as described in §2.3. Once T passes the conflict checking, T's writes must be made atomically visible to other transactions to prevent *fractured reads*. In a single machine implementation, T's writes can trivially be made atomically visible to other transactions, since all transactions are processed by the same machine.

However, systems like TASC must scale as the workload changes, which means that multiple machines will be executing transactions at the same time. This introduces two challenges:

1. Commit latency is dependent on how long it takes for *all* machines in the system to check for conflicts and make writes visible to other transactions.

2. A committed transaction's writes must be made atomically visible across *all* machines in the system.

These challenges motivate the design of TASC, which we introduce next in §3.

Chapter 3

Achieving Scalable Transactional Isolation

In this chapter, we discuss how TASC provides stronger isolation for serverless computing through a scalable transactional shim. We first present the system API (§3.1), architecture (§3.3), and protocols (§3.4). Then, we formally state the guarantees TASC provides (§3.5). Finally, we discuss how TASC can autoscale up and down as the underlying serverless application's workload changes dynamically (§3.6).

3.1 API

TASC provides a transactional key-value store API shown in Table 3.1. The typical TASC client is a serverless function or chain of functions (i.e., AWS Lambda invocation(s)). To achieve atomicity, TASC requires clients to encapsulate their storage operations within a transaction.

Start. Clients start a new transaction by calling *StartTransaction()*, which returns a globally unique transaction ID. We refer to the transaction ID as *TID*. It is defined as follows: $\langle uuid, nodeID \rangle$, where *uuid* is a unique local identifier and *nodeID* is a globally unique identifier for the node that generates the *TID*. The node use its local clock to generate a begin timestamp that is used for read (§3.4.1) and commit (§3.4.3) operations to ensure snapshot isolation. TASC does not require clocks to be synchronized for correctness. In §3.3 we discuss which TASC component is responsible for generating the *TID* and begin timestamp.

Get / Put. Clients can read and write with the *Get()* and *Put()* API calls. As a mechanism to ensure transactional isolation, writes are maintained in transaction-private storage by TASC until the transaction commits.

Abort. Clients can decide to abort transactions by calling *AbortTransaction()*. This will

API	Description
`StartTransaction() -> txid`	Starts a new transaction and returns a transaction ID.
`Get(txid, key) -> value`	Retrieves `key` in the context of the transaction keyed by `txid`.
`Put(txid, key, value)`	Performs an update for transaction `txid`.
`AbortTransaction(txid)`	Aborts transaction `txid` and discards any updates made by it.
`CommitTransaction(txid)`	Commits transcation `txid` and persists its updates; only acknowledges after all data and metadata has been persisted.

Table 3.1: TASC offers a simple transactional key-value store API.

lead TASC to discard any writes that have been previously buffered for the transaction.

Commit. Clients must commit transactions by calling *CommitTransaction()*, which atomically makes their writes visible to other transactions.

3.2 Definitions

In our notation, a transaction's T's *TID* is denoted by its subscript; transaction T_i has *TID* i. We say that T_i is newer than T_j if $i > j$. As described in §2.3, under snapshot isolation each transaction is associated with two timestamps: a begin timestamp and a commit timestamp. We refer to T_i's begin timestamp as T_i.BEGIN-TS and its commit timestamp as T_i.COMMIT-TS.

A key without a subscript refers to any version of that key and k_i is a version of k that was written by transaction T_i. Key versions are hidden from users; clients make requests to read and write keys, and TASC determines which versions are compatible with each request. A key version k_i is defined as: $\langle k, T_i.\text{COMMIT-TS}, i\rangle$. Thus, two key versions k_i and k_j are first compared by k_i.COMMIT-TS and k_j.COMMIT-TS (i.e., the commit timestamps of the transactions that wrote the key versions). In the unlikely event when the commit timestamps are equal, TASC compares k_i.TID and k_j.TID (i.e. the *TID*s of the transactions), which are guaranteed to be unique as *TID*'s are globally unique. Each key logically has an initial `NULL` version (which need never be stored) and zero or more non-`NULL` versions written by transactions.

We define a transaction T_i's readset, T_i.READSET, as the set of all key versions read by T_i. Similarly, T_i's writeset, T_i.WRITESET, is the set of all key versions written by T_i if and only if T_i is committed. For a key version k_i we define its cowritten set, k_i.COWRITTEN, as T_i.WRITESET. In other words, a key version's cowritten set is the writeset of the transaction that wrote that key version.

As described earlier in §2.5, atomic visibility is challenging to enforce for snapshot isolation in a distributed environment. In order for TASC to prevent *fractured reads* (§1), we rely on the concept of an *Atomic Readset* defined by Sreekanti et al. in [19], and present it below.

Definition 9 (Atomic Readset) *Let R be a set of key versions read by a transaction T. R is an* Atomic Readset *if* $\forall k_i \in R, \forall m_i \in k_i.cowritten, m_j \in R \Rightarrow j \geq i$.

In other words, consider a transaction T with readset R and assume T wants to read key m. Suppose there is key version k_i previously read by T (i.e. $k_i \in R$) that was cowritten with key version m_i. This means there is a committed transaction T_i that wrote k and m. Thus, T cannot read any version of m that is older than i; otherwise, a *fractured read* would occur. Note that Atomic Readsets only provide a lower bound on the version of a key that a transaction can read. TASC computes upper bounds during the read protocol (§3.4) based on snapshot isolation.

Other than forming Atomic Readsets and enforcing snapshot isolation, TASC offers a couple other useful guarantees for application developers. We briefly define them here.

Read Your Writes. Guaranteeing *read your writes* involves ensuring that transactions read the most recent version of a key it previously wrote. *Read your writes* does not apply if the transaction has not previously written the key.

Repeatable Reads. *Repeatable reads* requires that if a transaction reads key version k_i, all subsequent reads of k should also return k_i (until the transactions writes k, from which point the *read your writes* policy applies).

3.3 System Architecture

The system design for TASC is motivated by the dynamic workloads of serverless applications it is meant to support. In fact, it is crucial for any serverless shim to enable fine-grained tracking of resources in order for an autoscaling policy engine to transparently provision resources [7, 13]. One of TASC's major design principles is the decoupling of the two types of state any transactional store must track: transaction metadata and data. Figure 3.1 shows an overview of the system architecture. The following sections (§3.3.1-§3.3.2) introduce the core components of TASC.

3.3.1 Transaction Manager

Transaction managers are responsible for storing transaction metadata. In order to provide snapshot isolation, transaction managers keep track of the transaction ID, begin timestamp, and commit timestamp for each transaction they are in charge of. When clients make TASC API calls, the requests are initially sent to a transaction manager, which then communicates and coordinates with the other components as necessary. Clients interact with the same

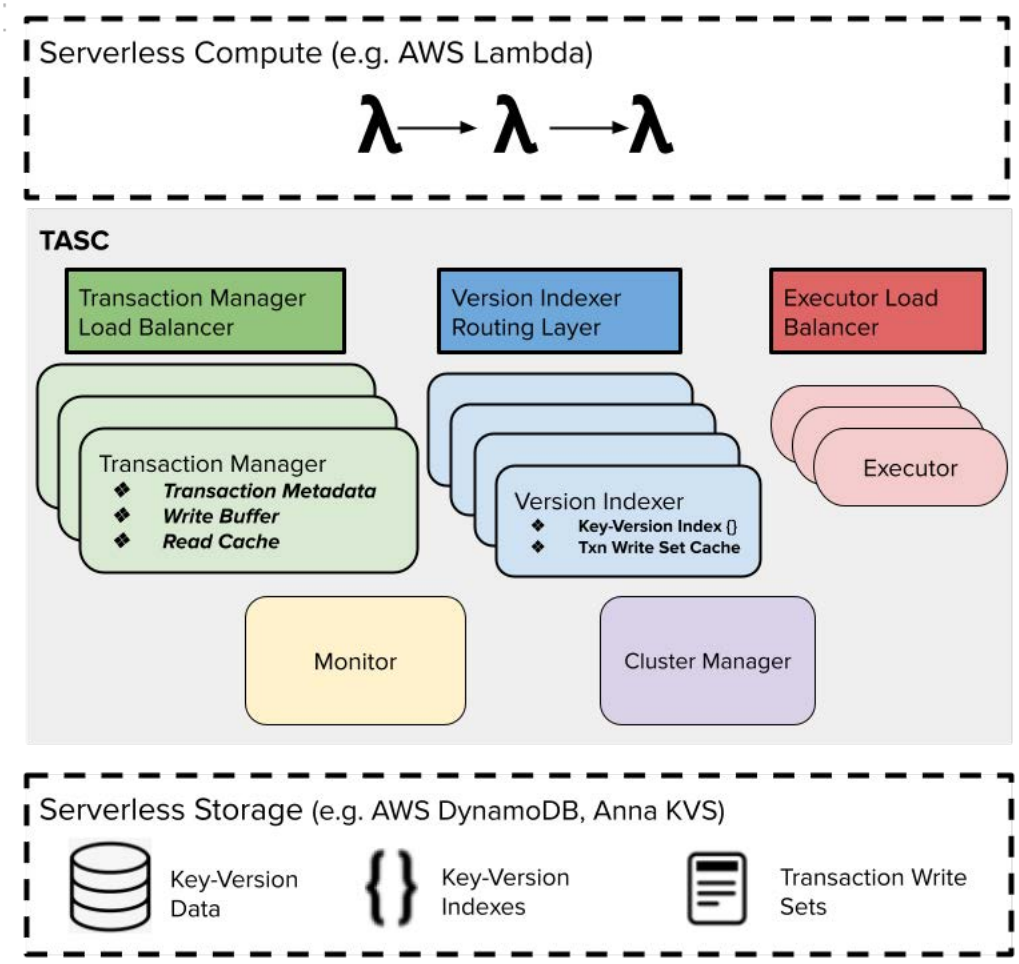


Figure 3.1: TASC system architecture.

transaction manager for the entire transaction. For a transaction T_i, the responsible transaction manager uses a write buffer, T_i.WRITEBUFFER, to hold any writes until T_i commits or aborts. The transaction manager also tracks the transaction's readset, T_i.READSET, and the cowritten set, k_i.COWRITTEN, for each key version k_i in the readset in order to maintain an Atomic Readset as defined above in Definition §9.

