# SconeKV: a Scalable, Strongly Consistent Key-Value Store

João Gonçalves, Miguel Matos, and Rodrigo Rodrigues

**Abstract**—For decades, relational databases provided a strong foundation for constructing applications due to their ACID properties. However, distributed applications reached a scale, both in terms of data volume and number of concurrent clients, that traditional databases cannot accommodate. NoSQL databases addressed this problem by trading consistency for scalability, namely through horizontal scalability schemes supported by optimistic replication protocols, which only guarantee eventual consistency.

In this paper, we explore a novel design between the two extremes, which is able to scale to large deployments while still offering strong consistency guarantees in the form of serializable transactions. Our key insight is to leverage recent advances in membership services that provide strongly consistent views at scale. Those assurances from the membership layer simplify building efficient and consistent storage protocols. Our evaluation of the resulting system, SconeKV, in a realistic scenario shows that it scales and performs better than CockroachDB while being competitive with Cassandra.

**Index Terms**—Distributed systems, storage, consistency, scalability

## 1 Introduction

Distributed systems began at a much smaller scale than today. Initially, these systems were comprised of a few client nodes connected to a centralized database for storage. Relational databases provided a dependable foundation, offering transactional support and strong consistency for client operations. Today, this paradigm has changed and users demand highly scalable systems that are always available. Traditional relational databases were able to scale vertically, but now systems require databases that scale horizontally, with low latency, worldwide.

Early approaches to distributed scalable storage leveraged peer-to-peer distributed hash tables (DHTs) [1], [2], [3], [4], [5], which provide the location of objects at a large scale, but only replicate immutable data or provide very weak consistency guarantees. Modern key-value stores leverage the same principles as DHTs but offer more robust guarantees. For instance, Apache Cassandra [6] combines high availability and reliability with low latency, and allows for custom quorum sizes to control the consistency probability. However, its consistency guarantees are much weaker than those in relational databases. Moreover, and despite scaling to a large number of nodes, the semantics of Cassandra are brittle in the presence of churn and faults [7], which are the norm rather than the exception as the system grows. At the other end of the consistency spectrum, CockroachDB [8], [9] is a distributed database with ACID properties, built on top of a transactional and strongly consistent key-value store. However, CockroachDB cannot scale above a few tens of nodes (around 20), as we show in §6.2, and the same applies to systems that follow similar approaches. This limitation stems from the use of consensus, which is an expensive primitive, known to scale poorly with the system size.

This state of affairs leaves modern application developers with a conundrum: either one sacrifices consistency for scalability, resulting in applications that are harder to program and maintain, or one chooses strong consistency but is limited to a small-scale system.

In this paper, we aim to show that programmers do not necessarily have to choose between consistency and scalability. To this end, we present SconeKV, a scalable transactional key-value store with strong consistency guarantees that provides serializable transactions. Our key insight to address this tension is that it is possible to simplify the protocols required for strong consistency, and minimize the amount of synchronization they impose, by layering them on top of a membership layer that offers strong semantics at scale, effectively pushing the burden of simultaneously achieving consistency and scalability to the lower layers of the system. In particular, we build on recent research that showed, for the first time, how to build a scalable consistent membership service (e.g., Rapid [10] or PRIME [11]) that guarantees that all correct nodes share a common view of the system. Then, with these strong guarantees in place, other fundamental aspects of a distributed key-value store such as data partitioning, replication, and transactional processing can be substantially simplified, resulting in a leaner and scalable design.

Nonetheless, several challenges need to be addressed by the design of SconeKV, such as designing mechanisms for horizontal partitioning, consistent replication and a coordination protocol, while also handling membership changes. We show, through our design, how starting from strong guarantees at the foundational levels allows the upper levels to be scalable, provide strong semantics, while also facilitating the reasoning about the system correctness.

In our experimental evaluation, we compare SconeKV with Cassandra and CockroachDB, two state-of-the-art production systems, using the YCSB [12] and TPC-C [13]

- *J. Gonçalves, M. Matos and R. Rodrigues are with the Instituto Superior Técnico (ULisboa) and INESC-ID*
  *E-mails: {joao.tiago.goncalves, miguel.marques.matos, rodrigo.miragaia.rodrigues}@tecnico.ulisboa.pt*

*Manuscript received DATE; revised DATE.*

benchmarks. The experimental results show that SconeKV outperforms CockroachDB in terms of throughput in write intensive workloads by up to 15× while being competitive with Cassandra in all workloads. We also show that SconeKV scales well in both write and read intensive workloads, whereas CockroachDB does not scale even with a small fraction of writes. Our contributions include:

- a layering of storage protocols on top of a novel class of membership service protocols, showing that these new models for group membership can significantly improve the characteristics of modern key-value stores;
- the design of SconeKV, a scalable transactional key-value store that provides serializable transactions;
- a prototype implementation of SconeKV, and a comparative evaluation with state-of-the-art production systems.