Since it can be inefficient to access each k_i.COWRITTEN, transaction managers use a map called the T_i.COWRITTENMAP. For each key version k_j in T_i.READSET, and for each key m in k_j.COWRITTEN, there is a map entry for m in T_i.COWRITTENMAP. The value in the map is the maximum key version of m found across all of the cowritten sets, as this provides the lower bound for the version of m that T_i is allowed to read to ensure atomic readsets. Consider the following example to understand how T_i.COWRITTENMAP is determined.

API	Description
`Read(txid, key) -> keyVersion`	Determines the valid key version for transaction `txid` to read `key`.
`Validate(txid, writeset) -> value`	Determines whether or not transaction `txid` can commit.
`Finish(txid, outcome)`	Finishes transaction `txid` based on its final `outcome`.

Table 3.2: Version Indexer internal API.

Let T_i.READSET $= \{k_1, j_2\}$, k_1.COWRITTEN $= \{a_1, b_1\}$, and j_2.COWRITTEN $= \{b_2, c_2\}$. This means there will be an entry for keys a, b, and c in T_i.COWRITTENMAP. Determining T_i.COWRITTENMAP$[a]$ and T_i.COWRITTENMAP$[c]$ is trivial since there is only one version of each key across all the cowritten sets for a key in T_i.READSET. For T_i.COWRITTENMAP$[b]$, we store the maximum (i.e. newest) version of b found across all of the cowritten sets for a key in T_i.READSET, which is b_2. Thus, T_i.COWRITTENMAP $= \{a : a_1, b : b_2, c : c_2\}$.

3.3.1.1 Transaction Manager Load Balancer

TASC uses a load balancer to load balance *StartTransaction* API calls across the transaction managers. After the transaction manager that handles the *StartTransaction* request returns its address in the response, the client can send subsequent request directly to the transaction manager.

3.3.2 Version Indexer

Version indexers track which versions k_i exist for each key k. They maintain a committed key version index and a pending key version index to track which key versions exist for committed and pending transactions. Committed transactions have completed the commit process, while pending transactions are still undergoing the commit process. The version indexers use this information to perform two important task. The first task is to determine the key version a transaction T_i should read based on T_i.BEGIN-TS and T_i.READSET. The second task is to validate whether or not a transaction can commit based on snapshot isolation's rules described in §2.3. Table 3.2 shows the internal version indexer API, which is used by other TASC components during the read and commit protocols discussed in §3.4

3.3.2.1 Version Indexer Routing Layer

In order to minimize the number of nodes that are involved in the commit protocol (§3.4), TASC shards the key version indexes across the version indexers. Our TASC implementation uses the Anna KVS [21] hash ring to perform hash partitioning. The hash ring allows for

simple routing queries to lookup which version indexer is in charge of a particular key. For the remainder of this thesis, we refer to the version indexer routing layer as the VIRL.

3.3.3 Executor

Executors are a stateless component that receive transaction outcomes (i.e. commit or abort) from transaction managers and notify the relevant version indexers. We defer further discussion about executors to §3.4.3.

3.3.4 Monitor and Cluster Manager

The monitor is responsible for collecting various performance metrics from each component in the cluster, which it then provides to the cluster manager. The cluster manager uses the statistics from the monitor to feed to its autoscaling policy engine, which determines when TASC components should be added or removed from the cluster. The cluster manager is then responsible for overseeing the scaling process, which we discuss further in §3.6.

3.4 Protocols

In this section we present details on how TASC performs reads (§3.4.1), writes (§3.4.2), commits (§3.4.3), and aborts (§3.4.4). When a client makes the relevant API call for transaction T_i, the request is sent directly to the transaction manager that handled the *StartTransaction* API call for T_i.

3.4.1 Read Protocol

When the transaction manager receives a read request for transaction T_i and key k, it first checks if $k \in T_i$.WRITEBUFFER (i.e. *read your writes* applies) or $k \in T_i$.READSET (i.e. *repeatable reads* applies). If *read your writes* applies, the transaction manager returns the value stored in T_i.WRITEBUFFER. If *repeatable reads* applies, the transaction manager must fetch the key version found in T_i.READSET from storage. As an optimization, transaction managers can maintain a read cache to improve the performance of *repeatable reads*.

When neither of the above cases applies, the transaction manager sends a read request to the appropriate version indexer. The transaction manager first uses the VIRL to lookup the version indexer responsible for key k. As an optimization, transaction managers can store an VIRL cache. In §3.6 we discuss when and how an VIRL cache would need to invalidate entries in the event that version indexers join or leave the cluster. Before sending a read request to the version indexer, the transaction manager uses T_i.COWRITTENMAP to determine if there exists a lower bound for the version of k that T_i can read based on Definition 9. Then, the transaction manager sends a read request to the version indexer with the following parameters: k, T_i.READSET, T.BEGIN-TS, and the lower bound if it exists.

The version indexer takes the read request parameters to form the lower bound (if it exists) and the upper bound. The upper bound is T_i.BEGIN-TS, as snapshot isolation requires that T_i can only read updates from a committed transaction T_j, such that T_j.COMMIT-TS $< T_i$.BEGIN-TS. The version indexer then uses k.COMMITTED-KVI, which is the set of key versions of k found in the committed key version index (§3.3.2), to determine the key versions k_j that exist within the bounds. Version indexers return the most recent version of k that falls within the bounds.

Suppose the version indexer finds a candidate key version k_a that falls within the computed bounds. The version indexer must also verify that T_i.READSET $\cup\ k_a$ is an Atomic Readset per Definition 9. This involves ensuring that $\nexists m_a \in k_a$.COWRITTEN, such that $m_b \in T_i$.READSET and $a < b$. We show below that there always exists a key version k_i that satisfies the upper bound and Atomic Readset requirements [1].

Theorem 5 *Given $k \notin T_j$ and T_j, $\exists k_i$, such that (1) T_i.COMMIT-TS $< T_j$.BEGIN-TS and (2) T_j.READSET $\cup\ k_i$ is an Atomic Readset.*

Proof. Assume by induction that T_j.READSET is an Atomic Readset. Every key k in TASC has an initial key version $k_{\texttt{NULL}}$. Condition (1) is guaranteed since $T_{\texttt{NULL}}$.COMMIT-TS $< T_j$.BEGIN-TS. Condition (2) is guaranteed by showing that $k_{\texttt{NULL}}$ satisfies the Atomic Readset requirement from Definition 9. The requirement $\forall m_{\texttt{NULL}} \in k_{\texttt{NULL}}$.COWRITTEN, $m_l \in T_j$.READSET $\implies l \geq$ NULL is trivially held true, since $k_{\texttt{NULL}}$.COWRITTEN $\in \emptyset$. Thus, T_j.READSET $\cup\ k_{\texttt{NULL}}$ is an Atomic Readset and both conditions are satisfied. □

On the other hand, due to the challenges presented in §2.5 regarding implementing snapshot isolation in a distributed environment, it is possible that there does not exist a key version, k_i that satisfies the lower bound provided in the read request. Version indexers address this case by blocking the request and periodically checking to see if newer key versions have been added to k.COMMITTED-KVI. The indexer will finally timeout and return NULL. We leave the decision to retry the read request to the client. We revisit this scenario in §3.5.4 and present a mitigation strategy.

Once the version indexer has identified a valid key version k_i, it returns the following information: k_i and k_i.COWRITTEN. The transaction manager uses this information to update T_i.COWRITTENMAP as necessary. For each $m_i \in k_i$.COWRITTEN, the transaction manager updates T_i.COWRITTENMAP if $m \notin T_i$.COWRITTENMAP $\vee\ T_i$.COWRITTENMAP$[m] < m_i$. In other words, the transaction manager adds m_i if no other key in T_i.READSET was cowritten with m or if m_i is newer than the existing key version for m in T_i.COWRITTENMAP. Finally, the transaction manager uses k_i to fetch the value from storage and returns it to the client [2]. As an optimization, the transaction managers maintain a read cache to improve

[1] Note that for efficiency, TASC does not track k_i.COWRITTEN for each key k; instead, indexers simply use T_i.WRITESET to determine k_i.COWRITTEN. As an optimization, version indexers maintain a transaction writeset cache.

[2] Recall that TASC clients have no notion of a key version, and simply receive a value (or NULL) in response to *Get* API calls.

Algorithm 1 TransactionManagerRead: For a key k and transaction T_i, return the value for k_j such that snapshot isolation is enforced and that the readset R combined with k_j does not violate Definition 9.

Input: $k, R, WriteBuffer, CowrittenMap, BeginTS, storage$

```
1: if k ∈ WriteBuffer then // read-your-write applies
2:     return WriteBuffer[k]
3: if k ∈ R then // repeatable-reads applies
4:     return storage.Get(R[k])
5: // Determines lower bound for T_i to read k
6: lower := NULL
7: if k ∈ CowrittenMap then
8:     lower = CowrittenMap[k]
9: k_target, k_target.COWRITTEN := IndexerRead(k, R, BeginTS, lower)
10: // Updates CowrittenMap to determine future lower bounds correctly
11: for m_j ∈ k_target.COWRITTEN do
12:     if m ∉ CowrittenMap ∨ m_j > CowrittenMap[m] then
13:         CowrittenMap[m] = m_j
14: if k_target == NULL then
15:     return NULL
16: // Update readset
17: R = R ∪ k_target
18: return storage.Get(k_target)
```

read performance. The entire read algorithm that takes place at the transaction manager and version indexer is shown in Algorithms 1 and 2 respectively.

3.4.2 Write Protocol

When a transaction manager receives a write request for transaction T_i with key k and value v, it simply adds the write to T_i.WRITEBUFFER, overwriting a previous write of k by T_i if one already exists. T_i's writes stay in T_i.WRITEBUFFER on the transaction manager until T_i commits (§3.4.3) or aborts (§3.4.4). If the transaction manager fails before T_i commits, T_i will abort and all writes will be lost, requiring the client to redo the entire transaction. We defer discussion of failure and recovery to §4.

3.4.3 Commit Protocol

When a transaction manager receives a commit request for transaction T_i, it first generates a timestamp for T_i.COMMIT-TS. Committing read-only transactions (i.e. T_i.WRITESET $\in \emptyset$) is trivial, as there are no updates to make visible to other transactions in the system. Therefore, the transaction manager can simply return COMMITTED to the client and mark T_i's metadata for garbage collection (§6.1). For transactions that perform writes, the transaction manager

Algorithm 2 VersionIndexerRead: For a key k and transaction T_i, return the key version k_j and its cowritten set such that snapshot isolation is enforced and that the readset R combined with k_j does not violate Definition 9.

Input: $k, R, BeginTS, lower, storage$

```
committedVersions := k.COMMITTED-KVI
candidateVersions := sort(filter(committedVersions, kv ≥ lowerBound))
for v ∈ candidateVersions.reverse() do
    if v.commitTS ≥ beginTS then
        continue
    valid := True
    for m_j ∈ k_v.cowritten do
        if m_j ∈ R ∧ j < v then
            valid = False
            break
    if valid then
        return k_v, k_v.COWRITTEN
return NULL, ∅ // No valid versions found
```

must begin a commit protocol with the relevant version indexers to determine whether or not T_i can commit. The transaction manager first queries the VIRL to determine the version indexer responsible for each key k in T_i.WRITEBUFFER. As mentioned in §3.4.1, the transaction managers can maintain an VIRL cache, which requires additional maintenance discussed in §3.6.

After having determined the complete set of relevant indexers, the transaction manager acts as the coordinator in a Two Phase Commit (2PC) protocol with the version indexers as participants [10]. 2PC requires that the set of participants does not change one the protocol begins, but due to TASC's autoscaling feature, it is possible that the set of version indexers in the cluster changes during the commit process. We defer discussion about how TASC achieves correctness despite a dynamically changing set of version indexers to §3.6.3. As described in §3.3.2.1, the decoupling of transaction metadata and key version metadata across transaction managers and indexers, along with the sharding of key version indexes across the version indexers, allows TASC's commit protocol to scale by pruning the participant set to only the relevant version indexers. A naive snapshot isolation implementation would require consensus of all nodes in the cluster to commit a transaction: data storage nodes as well as coordination handling nodes.

3.4.3.1 Phase One

In the first phase of commit, the transaction manager sends a validate request to the relevant version indexers with the following parameters: TID, T_i.BEGIN-TS, T_i.COMMIT-TS, and K_x. We define K_x as the set of relevant keys for the version indexer I_x (i.e. all keys $k \in T$.WRITESET, such that I_x is responsible for k).

The version indexer validates if T_i can commit by checking for concurrent write conflicts as described in §2.3. For each key $k \in K_x$, the version indexer checks each key version k_j in k.PENDING-KVI and k.COMMITTED-KVI. T_i cannot commit if $\exists k_j$, k_j.COMMIT-TS $\geq$ T_i.BEGIN-TS $\wedge$ k_j.COMMIT-TS $\leq$ T_i.COMMIT-TS. In other words, T_i cannot commit if a concurrent transaction T_j writes to a key in T_i.WRITESET and attempts to commit first. If the indexer finds no conflicts, for each key $k \in K_x$, it inserts k_i into k.PENDING-KVI and writes the updated k.PENDING-KVI to storage for persistence. Note that version indexers use locks on the pending and committed key version indexes to ensure atomicity across validation requests. We discuss in §4 why it is important to persist pending and committed version indexes to storage when they are modified. Once k.PENDING-KVI has been written to storage, the version indexer can respond to the transaction manager with a YES vote. Otherwise, if the version indexer finds a conflict, it does not need to modify (or write to storage) k.PENDING-KVI and responds to the transaction manager with a NO vote. The version indexer validation algorithm for the commit protocol is shown in Algorithm 3.

The transaction manager blocks until it receives all YES votes from the indexers or until it receives the first NO vote. The transaction manager will also decide to abort if it reaches a timeout waiting for a response from an indexer. We defer discussion about how indexers correctly recover from failure to §4.

3.4.3.2 Phase Two

Commit Optimization. In traditional 2PC, the coordinator collects responses from the participants and then sends out another round of messages to all participants about the final outcome of the transaction. The coordinator then waits for an ACK from each of the participants before notifying the client about the transaction outcome. While Phase 1 of TASC's commit protocol involves all necessary participants, TASC delegates Phase 2 to a single executor, which was introduced earlier in §3.4.3.2. The motivation for replacing all participants (i.e. relevant version indexers) with a single executor is to minimize the communication and coordination cost.

TASC is able to use this modified Phase 2 due to a few key observations that have to do with snapshot isolation semantics. The transaction manager can respond to the client that a T_i has COMMITTED or ABORTED once the following steps occur: 1) relevant version indexers validate T_i, 2) T_i.WRITEBUFFER is persisted to storage (if T_i COMMITTED), 3) T_i.OUTCOME is persisted to storage, and 4) *eventually*, yet definitively, version indexers are notified about T_i.OUTCOME, causing them to persist their updated committed (and pending) version indexes, which allow T_i's updates to be made visible to other transactions (if T_i COMMITTED). Note that Step 4 is not actually required by snapshot isolation, but is a liveness guarantee that ensures freshness of reads. Step 1 is guaranteed to be complete once all Phase 1 responses are received from the version indexers. Steps 2 and 3 are guaranteed to be complete once the appropriate writes are made at the storage layer. Step 4 provides us with some flexibility, due to the keyword *eventually*.

Consider the following to understand why TASC's consistency guarantees are not dependent on how long it takes for T_i's updates to be made visible to other transactions. We refer to this time to visibility as T_i.TT-VISIBILITY. Snapshot isolation does not prevent transactions from reading stale data. In fact, it would be correct for all transactions to read $k_{\texttt{NULL}}$ for each key k, since that would trivially satisfy snapshot isolation and Atomic Readset requirements, as stated in Theorem 5. However, such a system with T_i.TT-VISIBILITY $= \infty$ would not be useful for any meaningful application. Therefore, TASC aims to minimize T_i.TT-VISIBILITY, while taking advantage of this relaxed constraint.

Executor. Executors act as de facto coordinators for the second round of TASC's commit protocol. Each executor maintains a list of transactions it is working on. We refer to this list as the ACTIVE-LIST and it is persisted at the storage layer. The location of each executor's ACTIVE-LIST is configured when the executor starts and it remains fixed across failures. No TASC component or the client needs to know which executor is in charge of a particular transaction. In fact, the executors are stateless and can be autoscaled at will. We defer discussion about scaling executors to §3.6.2 and discuss their fault tolerance in §4.2.3.

Notifying the Client. Once the transaction manager decides on T_i.OUTCOME and has completed Step 3 from above, it sends a finish request to an executor[3] with the following parameters: *TID*, T_i.OUTCOME, and T_i.WRITESET. The executor appends *TID* to the ACTIVE-LIST, writes the updated ACTIVE-LIST to storage, and responds with an ACK. At that point, the transaction manager responds to the client with T_i.OUTCOME.

The client does not observe the latency of the rest of the actions the executor takes for T_i, as they are off of the critical path. The transaction manager writes T_i.COMPLETE to storage, but this can occur after the client has been notified about T_i.OUTCOME. T_i.COMPLETE indicates that the executor has ACKed the finish request, and is only used to speed up the process for recovering from failure (§4.2.1).

Finishing the Transaction. After sending an ACK to the transaction manager, the executor will perform its own lookup using the VIRL to determine which version indexers are relevant for T_i. This lookup is necessary as the VIRL's hash ring may have changed due to version indexers joining or leaving the cluster. We defer discussion on version indexer membership change to §3.6. The executor then sends a finish request to each version indexer with the following parameters: *TID*, T_i.OUTCOME, K_x, and T_i.WRITESET (if the outcome is COMMITTED). The definition for K_x is the same as the one provided in round one of the commit protocol (§3.4.3.1).

When a version indexer receives a finish request with T.OUTCOME = COMMITTED, $\forall k_i \in K_x$, it copies k_i from k.PENDING-KVI to k.COMMITTED-KVI and writes the updated k.COMMITTED-KVI to storage. At that point, T_i's writes are visible to other transactions,

[3] TASC has an executor load balancer to load balance finish requests from transaction managers to the executors.

Algorithm 3 Validate: Given transaction T_i and T_i.WRITESET, returns whether or not T_i can commit.