The rest of the paper is organized as follows. §2 presents some background on membership protocols. §3 discusses related work. §4 describes the design of SconeKV. §5 details the implementation of the prototype and explains some optimizations. §6 presents the experimental evaluation. §7 concludes the paper and discusses future work directions.

## 2 Background

In this section, we provide some background on membership protocols given their importance in the design of SconeKV. Group membership protocols fall into one of two categories. Logically centralized services [14], [15] present a simple solution, using a small group of processes to maintain a system view, while the majority of members query it periodically. Besides the simple design, this approach also offers strong consistency semantics, assuming that there is an agreement between this small group of nodes. However, relying on that small group limits the scalability and availability of the membership service.

Alternatively, fully decentralized solutions [16], [17] have been proposed as a means to tackle the aforementioned downsides of centralized services. These approaches use gossip-based techniques to disseminate membership updates, thus achieving a much higher scale while also being more resilient. However, this comes at the cost of sacrificing the consistency of views across large-scale clusters.

Recently, a novel class of membership protocols has emerged. PRIME [11] and Rapid [10] are fully decentralized but manage to offer strongly consistent views at scale. Essentially, both protocols detect failures by having each node monitor $K$ other nodes, and require multiple reports to remove a node from the group. They differ in the way the updates are processed and disseminated. Rapid [10] employs multi-process cut detection to combine multiple node failures in a single membership update, using leaderless Fast-Paxos [18] to reach a decision in the normal case, or classic Paxos [19], [20] if it is unable to reach a fast agreement. PRIME [11] processes failures individually but uses a probabilistic total order dissemination algorithm [21] to convey membership updates. This algorithm ensures that nodes eventually agree on the set of updates received with high probability and process these updates in a total order, thus guaranteeing that views progress consistently. In sum, both approaches offer a scalable membership abstraction that delivers view updates in a consistent way.

## 3 Related Work

Early approaches to distributed storage leveraged the routing mechanisms of distributed hash tables [1], [2], [3] to scale. For example, OceanStore [5] is a globally persistent storage service that provides serialized updates on replicated objects on an untrusted infrastructure. It uses Tapestry [4] to construct a routing overlay and allows for concurrent updates without wide-area locking by employing predicate-based update conflict resolution. OceanStore resolves conflicts by determining an order for the updates, evaluating the predicates and applying them atomically.

Dynamo [22] and Cassandra [6] are systems that opted to weaken their consistency guarantees in order to scale and be highly available, employing optimistic replication protocols and delegating conflict resolution to the client. Cassandra relies on a membership solution with weak consistency guarantees, further affecting the consistency the system is able to provide during view changes. As different nodes can have conflicting views, nodes can be assigned overlapping token ranges upon joining [7], resulting in inconsistent client operations. This also impacts system bootstrap - deploying large clusters takes a long time as nodes need to be slowly added one at a time [7] to allow the token range selections to propagate throughout the system.

Causally consistent systems, such as Eiger [23], COPS [24] or ChainReaction [25], strike a balance between eventual and strong consistency by guaranteeing the order between causally dependent updates. Moreover, these systems remain available and maintain their consistency guarantees even in the event of a network partition if the client remains connected to the same server nodes. However, they provide weak guarantees regarding write conflict resolution, specially when involving multiple objects, which might result in a divergence of replica's state. These solutions can scale but are limited to applications for which this level of guarantees is enough.

On the strongest end of the consistency spectrum we have systems such as Google Spanner [26], which is a highly scalable SQL database with ACID guarantees. It shards data across Paxos [19], [20] state machines, and relies on GPS and atomic clocks to order transactions. It provides strong consistency at scale but requires specialized hardware and an infrastructure not generally accessible to smaller players. CockroachDB [8], [9] is an industry solution that draws inspiration from Spanner's design. It is a distributed SQL database with ACID properties, built on top of a strongly consistent key-value store. It uses Raft [27] for state machine replication and replaces Spanner's atomic clocks and GPS with a software solution relying on Hybrid Logical Clocks (HLC) [28]. HLC combine physical time with logical clocks, offering wait-free transaction ordering and consistent snapshots for a specified timestamp. HLC allows for some clock skew, but CockroachDB still requires replica clocks to be within a configurable offset (500ms by default) to work correctly, shutting down nodes that get out of sync with 80% of the cluster. Besides this reliance on clock synchronization, which is harder to achieve in a geo-replicated deployment, CockroachDB is also limited in its scalability, due to the inherent cost of consensus, as we will show in our evaluation. Physalia [29] and EdgeKV [30] are examples of a strongly consistent key-value stores that achieve scalability through sharding, however neither supports cross-shard transactions. SCORe [31] leverages vector clocks to guarantee serializability and abort-free read-only transactions. This main contribution, i.e. the use of vector clocks, is orthogonal to SconeKV's layered design, in the sense that, it could be integrated with our proposed transaction management and replication layers and guarantee the same semantics — abort-free read-only transactions. This problem was also targeted by recent blockchain-based solutions but with a different fault-model (i.e. byzantine) and different application scenarios and performance characteristics [32], [33], [34].