Input: $WriteSet, BeginTS, CommitTS, storage$

```
// Check for write-write conflicts with pending and committed transactions
for k ∈ WriteSet do
    versions := k.PENDING-KVI ∪ k.COMMITTED-KVI
    versions = filter(versions, v.commitTS ≥ BeginTS ∧ v.commitTS ≤ CommitTS)
    if versions ≠ ∅ then
        return False
// No conflicts found
for k ∈ WriteSet do
    insert(k.PENDING-KVI, k_i) // Mark T_i's writes as pending
    storage.Put(k + "pkvi",k.PENDING-KVI) // Write pending index to storage
return True
```

which concludes step 4 defined above. Note that the read protocol described in §3.4.1 ensures that read atomic sets are maintained and snapshot isolation is guaranteed even if version indexers stall for a long time. The version indexer will also add T_i.WRITESET to its transaction writeset cache, as described in §3.4.1. For both cases T.OUTCOME = COMMITTED and T.OUTCOME = ABORTED, $\forall k_i \in K_x$, the version indexer removes k_i from k.PENDING-KVI and writes the updated k.PENDING-KVI to storage. The version indexer finally responds with an ACK.

Once the executor receives an ACK from all version indexers, it removes *TID* from the ACTIVE-LIST and writes the updated ACTIVE-LIST to storage. The executor will keep resending finish requests to version indexers it does not receive an ACK from, which ensures Step 4 from above.

3.4.4 Abort Protocol

When a transaction manager receives an abort request for transaction T_i, it simply changes the status of T_i from RUNNING to ABORTED, in order to prevent clients from unintentionally making subsequent operations on aborted transactions. All of the transaction metadata for T_i can be safely deleted either immediately or via a background garbage collector along with any writes in T_i.WRITEBUFFER, which will not be visible to any other transaction.

3.5 Guarantees

In this section, we formally present the guarantees TASC provides (§3.5.1-§3.5.3). We also discuss a potential downfall for reads and offer a mitigation strategy (§3.5.4).

3.5.1 Preventing Dirty Reads

TASC prevents *dirty reads* by ensuring that writes are visible only after commit; before commit they are buffered at the transaction manager. Line 1 from Algorithm 2 ensures that for a key k, transactions will only read committed versions of k that are found in k.COMMITTED-KVI.

Users may be concerned about the memory requirements to buffer all transaction writes at the transaction manager, especially for long-running transactions with large writes (as TASC does not place any restriction on the size of the write). However, TASC can proactively write intermediary data to storage without any concern that *dirty reads* will occur. Key version k_i can be identified in storage by knowing k and T_i's *TID*. In fact, as an optimistic optimization, data can be preemptively written to storage to minimize the time the transaction manager must wait for storage writes at commit time.

3.5.2 Preventing Fractured Reads

Algorithms 1 and 2 show the read protocol, which prevents *fractured reads*.

Theorem 6 *Given k, $R = T_i$.READSET, and $R_{new} = T_i$.READSET after Algorithms 1 and 2 are executed, R_{new} is an Atomic Readset, as defined in Definition 9.*

Proof. We prove by induction on the size of R.

Base Case. Before the first read is issued, R is empty and trivially an Atomic Readset. After Algorithms 1 and 2 are invoked for the first read, R_{new} contains a single key version, k_{target}, so Theorem 6 holds; R_{new} is an Atomic Readset.

Inductive hypothesis. Let R be an Atomic Readset up to this point in the transaction, and let k_{target} be the key version found by Algorithm 2. We show that R_{new} is also an Atomic Readset. We must show that (1) $\forall l_i \in R, k_i \in l_i.cowritten \Rightarrow target \geq i$, and (2) $\forall l_{target} \in k_{target}.cowritten, l_i \in R \Rightarrow i \geq target$.

Line 8 of Algorithm 1 and line 2 of Algorithm 2 ensure (1), because the lower bound of *target* is computed from the largest *TID* in R that modified k. Lines 7-10 of Algorithm 2 check if each version satisfies case (2). We iterate through all of the cowritten keys of each candidate version. If any cowritten key is in R, we declare the candidate version valid if and only if the cowritten key's version is *not newer* than the version in R. If there are no valid versions, we return `NULL` □

3.5.3 Preventing the Same Consistency Anomalies as Snapshot Isolation

TASC prevents the same consistency anomalies as snapshot isolation, which we outlined previously in §2.3, by enforcing snapshot isolation's rules for reads and commits. Lines 4-5 of Algorithm 2 show that transaction T_j will read key version k_i if T_i.COMMIT-TS <

T_j.BEGIN-TS. Lines 2-6 of Algorithm 3 show that a transaction T_i will only commit if there is not a concurrent transaction T_j (i.e. T_j.COMMIT-TS $\geq T_i$.BEGIN-TS $\wedge$ T_j.COMMIT-TS $\leq$ T_i.COMMIT-TS), where key $k \in T_i$.WRITESET $\wedge$ $k \in T_j$.WRITESET.

3.5.4 Read Liveness Downfall

As mentioned in §3.4.1, it is possible that an indexer is unable to find a valid key version k_i that satisfies the lower bound provided in the read request. Consider the following example.

Suppose initially that k.COMMITTED-KVI $= \{k_0\}$ and m.COMMITTED-KVI $= \{m_0\}$[4]. Assume that indexer I_k is responsible for k and I_m is responsible for m. Transaction T_1 writes to k and m and is able to commit. Therefore, an executor will send a finish request to I_k and I_m to commit T_1, which involves moving the key version from the pending key version index to the committed key version index for the respective key. Another transaction T_2, with T_2.READSET $= \emptyset$, reads k and then reads m. We consider two cases:

(1) If I_k processes T_2's read request before I_k processes the finish request for T_1, T_2 will end up reading k_0.

(2) Otherwise, T_2 will read k_1.

If case (1) applies, T_2 must read m_0, because $\{k_0, m_1\}$ does not form an Atomic Readset as defined in Definition 9. Likewise, if case (2) applies, T_2 must read m_1, because $\{k_1, m_0\}$ does not form an Atomic Readset. The problem arises in case (2) if I_m has not yet received or processed the finish request for T_1. In that case, I_m will find that m.COMMITTED-KVI $= \{m_0\}$. Therefore, as described in §3.4.1, I_m will have to block until m_1 is added to m.COMMITTED-KVI, or it must return that no valid versions were found for m (i.e. it returns `NULL`).

While this scenario is very unlikely in practice when the system is running without any failures, the probability increases when indexers start to fail. Therefore, it is necessary to minimize the recovery time for indexers. We defer discussion to §4 for the steps that need to take place to detect indexer failure and to restart the node. An optimization that can be made is to have standby-by replicas of indexers that are ready to immediately take over in the event of failure. This would add overhead to the commit protocol, by requiring that updates to the pending and committed key version indexes are made persistent to both storage and the replica indexers, but would make reads more resilient to this downfall.

3.6 Scalability

TASC is intended to be used by serverless applications that have dynamic workloads; therefore, it is necessary to discuss how TASC scales up (and down). We consider when and how

[4]Note that each key has an initial version `NULL` that we do not show here.

to scale each component of TASC in §3.6.1-§3.6.3. In our implementation, the cluster manager uses a naive autoscaling policy. Different autoscaling policies can be plugged in, but discussion about autoscaling policies is beyond the scope of this thesis.

3.6.1 Scaling Transaction Managers

As the number of transactions or the frequency of operations increases in TASC, new transaction managers can be added to the cluster. Adding transaction managers does not require any coordination, as it only involves notifying the transaction manager load balancer about the new node. Once the new transaction manager is added to the load balancer, new transactions will be sent to it. Scaling down the number of transaction managers is more challenging, because shutting down a node with active transactions would result in them getting aborted. Therefore, the first step to take to remove transaction manager M from the cluster is to remove it from the transaction manager load balancer. Once that occurs, no new transactions will be sent to M. Then, M can be safely shut down once all of its active transactions have completed. The cluster manager logs any soft state to the storage layer in case it fails while waiting for the transaction manager to complete its active transactions.

3.6.2 Scaling Executors

Scaling the number of executors simply involves starting a new executor and adding it to the executor load balancer described in §3.4.3. At that point, transaction managers will begin sending finish requests to the new executor. Scaling down the number of executors is similar to the equivalent process for transaction managers. Removing executor E from the cluster first involves removing E from the executor load balancer. Then, E can be shut down once its ACTIVE-LIST becomes empty. The cluster manager logs any soft state to the storage layer in case it fails while waiting for the executor to have an empty ACTIVE-LIST.

3.6.3 Scaling Version Indexers

The indexers are the most difficult component of TASC to scale, because there is a need for coordination. It is important to understand how the VIRL performs routing to the indexers to understand the steps that are necessary to add or remove a version indexer. As described in §3.3.2.1, the VIRL consists of a hash ring. The hash ring ensures that if I_1 and I_2 are part of the hash ring before any changes, there does not exists a key k, such that I_1 is responsible for k before the change and I_2 is responsible for k after the change. This is important as it minimizes the amount of ownership change that occurs for version indexers with respect to the keys they are in charge of. We now consider how ownership of keys changes when indexers are added to and removed from the cluster.

Let X be the set of version indexers in the cluster before any changes. When version indexer I_j is added to the VIRL's hash ring, it is possible $\forall I_i \in X,\ \exists$ key k, such that I_i was

responsible for k before the hash ring changed and I_j is responsible for k after the hash ring changed.

Again, let X be the set of version indexers in the cluster before any changes. When version indexer I_j is removed from the VIRL's hash ring, it is possible $\forall I_i \in X$, $\exists$ key k, such that I_j was responsible for k before the hash ring changed and I_i is responsible for k after the hash ring has changed.

Since the hash ring must change when version indexers are added to or removed from the cluster, we introduce an epoch number. The epoch number is provided to each version indexer when it starts, as well as to the the VIRL. When transaction managers make a routing lookup request to the VIRL, they are provided the current epoch number, and store it with entries in the router cache if the optimization described in §3.4.1 is applied. In addition, when transaction managers make requests to the version indexers, they must provide the epoch number returned by the VIRL for the corresponding routing lookup request. We also introduce the following rule at the version indexers that if the version indexer's epoch number is greater than the request's epoch number, the version indexer immediately responds with INVALID EPOCH. This allows transaction managers to invalidate routing cache entries from previous epochs. If the request's epoch number is greater than the version indexer's epoch number, the version indexer simply drops the request. Finally, each version indexer also tracks how many active requests they have for each epoch.

The following steps are taken when a new version indexer I is added to the cluster. The cluster manager logs any soft state to the storage layer in case it fails during the process.

1. The cluster manager creates a new version indexer I, which is configured with the current epoch number.
2. The cluster manager sends $\langle \text{EPOCH CHANGE}, \epsilon \rangle$ to the VIRL to change the epoch number to ϵ.
3. The cluster manager sends $\langle \text{ADD INDEXER}, I \rangle$ to the VIRL, which then adds I to the hash ring.
4. The cluster manager sends $\langle \text{EPOCH CHANGE}, \epsilon \rangle$ to all current version indexers to change the epoch number to ϵ.
5. The cluster manager waits for all current version indexers to finish all active requests from the previous epoch number.
6. The cluster manager sends $\langle \text{EPOCH CHANGE}, \epsilon \rangle$ to I. At this point, I will begin processing requests, as earlier, it had an epoch number smaller than any requests that may have been sent to it.

Theorem 7 *Given key k and a version joining indexer I, such that version indexer I_k is responsible for k before I is added to the* VIRL *hash ring and I is responsible for k after I is added to the* VIRL *hash ring, it is not possible for I_k and I to simultaneously process a request for k.*

Proof. Let the previous epoch number be ϵ_0 and the new epoch number be ϵ_1. The ϵ_0 hash ring will route requests for k to I_k, while the ϵ_1 hash ring will route requests to I. There are two cases to consider:

1. I_k receives a request for k before step 4. In this case, I_k is guaranteed to have finished processing all such requests for k by the end of step 5.

2. I_k receives a request for k after step 4. In this case, I_k now has an epoch number of ϵ_1, but these requests have an epoch number of ϵ_0 (otherwise they would not have been routed to I_k); therefore, I_k will simply drop the requests as described above.

Since I_k is guaranteed to finish processing all of its requests for k by the end of step 5, and I will not begin processing requests for k (or any other key) until step 6, it is guaranteed that I_k and I will not simultaneously process a request for k. □

The following steps are taken when a version indexer I_k is removed from the cluster. The cluster manager logs any soft state to the storage layer in case it fails during the process.

1. The cluster manager sends $\langle \text{EPOCH CHANGE}, \epsilon \rangle$ to the VIRL to change the epoch number to ϵ.

2. The cluster manager sends $\langle \text{REMOVE INDEXER}, I_k \rangle$ to the VIRL, which then removes I_k from the hash ring.

3. The cluster manager waits for I_k to finish all active requests from the previous epoch number.

4. The cluster manager sends $\langle \text{EPOCH CHANGE}, \epsilon \rangle$ to all indexers (except I_k) to change the epoch number to ϵ.

5. The cluster manager shuts down I_k.

Theorem 8 *Given key k and a departing version indexer I_k, such that version indexer I_k is responsible for k before I_k is removed from the* VIRL *hash ring and I_j is responsible for k after I_k is removed from the* VIRL *hash ring, it is not possible for I_k and I_j to simultaneously process a request for k.*

Proof. Let the previous epoch number be ϵ_0 and the new epoch number be ϵ_1. The ϵ_0 hash ring will route requests for k to I_k, while the ϵ_1 hash ring will route requests to I_j. There are two cases to consider:

1. I_j receives a request for k before step 4. In this case, I_j still has epoch number ϵ_0, but the request has epoch number ϵ_1, so the request is dropped.

2. I_j receives a request for k after step 4. In this case, I_j processes the request, because its epoch number is now ϵ_1 and is equal to that of the request.

Since I_k is guaranteed to finish processing all of its requests for k by the end of step 3, and I_j will not begin processing requests for k until step 4, it is guaranteed that I_k and I_j will not simultaneously process a request for k. □

Theorems 7 and 8 guarantee that for each key k, only one version indexer will be responsible for it at a time, which ensures the safety and correctness of TASC's scaling process for indexers.

Chapter 4

Fault Tolerance

Now, we turn to guaranteeing fault tolerance for the TASC protocols introduced previously in §3. To prove safety we must ensure that TASC prevents *dirty reads*, *fractured reads*, and reads in violation of snapshot isolation in the presence of failure, and durability is guaranteed. TASC prevents *dirty reads* by ensuring that writes are visible only after commit; before commit they are buffered at the transaction manager. TASC prevents *fractured reads* and ensures all reads follow snapshot isolation through the read protocol described in §3.4.1. Demonstrating durability is more involved; therefore, we dedicate the rest of this chapter to proving that once a client learns that T_i has committed, T_i's updates are guaranteed to be persistent and visible to other transactions.

4.1 Fault Detection

Before discussing the various failure cases that can occur in TASC, we introduce the process used to detect failure. TASC is deployed using Kubernetes [14], a cluster management tool with its own fault detector. We rely on Kubernetes' default policy of restarting failed pods for all core TASC components: transaction managers, version indexers, and executors. We defer studying the impact of failure detection and recovery on TASC's performance to §5.

4.2 Failure Cases and Correctness

In order to prove durability, we consider failures at each component: transaction manager, version indexer, and executor in the following sections (§4.2.1-§4.2.3).

4.2.1 Transaction Manager Failures

We rely on the finite state machine diagram shown in Figure 4.1 to guide our discussion about transaction manager failure and recovery. One of the assumptions we make in TASC is that if the transaction manager responsible for transaction T_i fails before the client is

able to send a *Commit* API call, all state for T_i is lost and the transaction is aborted. Clients can detect that a transaction manager has failed either through a broken connection, since our implementation uses gRPC [11] for communication between clients and transaction manager, or through an observed timeout. If the failure is detected during a *Commit* API call, the client must initiate the Transaction Resolution Process, which we refer to as the TRP.

The TRP involves the following steps:

1. The client detects that transaction manager M_i has failed during the *Commit* API call for transaction T_i.
2. The client contacts the transaction manager load balancer (§3.3.1.1) to discover a new transaction manager M_j.
3. The client sends an inquiry request to M_j to learn about the outcome of T_i.
4. M_j checks if T_i.OUTCOME has been written to storage (completion of Step 3 in Figure 4.1), and if it does it checks if T_i.COMPLETE has been written to storage (completion of Step 5 in Figure 4.1). One of the following three cases will apply, and we identify which states from Figure 4.1 they correspond to.
   a. If T_i.OUTCOME does not exist, that means T_i must be aborted, because M_i did not finish writing T_i.WRITESET to storage. Therefore, M_i did not send any finish request to an executor for T_i. As a hint for future resolution efforts, M_j will write T_i.OUTCOME to storage with the value ABORTED, and to recover resources M_j will send a finish request to an executor for T_i. Once the ACK from the executor is received, M_j will write T_i.COMPLETE to storage before notifying the client that T_i has aborted.
      This case corresponds to failure during states 1-3.
   b. If T_i.OUTCOME exists, but T_i.COMPLETE does not exist, it is possible that M_i failed before sending a finish request to an executor. Therefore, M_j will send a finish request for T_i to an executor to ensure that all version indexers learn about T_i.OUTCOME and take the appropriate steps. Even if M_i had managed to send out the finish request to an executor, it is safe for the version indexers to receive redundant messages about T_i.OUTCOME, since the version indexer finish process is idempotent, as described in §3.4.3. Finally, M_j can respond to the client with T_i.OUTCOME.
      This case corresponds to failure during state 4.
   c. If T_i.COMPLETE exists, M_j simply responds to the client with T_i.OUTCOME.
      This corresponds to failure after state 5.

The enumeration of the above cases proves that TASC will eventually and correctly can correctly recover from the failure of transaction managers through the TRP. Importantly, note that the TRP is idempotent and reentrant: it works properly regardless of when or how often it is invoked.

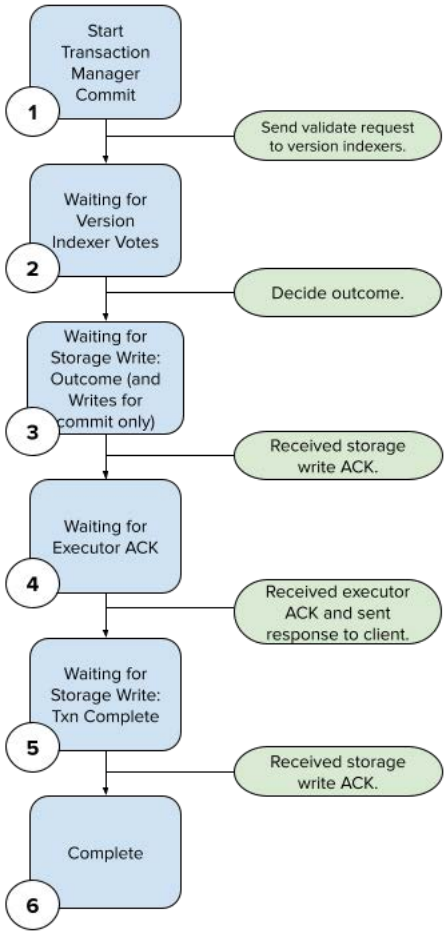


Figure 4.1: The finite state machine diagram for the transaction manager.