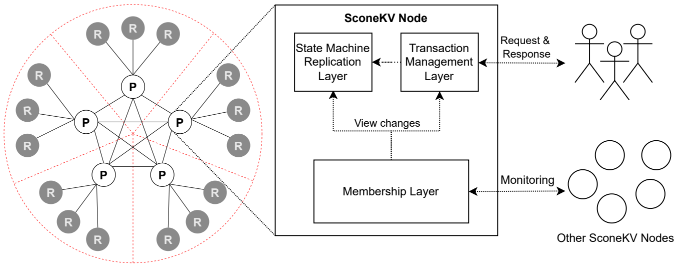


Fig. 1: SconeKV cluster topology and layered architecture of each node. $P$ represents a primary node, $R$ represents a replica. The red dotted lines divide the cluster into buckets.

## 4 SconeKV

In this section, we present the design of SconeKV, a distributed key-value store that provides strictly serializable transactions (without opacity). We assume a shared-nothing architecture and a crash-failure model. We also assume the partial synchrony model [35]. In this model, there may be an unstable period, where messages exchanged between correct processes are arbitrarily delayed. However, there is a known bound $\Delta$ on the worst-case network latency and an unknown Global Stabilization Time (GST), such that after GST, all messages between correct processes arrive within $\Delta$. Note that safety is always preserved even in the presence of asynchrony and the partial synchrony assumptions are only necessary to ensure liveness.

### 4.1 Overview

We start by presenting an overview of SconeKV and discussing its layered design. The key insight is that using a scalable and consistent membership base layer (§2) simplifies the design of the layers above, particularly when trying to enforce strong semantics. This lower layer interacts with the layers above by asynchronously delivering a new view after each update, consisting of a view identifier and a list of nodes. Finally, we note that the membership layer is not in the critical path of the storage protocols, as described next.

The next problem that needs to be addressed is ensuring consistent replication. For the sake of better scalability and the flexible reconfiguration, based on balancing data or processing load, we opted to employ horizontal partitioning. SconeKV's namespace is an identifier ring divided into sections we call buckets. Nodes and data items are assigned to buckets, rather than points in the space, using consistent hashing [36]. Each data item is assigned to a single bucket and is managed and replicated by the set of nodes that belong to that bucket. To ensure consistency within a bucket, we employ Viewstamped Replication (VSR) [37], [38], turning each bucket in an independent state machine following a primary-backup scheme.

In VSR, each update to the state of a bucket represents an entry in a log, and log entries flow from the primary to the replicas. Briefly, the algorithm works as follows. To replicate a log entry, the primary issues a Prepare. Once a replica receives the message, it processes the entry iff it has processed all previous entries and replies to the primary with PrepareOK. When the primary receives $f$ PrepareOK's, the entry is consistently propagated and can be safely committed to the log. This ensures safety inside each bucket if no more than $f$ nodes are faulty at any given moment, provided that each bucket has at least $2f+1$ nodes. For full details on VSR we refer the reader to [37], [38]

Through the use of a consistent membership layer, we guarantee that all nodes in a bucket agree on its configuration for a given view. Thus, to determine the primary, each member runs a deterministic function (for instance, selecting the node with the lowest identifier). This, combined with the view change algorithm described in the revisited paper [38], allows each bucket to be abstracted as a single entity that guarantees linearizable updates and is tolerant to faults.

Finally, SconeKV needs to coordinate different buckets in order to provide distributed transactions. For that, we employ a two-phase commit protocol (2PC) with locking semantics. Each transaction is decided on by the primaries of the buckets involved and each phase of the protocol is consistently replicated inside the respective bucket, as an entry in the log, before interacting with the other buckets. This, combined with a retry mechanism, ensures that we tackle 2PC's known weak liveness properties [39]. Furthermore, the locking semantics ensure that we guarantee serializable transactions. Since the consistent membership layer ensures that all nodes agree on the configuration of the system as a whole for a given view, and thus agree on the primaries of each bucket for that same view, it does not require a leader election to determine the coordinator for a given transaction.

The cluster topology, as well as the layered architecture of each SconeKV node, is depicted in Figure 1. To summarize, the design of SconeKV leverages the strong properties of the foundational membership layer to reduce the complexity and synchronization costs of the replication and transaction management protocols employed above. In contrast to traditional databases, that aggregate all responsibilities (including membership management, replication, and transaction management) in a single protocol, leading to complex solutions that are difficult to reason about, desegregating these responsibilities in different layers allows for the delegation of certain guarantees, which in turn simplifies the design and reasoning about the behaviour of the final protocol. Notably, SconeKV accomplishes this without compromising on the properties offered by the system as a whole, namely scalability, safety, and liveness.

### 4.2 Client Interactions

SCONEKV exposes the following operations to its clients: `read(key)` returns the current value for that key, `write(key, value)` inserts or updates the key with the given value, `delete(key)` removes the key-value pair from the store, `commit()` attempts to apply all the pending modifications (write and delete operations) to the system, and `abort()` discards all pending operations. Values are arrays of bytes, opaque to the system, and a write to a given key overwrites the previous value. Each key-value pair is associated with a version, its own lock and lock queue. Versions are returned by all operations. They are initially set to zero and incremented by one with each write operation.