4.2.2 Version Indexer Failures

When Kubernetes detects that a version indexer has failed, it will simply restart the pod. Restarting the version indexer pod does not require any changes at the VIRL, because Kubernetes ensures that pods have the same cluster IP upon restart. The cluster IP addresses are the ones that are provided by the VIRL to the transaction managers on lookup requests. We consider two cases for version indexer failure:

1. If a version indexer fails before sending out its vote to the transaction manager, the transaction manager will observe a timeout and decide to abort the transaction. Even if the version indexer had managed to update the pending key version index(es) and persisted them to storage, the second phase of the commit protocol will ensure that the version indexer ultimately aborts the transaction, which means the relevant key versions will be deleted from the pending key version indexes.

2. If a version indexer fails after it has sent its vote to the transaction manager, TASC ensures that the version indexer will take the appropriate action during the second phase of the commit protocol, as the executor for the transaction will keep resending finish requests until the version indexer is restarted and performs the action.

By persisting the pending key version indexes to storage before sending the transaction manager its vote, the version indexer ensures that conflicting transactions are not allowed to commit and that ultimately the transaction's updates are visible if the transaction manager decides to commit. Similarly, the version indexer persists the updated committed key version index before responding with an ACK to the executor when processing a finish request for a committed transaction. Therefore, TASC is able to guarantee that if a client learns that transaction T_i has committed, the appropriate key version indexes will be updated and persistently stored even if relevant version indexers fail.

An important consideration to make is what happens to TASC's operations between the time that a version indexer fails and restarts. We briefly discuss each operation below.

Read. If the version indexer I_k for key k fails, reads for k will be unavailable until I_k restarts. However, an optimization TASC can perform is if the transaction manager M observes that I_k is unresponsive, M can send the read request for k to another version indexer I_j. It is safe for I_j to perform the read request for k, as the version indexer simply needs to fetch k.COMMITTED-KVI from storage (§3.4.1). Since I_j does not modify k.COMMITTED-KVI, it will not conflict with any operations I_k resumes when it restarts. The correctness of this process can also be exploited for performance gain in the absence of failures.

Write. Writes are unaffected, because they do not involve the version indexer and are buffered at the transaction manager as described in §3.4.2.

Commit. If version indexer I_k fails, we only need to consider the commit impact on a transaction T_i such that $k \in T_i$.WRITESET. Otherwise, I_k would not be involved in the commit process for T_i and the commit can proceed as normal. We now consider two cases for version indexer failure given the condition above applies.

1. If the transaction manager M is in state 1-2 from Figure 4.1 when I_k fails, transaction T_i will abort since M will observe a timeout while waiting for I_k's validation vote. Note this means that all new transactions that attempt to write to a key managed by I_k are doomed to abort while I_k is down. As a "pessimistic" optimization, the transaction managers can be warned of the state of I_k and abort such transactions preemptively upon any write attempt to reduce wasted resource utilization.

2. If the transaction manager M is in state 3-6 from Figure 4.1 when I_k fails, that means I_k has already sent its validation vote to M. Therefore, I_k's failure does not affect the commit process for T_i. Even if I_k remains in the failed state when the executor for T_i sends

a finish request, the outcome of T_i will be correct, as the executor will keep resending the finish request until it receives a response as described in §3.4.3.

4.2.3 Executor Failures

There are two cases for an executor to fail when it is performing a finish request of transaction T_i.

1. The executor fails before receiving all relevant version indexers' ACKs, which prevents it from deleting the *TID* from the ACTIVE-LIST (§3.4.3), and persisting the updated ACTIVE-LIST to storage.

   In this case, the restarted executor will notice that the *TID* for T_i is still in the ACTIVE-LIST due to the bootstrapping process described in §3.4.3. Thus, the executor will resend finish requests to all relevant version indexers, which will result in version indexers performing the correct action. The finish requests are idempotent at the version indexers, so it is safe for a version indexer to receive multiple finish requests for the same transaction.

2. The executor fails after receiving all relevant version indexers' ACKs, which prevents it from deleting the *TID* from the ACTIVE-LIST, and persisting the updated ACTIVE-LIST to storage.

   In this case, the version indexers are guaranteed to have taken the correct action since all ACKs were received. Therefore, there is no further action to be taken for T_i.

Chapter 5

Evaluation

In this chapter, we provide a detailed evaluation of TASC. In order to measure TASC's overhead of enforcing snapshot isolation, we compare its performance to AFT [19], which provides read atomic isolation, a weaker consistency model. Since both shims offer flexibility to run on various serverless storage backends, we configured them to run on ANNA KVS [21], a highly performant, scalable cloud key-value store that provides eventual consistency. To replicate the serverless application environment, the clients were AWS Lambda invocations in all experiments.

First, we measure TASC's performance overhead compared to AFT and ANNA KVS (§5.1). Then, we evaluate TASC's scalability (§5.2) by carefully studying the impact of scaling each core TASC component described in §3. Finally, we measure TASC's ability to recover quickly from failure (§5.3).

TASC is implemented in under 3,000 lines of codes and is deployed using Kubernetes [14]. We use a stateless load balancer to route *StartTransaction* (§3.1) requests to transaction managers in a round-robin manner. All experiments were run in the `us-east-1a` AWS availability zone (AZ). Each transaction manager, indexer, and executor ran on a `c5.2xlarge` EC2 instance with 8vCPUs (4 physical cores) and 16GB of RAM.

5.1 Overhead

We measure the performance overheads of TASC in comparison to AFT when running both shims on top of ANNA KVS. In all of our experiments, we measure the latency observed by serverless clients (i.e. AWS Lambda invocation), and do not include any overhead cost Functions-as-a-Service platforms may incur in creating the clients.

5.1.1 Transaction Latency

We first compare the cost of writing to TASC interposed on ANNA KVS to the the cost of writing to AFT interposed on ANNA KVS and to the cost of writing to ANNA KVS directly.

Figure 5.1 shows the median and 99th percentile latencies for write-only transactions with 1, 5, and 10 writes respectively. As expected, writes to ANNA KVS scale linearly. The overhead of AFT and TASC is largely attributed to the round trip time (RTT) of three API calls (*Start, Write, and Commit*) involved in the transaction. TASC is able to closely match AFT's performance for writes. The additional latency penalty of TASC can be attributed to the RTT between the transaction manager and the indexer, as well as the cost of persisting the pending version indexes to storage. It is also important to note that TASC writes data and pending version indexes to storage in parallel, which allows it to scale better than linearly as the number of writes in a transaction increases.

We next compare the cost of reading from TASC interposed on ANNA KVS to the the cost of reading from AFT interposed on ANNA KVS and to the cost of reading from ANNA KVS directly. Figure 5.2 shows the median and 99th percentile latencies for read-only transactions performing 1, 5, and 10 reads respectively. Again, as expected, reads from ANNA KVS scale linearly. Once again, both AFT and TASC have an overhead cost that is partially attributed to the RTT of the transaction's API calls. The latency penalty of TASC in comparison to AFT increases linearly as the number of reads increases, by a factor of about 3×, because TASC's read protocol (§3.4.1) involves fetching metadata, such as the committed version indexes of the keys being read and the transaction writeset of each candidate key version, from storage.

5.1.2 Data Skew

We now focus on evaluating the effect of the workload's access skew on TASC's performance. In this experiment, each transaction performed 1 read and 1 write. We measured 3 Zipfian distributions: 1.0 (lightly contended), 1.5 (moderately contended), and 2.0 (heavily contended). Figure 5.3 shows the median and 99th percentile latencies of TASC and AFT at various Zipfian distributions. TASC's latency decreases as the workload becomes more contended, because the committed version indexes are cached at the indexer after the first access.

As described in §2.3, snapshot isolation aborts transactions that write to keys previously written to by a concurrent transaction. Thus, we next study the impact of workload contention level on the abort rate. In this experiment, we deployed 5 parallel clients to perform 1000 transactions each consisting of 3 writes. Figure 5.4 shows the percent of transactions that abort for the various Zipfian distributions. As expected, the abort rate increases exponentially as the workload becomes highly contended. At a high level, this result is not specific to TASC: it's simply the pitfall that arises when using snapshot isolation to protect highly-contended transactions. For extremely high contention, other transactional techniques may be required that more explicitly sequentialize the transactions being issued [8].

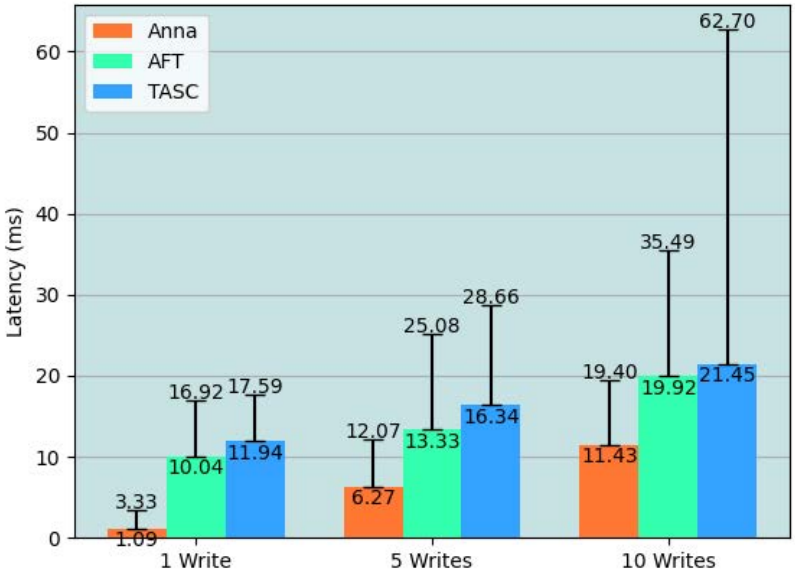


Figure 5.1: Median and 99th percentile latency for TASC over Anna with varying number of writes in a transaction. Compared with AFT over Anna and Anna directly. Numbers are reported from running 1,000 transactions.

5.2 Scalability

In this section, we focus on evaluating how well TASC scales, as that is a crucial aspect of any system meant to support serverless applications with dynamic workloads.

5.2.1 Determining Optimal Cluster Configuration

For systems like AFT, where there is only one component, evaluating scalability amounts to measuring the peak throughput as the number of servers are increased. However, in TASC, there are three core components: transaction managers, version indexers, and executors, that need to be configured. Therefore, it is necessary to first identify the target ratio of each component with respect to each other by measuring the component's impact on the overall throughput of the system. We make the assumption that each component has a linear relationship with the overall throughput of the system, so we use linear regression to model TASC's throughput as a function of the count of each core component. We derived a target ratio of 2 transaction managers, 3 version indexers, and 1 executor.

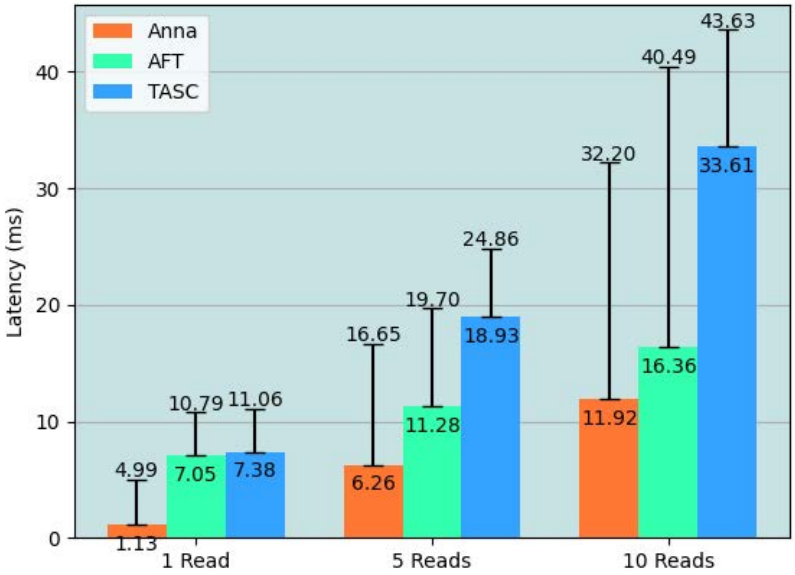


Figure 5.2: Median and 99th percentile latency for TASC over Anna with varying number of reads in a transaction. Compared with AFT over Anna and Anna directly. Numbers are reported from running 1,000 transactions.

5.2.2 Cluster Scalability

Using the linear relationship we derived, we define a TASC cluster size of 1 to consist of 2 transaction managers, 3 version indexers, and 1 executor. On the other hand, an AFT cluster size of 1 simply consists of 1 AFT node. Figure 5.5 shows how the peak throughput changes as we increase the cluster size for TASC and AFT. From the results we see that AFT is able to scale within 90% of its ideal slope, and TASC is able to scale within 80% of its ideal slope. Furthermore, TASC is able to seamlessly scale to over 15,000 transactions per second. We originally intended to demonstrate TASC's scalability beyond 15,000 transactions per second, but were restricted by AWS resource limits.

5.2.3 Cluster Change

Since TASC is meant to support serverless workloads, it is important to measure how quickly TASC is able to scale up its cluster and what the impact on throughput is during that period of time. As described in §3.6, scaling up transaction managers and executors does not require any coordination, so we focus on the interesting case of adding an indexer to the cluster. In this experiment, we begin with a moderately contended workload; then, we transition

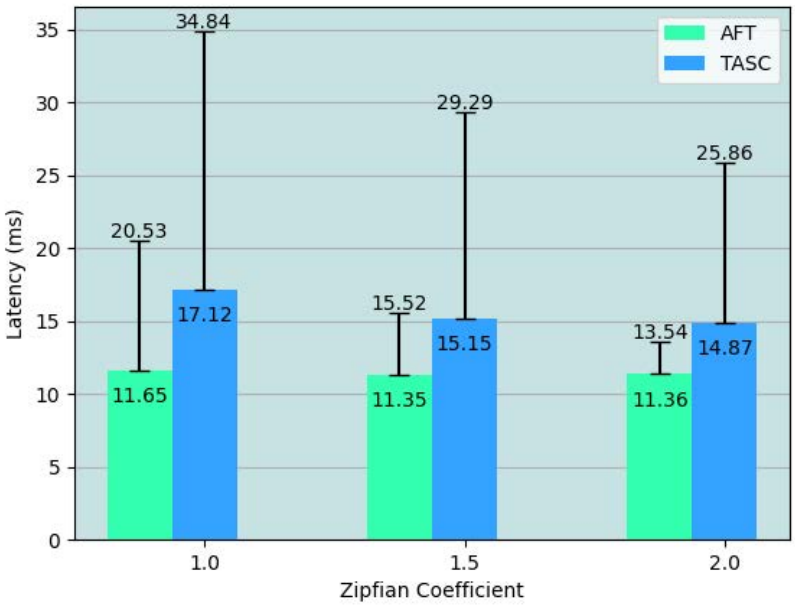


Figure 5.3: Median and 99th percentile latency for TASC and AFT over Anna. We vary the skew of the data access distribution to demonstrate the effects of contended workloads.

to a highly contended workload and add a new indexer to the cluster. Figure 5.6 shows our results. TASC's throughput falls for a short period of time, due to the scaling process described in §3.6.3, but it quickly returns the point before the the highly contended workload began.

5.3 Recovering from Failure

In this experiment, we measure TASC's ability to recover quickly from failure. We focus on failure of an indexer to demonstrate what happens to the throughput between the time of failure and when the indexer is able to recover. Figure 5.7 shows our results. TASC's throughput becomes 0 for a small period of time following the crash, but is able to pick back up as the indexer recovers quickly following the recovery process described in §4.2.2.

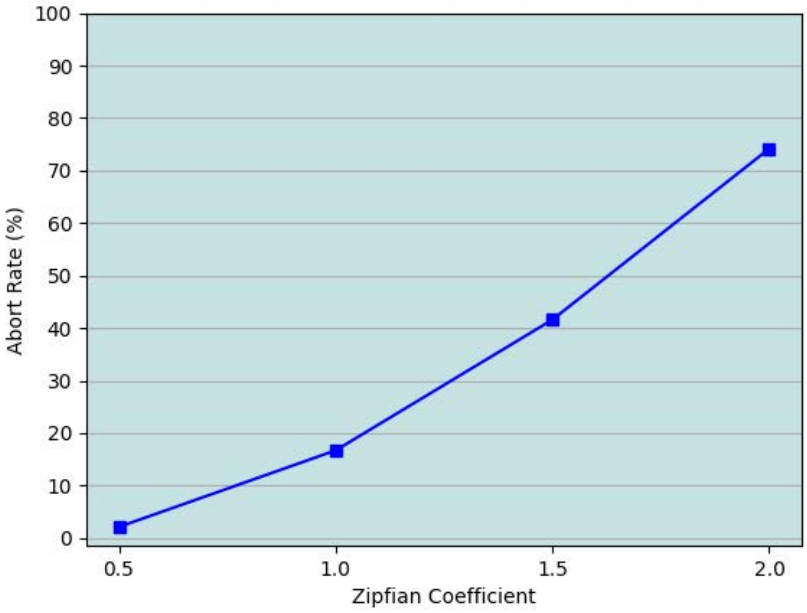


Figure 5.4: Abort rate for TASC as data access distribution skew is varied.

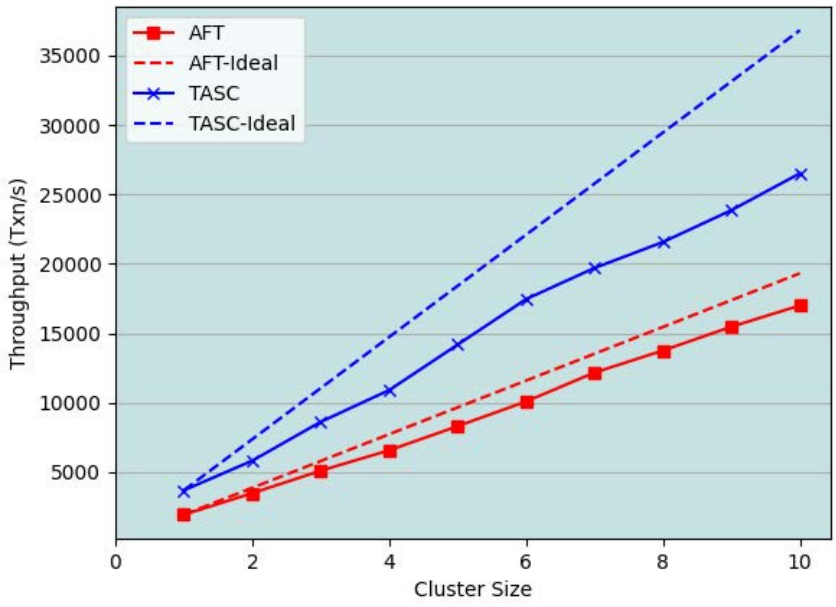


Figure 5.5: Throughput of TASC and AFT as a function of the cluster size. Cluster size of 1 for TASC includes 2 transaction managers, 3 version indexers, and 1 executor. Cluster size of 1 for AFT includes 1 AFT node.