Each operation performed by a client corresponds to a request to a SCONEKV node belonging to the bucket that holds the respective key. All operations are performed in the context of a transaction. The client library locally maintains the read-write set for each transaction, containing the versions for each accessed key. Each transaction has an identifier (*txID*), generated by the client library. The *txID* is the concatenation of an ascending local counter with the client identifier, thus ensuring uniqueness. To externalize the pending operations, which are stored in the local read-write set, the client must issue a `commit`. This operation will be successful or unsuccessful, depending on whether the versions observed by the transaction (those in the read-write set) match the current most up-to-date versions for the same keys at the time the commit is issued.

SCONEKV offers strictly serializable transactions but without opacity. This means that, at any point in time, the state of the system as a whole is equivalent to some serial order of the transactions committed up to that point. Additionally, this ordering is consistent with real-time, meaning that if $T_A$ is committed before $T_B$ begins, then $T_A$ precedes $T_B$ in the serial order. Not providing opacity means that aborted transactions do not necessarily observe a consistent snapshot of the database.

Finally, it is worth emphasizing that clients are not part of the system membership. Upon starting, a client contacts any node in the system to obtain the current view. This view is used throughout the client's lifetime and is only updated in case of a timeout contacting a node (resulting in the client requesting a new view to another node) or if the client sends a request to an incorrect node (wrong bucket and/or incorrect primary), in which case the contacted node replies with an updated view of the system.

### 4.3 Transaction Processing

We now present how SCONEKV processes transactions and guarantees strict serializability. For simplicity, we present the algorithm in a fault-free scenario and further discuss how SCONEKV handles failures in §4.5.

As described before, clients operations do not modify the system state until the client attempts to commit the transaction. Commit requests are routed to the primary which then replicates them inside the bucket. Depending on the keys accessed, a transaction can span multiple buckets. In that case, the primaries of the buckets involved need to coordinate to determine whether the transaction can commit or not, using a two-phase commit protocol with locking semantics. The coordinator of a transaction spanning multiple buckets is selected deterministically as the primary of the bucket with the lowest identifier.

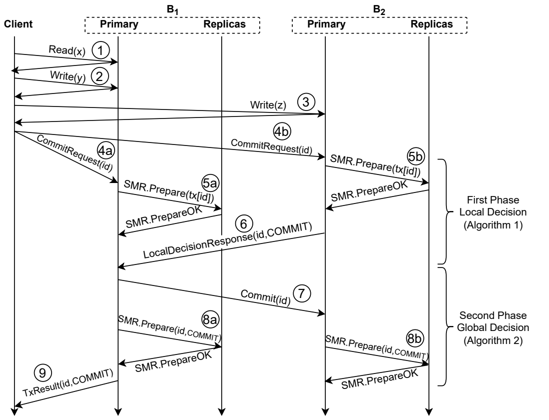


Fig. 2: Messages exchanged in a transaction involving 2 buckets. Primary 1 is the transaction coordinator, both primaries accept the transaction locally and commit it. For simplicity, the replicas for each bucket are represented as a single entity each. The replication and the client interactions (with the exception of the commit request) were omitted from the algorithms due to space constraints.

Following 2PC's semantics, transactions are processed in two distinct phases entailing a local and a global decision. Figure 2 shows an example of the messages exchanged during a transaction that spans multiple buckets. Suppose that a transaction accesses keys $x$, $y$ and $z$ assigned to two different buckets. The client initiates a transaction and performs a series of operations by contacting nodes in $B_1$ (steps ① and ②) and $B_2$ (③). The client attempts to commit the transaction by sending to the primaries of each bucket the read-write subset of keys assigned to their bucket and a list of all buckets involved (④a and ④b). Upon receiving the commit request, each primary locally decides whether the transaction can commit (following Algorithm 1, which we detail later), and replicates the request and local decision to the bucket's replicas (⑤a and ⑤b). Once replicated, the primary of $B_2$ communicates the local decision to the transaction coordinator (⑥), ending the first phase of the protocol. Once the transaction coordinator receives the local decisions of all buckets involved, it starts the second phase of the protocol by deciding the outcome of the transaction (commit or abort) and communicates this decision to all the other primaries (⑦) which in turn replicate the global decision within their respective buckets (⑧a and ⑧b). Finally, the coordinator replies to the client (⑨).