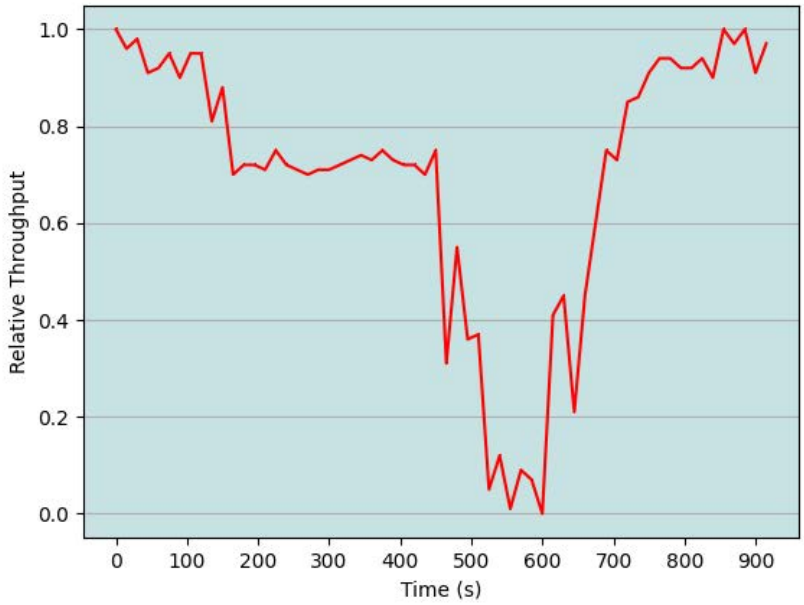


Figure 5.6: Time series of throughput of TASC as a version indexer is added in response to a high contention workload.

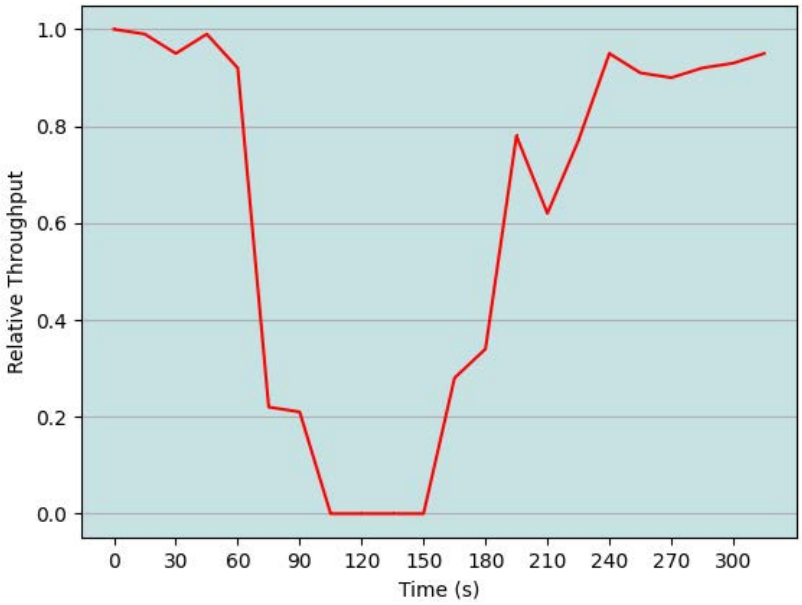


Figure 5.7: Time series of throughput of TASC during version indexer failure and recovery.

Chapter 6

Conclusion

In this thesis, we presented TASC, a low-overhead transactional shim with strong consistency for serverless computing. TASC interposes between commodity FaaS platforms and storage engines to provide snapshot isolation. TASC adds minimal overhead to existing serverless architectures, while providing atomicity, durability, and strong consistency. We introduced an architecture that decouples transaction metadata from data, allowing each TASC component to scale independently and ultimately support thousands of transactions per second.

6.1 Future Work

Although we have provided a strong consistency serverless shim, there are still future avenues of work to consider.

Garbage Collection. In our current implementation of TASC, we don't have a garbage collection for old transactions. With a proper garbage collector optimized for snapshot isolation, it would be possible to estimate the oldest running transaction in a system and reduce the state that needs to be stored on transaction managers and version indexers, as well the amount of data stored in the storage layer.

Considering Other Consistency Models. We demonstrate that snapshot isolation provides stronger isolation for serverless transactions than previous work, while having comparable overheads. However, snapshot isolation is not sufficient for a number of applications that require strict serializability [9], which suggests the need to investigate providing even stronger levels of consistency for serverless transactions.

Autoscaling Policy. Our current implementation does not have a robust autoscaling policy. A more comprehensive autoscaling engine could make decisions that optimize for maximum performance and minimum cost.

6.2 Related Works

Serverless Consistency. The rise of serverless computing has led to a need to provide stronger consistency for serverless computing. AFT [19] provides read atomic isolation, which is a weaker form of consistency than snapshot isolation, but is able to provide better performance due to a lack of coordination cost. Beldi [22] offers serializable transactions for serverless computing by using 2 phase locking and using shared storage as a log.

In-Memory Stores. There has been a recent trend in considering new architectures and principles in the design of in-memory storage systems. Meerkat [20] is a replicated, in-memory, transactional system that focuses on ensuring that non-conflicting transactions do not share resources, which is similar to TASC's principle of decoupling state across its various components.

Bibliography

[1] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. Tech. rep. USA, 1999.

[2] *Azure Functions*. https://azure.microsoft.com/en-us/services/functions/.

[3] Peter Bailis et al. "Scalable Atomic Visibility with RAMP Transactions". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: ACM, 2014, pp. 27–38. ISBN: 978-1-4503-2376-5. DOI: 10.1145/2588555.2588562. URL: http://doi.acm.org/10.1145/2588555.2588562.

[4] Hal Berenson et al. "A Critique of ANSI SQL Isolation Levels". In: *SIGMOD Rec.* 24.2 (May 1995), pp. 1–10. ISSN: 0163-5808. DOI: 10.1145/568271.223785. URL: https://doi.org/10.1145/568271.223785.

[5] E. Brewer. "CAP twelve years later: How the "rules" have changed". In: *Computer* 45.2 (Feb. 2012), pp. 23–29. ISSN: 0018-9162. DOI: 10.1109/MC.2012.37.

[6] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. "Paxos made live: an engineering perspective". In: *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. ACM. 2007, pp. 398–407.

[7] Erwin van Eyk et al. "The SPEC Cloud Group's Research Vision on FaaS and Serverless Architectures". In: *Proceedings of the 2nd International Workshop on Serverless Computing*. WoSC '17. Las Vegas, Nevada: Association for Computing Machinery, 2017, pp. 1–4. ISBN: 9781450354349. DOI: 10.1145/3154847.3154848. URL: https://doi.org/10.1145/3154847.3154848.

[8] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. "High Performance Transactions via Early Write Visibility". In: *Proc. VLDB Endow.* 10.5 (Jan. 2017), pp. 613–624. ISSN: 2150-8097. DOI: 10.14778/3055540.3055553. URL: https://doi.org/10.14778/3055540.3055553.

[9] Alan Fekete et al. "Making Snapshot Isolation Serializable". In: *ACM Trans. Database Syst.* 30.2 (June 2005), pp. 492–528. ISSN: 0362-5915. DOI: 10.1145/1071610.1071615. URL: https://doi.org/10.1145/1071610.1071615.

[10] J. N. Gray. "Notes on data base operating systems". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1978, pp. 393–481. DOI: 10.1007/3-540-08755-9_9. URL: https://doi.org/10.1007/3-540-08755-9_9.

[11] *gRPC: A high performance, open source universal RPC framework*. https://grpc.io/.

[12] Theo Haerder and Andreas Reuter. "Principles of Transaction-Oriented Database Recovery". In: *ACM Comput. Surv.* 15.4 (Dec. 1983), pp. 287–317. ISSN: 0360-0300. DOI: 10.1145/289.291. URL: https://doi.org/10.1145/289.291.

[13] Eric Jonas et al. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Tech. rep. UCB/EECS-2019-3. EECS Department, University of California, Berkeley, Feb. 2019. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html.

[14] *Kubernetes: Production-Grade Container Orchestration*. http://kubernetes.io.

[15] *AWS Lambda*. https://aws.amazon.com/lambda/.

[16] *Make a Lambda Function Idempotent*. https://aws.amazon.com/premiumsupport/knowledge-center/lambda-function-idempotent/.

[17] Yi Lin et al. "Snapshot Isolation and Integrity Constraints in Replicated Databases". In: *ACM Trans. Database Syst.* 34.2 (July 2009). ISSN: 0362-5915. DOI: 10.1145/1538909.1538913. URL: https://doi.org/10.1145/1538909.1538913.

[18] Masoud Saeida Ardekani et al. "On the Scalability of Snapshot Isolation". In: *Euro-Par 2013 Parallel Processing*. Ed. by Felix Wolf, Bernd Mohr, and Dieter an Mey. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 369–381. ISBN: 978-3-642-40047-6.

[19] Vikram Sreekanti et al. "A Fault-Tolerance Shim for Serverless Computing". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387535. URL: https://doi.org/10.1145/3342195.3387535.

[20] Adriana Szekeres et al. "Meerkat: Multicore-Scalable Replicated Transactions Following the Zero-Coordination Principle". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387529. URL: https://doi.org/10.1145/3342195.3387529.

[21] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. "Autoscaling Tiered Cloud Storage in Anna". In: *Proc. VLDB Endow.* 12.6 (Feb. 2019), pp. 624–638. ISSN: 2150-8097. DOI: 10.14778/3311880.3311881. URL: https://doi.org/10.14778/3311880.3311881.

[22] Haoran Zhang et al. "Fault-tolerant and transactional stateful serverless workflows". In: *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 2020, pp. 1187–1204.