According to 2PC, a transaction is committed iff all participants accept the transaction locally. It is well-known that two-phase commit has weak liveness properties [39], and requires a recovery mechanism upon failures. In SCONEKV, due to our layered design, we are able to reduce this complexity. In particular, upon a failure, all nodes will receive a new view without the failed node and restart the protocol if needed (for instance, due to a failure of the coordinator). This is also the reason why the coordinator does not need to wait for an acknowledgment from all the other primaries

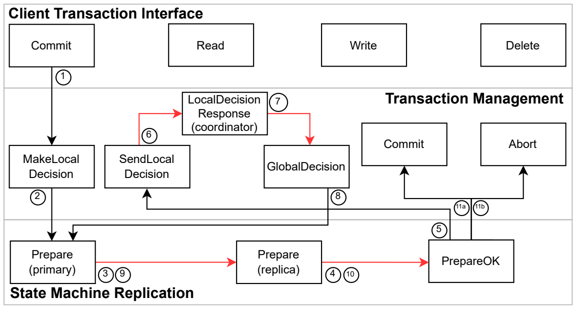


Fig. 3: Sequence of events to commit a transaction. Red arrows indicate the event was triggered in a remote node.

before replying to the client (end of the second phase in Figure 2). Since the primaries replicate the request and local decision before sending the local decision to the coordinator ((5a), (5b)), upon the failure of a primary, the replicas will run the view change protocol described in [38] and guarantee that the new primary has the required log entries to apply the pending transaction. We detail this in §4.5.

We now detail the life cycle of a transaction from when it is submitted by the client until it is committed or aborted. Figure 3 outlines the general flow, and Algorithm 1 and Algorithm 2 show the pseudocode for the first and second phases, respectively. In detail:

**1)** The client issues `commit` to each primary involved in the transaction. Each request includes only the subset of keys that are assigned to that specific bucket. Once a primary receives the request, it sets the transaction state as *received* and triggers MAKELOCALDECISION (Figure 3, step ①, and Algorithm 1 lines 13–35). Each local decision is processed independently by the primary of each bucket involved by verifying the following:

**a)** Guarantee that each operation inside the transaction was performed on the most recent version of that key (Algorithm 1 line 15 and lines 46–54), otherwise the transaction is locally rejected, eventually leading to an abort because it does not respect serializability (Algorithm 1, lines 31–32).

**b)** Check if all locks of the keys accessed by the transaction inside that bucket are available. If that is the case, they are acquired (Algorithm 1 lines 16–18). If any lock acquisition fails, all lock acquisitions are queued (Algorithm 1 line 22) to reduce contention and, possibly, allow other transactions to commit.

Note that, to simplify the presentation, the pseudocode assumes that events are not processed concurrently. In practice, to ensure correctness, the lock must be acquired before checking the version of a key.

**2)** If the transaction failed to acquire all locks, it will wait in a queue for the conflicting transaction(s) to conclude and trigger MAKELOCALDECISION to restart the process. Otherwise, the transaction state is set to *prepare-commit* and the local decision is replicated inside the bucket (②, ③, ④, Algorithm 1 lines 19–20).

**3)** Once replicated, the local decision is sent to the transaction coordinator (⑤ and ⑥, Algorithm 1 lines 37–44).

**4)** The coordinator reaches a global decision and informs all participants (⑦, Algorithm 2 lines 1–16).

**5)** Upon receiving the global decision, each primary changes the transaction state to *to-commit* or *to-abort*, as appropriate,

**Algorithm 1** Local Decision - First Phase

```
 1: upon event ⟨INIT⟩ do
 2:     $txs \leftarrow \varnothing$
        ▷ map with all transactions, indexed by the transaction ID, containing rwSet (only with keys assigned to that bucket), buckets involved, current transaction state and local decision responses
 3: end event
 4:
    ▷ Triggered by the client issuing the commit request
 5: upon event ⟨primary, COMMITREQUEST| txID, rwSet, buckets⟩ do
 6:     $txs[txID].rwSet \leftarrow rwSet$
 7:     $txs[txID].buckets \leftarrow buckets$
 8:     $txs[txID].state \leftarrow$ received
 9:     $txs[txID].responses \leftarrow 0$
10:     trigger ⟨MAKELOCALDECISION|txID⟩
11: end event
12:
13: upon event ⟨primary, MAKELOCALDECISION| txID⟩ do
14:     if $txs[txID].state =$ received then
15:         if CHECKVALIDTRANSACTION($txID$) then
    ▷ validate the versions used in the tx
16:             $owners \leftarrow$ GETLOCKOWNERS($txID$)
17:             if $owners = \varnothing$ then
18:                 ACQUIRELOCKS($txID$)
19:                 $txs[txID].state \leftarrow$ prepare-commit
20:                 trigger ⟨SMR.PREPARE|txs[txID]⟩
    ▷ PREPARE is triggered in the SMR layer, and once the decision is replicated it triggers SENDLOCALDECISION
21:             else
22:                 QUEUELOCKS($txID$)
23:                 if $txID <$ MIN($owners$) then
    ▷ if txID as a lower identifier than all current lock owners, it should be executed first to avoid a distributed deadlock
24:                     for each $otherTxID \in owners$ do
25:                         $otherCoord \leftarrow$ GETCOORD($txs[t].buckets$)
26:                         send ⟨REQUESTREVERTLOCALDECISION|otherTxID⟩ to otherCoord
27:                     end for
28:                 end if
29:             end if
30:         else
31:             $txs[txID].state \leftarrow$ prepare-abort
32:             trigger ⟨SMR.PREPARE|txs[txID]⟩
33:         end if
34:     end if
35: end event
36:
37: upon event ⟨primary, SENDLOCALDECISION| txID⟩ do
38:     $txCoord \leftarrow$ GETCOORD($txs[txID].buckets$)
39:     if $txs[txID].state =$ prepare-commit then
40:         send ⟨LOCALDECISIONRESPONSE|txID, commit⟩ to txCoord
41:     else
42:         send ⟨LOCALDECISIONRESPONSE|txID, abort⟩ to txCoord
43:     end if
44: end event
45:
46: function CHECKVALIDTRANSACTION(txID)
47:     for each $(key, \_, version, \_) \in txs[txID].rwSet$ do
48:         $currentVersion \leftarrow$ GETVERSION($key$)
49:         if $currentVersion \neq version$ then
50:             return $False$
51:         end if
52:     end for
53:     return $True$
54: end function
55:
56: function GETLOCKOWNERS(txID)
57:     $owners \leftarrow \varnothing$
58:     for each $(key, \_, \_, \_) \in txs[txID].rwSet$ do
59:         $lockOwner \leftarrow$ GETLOCKER($key$)
60:         if $lockOwner \neq$ NULL $\wedge$ $lockOwner \notin owners$ then
61:             $owners \leftarrow owners \cup lockOwner$
62:         end if
63:     end for
64:     return $owners$
65: end function
```

**Algorithm 2** Global Decision - Second Phase

```
1:  upon event ⟨coordinator, LocalDecisionResponse| txID, response⟩ do
2:      if $response = abort \wedge txs[txID].state \notin \{aborted, to\text{-}abort\}$ then
        ▷ transaction was rejected locally but not yet globally
3:          for each $bucket \in txs[txID].buckets$ do
4:              $primary \leftarrow$ GetPrimary($bucket$)
5:              send ⟨GlobalDecision|txID, abort⟩ to primary
6:          end for
7:      else
        ▷ transaction was accepted locally
8:          $txs[txID].responses \leftarrow txs[txID].responses + 1$
9:          if $txs[txID].responses = \#txs[txID].buckets$ then
10:             for each $bucket \in txs[txID].buckets$ do
11:                 $primary \leftarrow$ GetPrimary($bucket$)
12:                 send ⟨GlobalDecision|txID, commit⟩ to primary
13:             end for
14:         end if
15:     end if
16: end event
17:
18: upon event ⟨primary, GlobalDecision| txID, decision⟩ do
19:     if $decision = commit$ then
20:         $txs[txID].state \leftarrow to\text{-}commit$
21:     else if $decision = abort$ then
22:         $txs[txID].state \leftarrow to\text{-}abort$
23:     end if
24:     trigger ⟨SMR.Prepare|txs[txID]⟩
25: end event
26:
27: upon event ⟨primary, Commit| txID⟩ do
28:     for each $(key, value, version, type) \in txs[txID].rwSet$ do
29:         if $type =$ WRITE then
30:             $newVersion \leftarrow version + 1$
31:             Put($key, value, newVersion$)
32:         else if $type =$ DELETE then
33:             Delete($key$)
34:         end if
35:     end for
36:     $txs[txID].state \leftarrow committed$
37:     if IsCoordinatorTx($txID$) then
38:         send ⟨TxResult|txID, commit⟩ to txID.client
39:     end if
40:     ReleaseLocks($txID$)
41: end event
42:
43: upon event ⟨primary, Abort| txID⟩ do
44:     $txs[txID].state \leftarrow aborted$
45:     if IsCoordinatorTx($txID$) then
46:         send ⟨TxResult|txID, abort⟩ to txID.client
47:     end if
48:     ReleaseLocks($txID$)
49: end event
50:
51: function ReleaseLocks(txID)
52:     $restartTx \leftarrow \varnothing$
53:     for each $(key, \_, \_, \_) \in txs[txID].rwSet$ do
54:         if UnlockKey($key, txID$) then
55:             $nextInQueue \leftarrow$ GetNextInLockQueue($key$)
56:             if $nextInQueue \neq \varnothing \wedge nextInQueue \notin restartTx$ then
57:                 $restartTx \leftarrow restartTx \cup nextInQueue$
58:             end if
59:         end if
60:     end for
61:     for each $tx \in restartTx$ do
62:         trigger ⟨MakeLocalDecision|tx⟩
63:     end for
64: end function
65:
66: function UnlockKey(key, txID)
    ▷ If txID holds the lock associated with key, the lock is released and
    returns true, otherwise returns false.
67: end function
68:
69: function GetNextInLockQueue(key)
    ▷ Returns the lowest txID in the queue for the lock associated with
    key, if one exists.
70: end function
```

and replicates it (⑧, ⑨, ⑩, Algorithm 2 lines 18–25).

**6)** Once the global decision is consistently replicated inside each bucket involved, and according to it, each node commits or aborts the transaction and sets its final state as *committed* or *aborted* ((11a) or (11b), Algorithm 2 lines 27–41 or 43–49, respectively).

**7)** The transaction coordinator responds to the client.

**8)** Finally, each primary releases the locks of the transaction, and triggers MakeLocalDecision to process the remaining transactions that have queued locks, if any (Algorithm 2, lines 51–64).

## 4.4 Avoiding Distributed Deadlocks

This design, like any lock-based protocol, may lead to a distributed deadlock, e.g., when transactions, $T_A$ and $T_B$ concurrently acquire locks on keys that belong to two different buckets, $B_1$ and $B_2$, and the primary of $B_1$ ($P_1$) receives $T_A$ first, whereas the primary of $B_2$ ($P_2$) receives $T_B$ first. We address this problem by giving priority to the transaction with the lowest identifier. Figure 4 illustrates the example above with two conflicting transactions $T_A$ and $T_B$. Primary $P_2$ that locally accepted transaction $T_B$ (Figure 4, ①) upon receiving a conflicting transaction $T_A$, informs the transaction coordinator ($P_1$) that $P_2$'s decision for $T_B$ should be reverted as $T_A$ has a lower identifier and hence higher priority (Figure 4, ②). This request is granted (Figure 4, ③) iff the transaction coordinator ($P_1$) has not yet issued a global decision for $T_B$.

This logic is triggered during MakeLocalDecision (Algorithm 1). If a transaction fails to acquire the locks, the primary queues the locks and determines if that transaction should be processed before all other transactions that currently own any of those locks (Algorithm 1, lines 14–21). If that is the case, that primary will request to revert its local decision for those transactions. For each transaction with a lower priority, the respective transaction coordinator accepts the request to revert the local decision iff it has not issued a global decision for that transaction. Upon that, the requesting node replicates the reversion of the decision inside its bucket, then it releases the locks and allows the other transactions to proceed.

Our objective with this ordering policy is to ensure liveness without compromising safety. Note that in an extreme scenario this might lead to some starvation. However, this is unlikely since clients generate *txIDs* in ascending order, and thus any transaction that experiences momentary starvation will eventually have the lowest identifier. Nevertheless, this ordering policy and/or identifier generation procedure could be replaced by others that further reduce the potential for starvation (e.g. assigning ranges of identifiers to clients).

## 4.5 Fault Tolerance

We now discuss how SconeKV handles failures and, generically, any membership changes. Recall that, although all nodes in the system belong to a single membership group, each bucket effectively works as an independent state machine with linearizable semantics.

As discussed before, the membership layer monitors nodes and provides a consistent view to all correct processes even in the presence of failures. This by itself does not

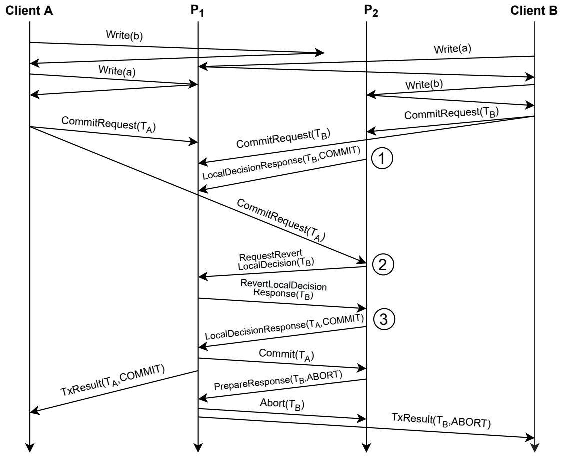


Fig. 4: Messages exchanged during two concurrent transactions that need to acquire locks for the same keys. (1) Primary $P_2$ begins by locking key $b$ for transaction $T_B$, (2) later receives transaction $T_A$ (which also requires the lock for $b$ and has a lower identifier) and thus asks to revert the first decision, in order to release the lock. The request is accepted (3) and $T_A$ commits, after which $T_B$ aborts. Replication was omitted for simplicity.

guarantee that our system exhibits correct behaviour in the presence of faults. For example, if the primary of a bucket participating in a transaction fails before externalizing its local decision, all other primaries would wait indefinitely for its response without reaching a global decision nor releasing the locks they acquired. To address this, we once more leverage Viewstamped Replication's [37], [38] view-change algorithm with minor adaptations that do not affect the main protocol logic (the algorithm details were omitted due to space constraints). First, since buckets work as independent state machines, view-changes are restricted to the buckets of their respective nodes, *i.e*, if a node $n$ is added or removed from bucket $i$ in a membership update, this does not affect any bucket $j$, where $j \neq i$. Second, the dedicated membership layer allows to remove the failure detection logic from the state machine replication layer. Third, as the membership layer provides consistent views across all members, we do not need a leader election to determine the primary of the bucket. We simply require a deterministic function such that any node from inside or outside the bucket or a client can determine the primary of a bucket by knowing its participants. This eliminates the need for leader-election and facilitates communication with the client. In fact, the client has a copy of the view which provides one-hop access to all the buckets without the need for extra synchronization between client and server nodes.

Finally, inside a given bucket, a view change results in one of three scenarios. First, the simplest scenario is when a replica failed and was removed from the membership. If it happens, the primary remains the same and the bucket remains available, assuming there are at least $2f + 1$ nodes in the bucket. Second, if a new node joins the bucket, but the primary remains the same, then this new node becomes a replica and requests a state exchange to bring it up to date. Third, the view change results in a change of the primary for that specific bucket. This can happen either because the previous primary failed and was removed from the membership group or because a new node joined the bucket and the deterministic function that selects the primary determines that the new node should be the primary. In either case, we execute the view change algorithm presented in [38]. This guarantees that the new primary has the most up-to-date log from all replicas, and thus guarantees that the bucket remains consistent. During the view change, a bucket can be temporarily unavailable, as a stable bucket (meaning it is not undergoing a view change affecting its primary) is required to ensure serializability.

The safety of 2PC is ensured as follows. The primaries keep information about each ongoing transaction (Algorithm 1 in lines 6–9). More precisely, the read-write set and the set of buckets involved in the transaction are replicated before the local decision is externalized, while the transaction state is replicated upon each modification as described earlier. After a view change that results in a change of primary for a specific bucket, and before the bucket becomes available for new requests, the new primary checks all currently active transactions involving that bucket (meaning transactions whose state is not *committed* nor *aborted*), acquire all the necessary locks (when appropriate), and determine whether its bucket should act as the coordinator of that transaction or not. If so, it requests local decisions from all other buckets, after which it performs the second phase of 2PC as normal. Otherwise, the new primary asks the coordinator if the system reached a global decision during the view change, and acts accordingly.

### 4.6 Correctness

We sketch a correctness proof for the system, which follows from the correctness of its individual building blocks.

In particular, as far as safety is concerned, the upper layer consists of a classical 2PC protocol, which was proven [39] to ensure that all participants in a transaction agree on its outcome and this outcome can only be a commit if all participants agree to commit (i.e., that the reads and writes of the transaction are compatible with a serialization of all transactions). The participants of this protocol are the buckets. The state of the buckets is maintained by Viewstamped Replication which ensures linearizable semantics and implies that each bucket behaves as a single centralized node with correct behavior even across changes to the bucket replicas [37], [38]. However, VSR requires the participants to receive a consistent set of views, with the property that all nodes agree on the contents of each view. That safety property, in turn, is ensured by the design of the membership layer [10], [11]. Note that there is an inevitable delay between a view change and that change being conveyed to the upper layers but such discrepancies between real instantaneous state and what is perceived by nodes still exist in tightly coupled systems. Although this could be exacerbated in a layered system, such delays never compromise safety as explained above.

A similar analysis applies to liveness. In particular, the termination property of 2PC states that all processes eventually decide if there are no failures [39]. This is enforced by the lower layer, since VSR is able to mask individual node failures and ensure that the replicated system as a whole makes progress, as long as the system moves between views until a view containing a set of non-faulty nodes with network links that meet synchrony bounds is reached. This condition is met by the properties of the membership layer that enables it to replace faulty nodes [10], [11], and by our partial synchrony assumptions, which are required for any system that can be used to solve consensus with a single faulty node [40].

Finally, it is important to clarify that none of the modifications we introduce in these protocols break any of their safety and liveness properties. This is true because the only modification to VSR was to replace its fault detection mechanism with the membership layer, and using the latter's view identifiers as VSR's view numbers. This does not affect the correctness of VSR, as the protocol itself is unchanged.

## 5 Implementation

SconeKV is implemented in Java 13 and Kotlin, following an event-based architecture. Every time an event is triggered, it is added to an event queue and processed by worker threads. This allows the implementation to closely follow the algorithms and rationale presented in §4. The communication is done via TCP using ØMQ which provides the abstraction of an asynchronous message queue, and Cap'N Proto is used for message serialization. To provide durability, in the event of a catastrophic failure, updates are batched and persisted to disk using RocksDB with a configurable time period, following an approach similar to Cassandra [6]. For the membership layer we rely on PRIME [11] but other scalable and consistent implementations such as Rapid [10] could be used. The full implementation of SconeKV consists of 5500 lines of code. We now discuss some implementation optimizations.

**Fast Aborts** The algorithm presented in §4.3 can lead to long lock queues on frequently accessed keys, especially when running workloads with skewed key access distributions. All transactions on the queue for a key expect a specific version. If a write on that key occurs, the version is incremented and therefore all transactions waiting in the queue for that key can be immediately aborted.

**Read-Write Locks** To mitigate potential long lock queues for popular keys, we use read-write locks, which allow for greater parallelism in read-intensive workloads.

**Request Targets** Clients can select which nodes they wish to connect to when performing requests. To ensure serializabilty, commit requests are always sent to the primaries. However, read, write, and delete requests can be addressed to either primaries or replicas. Targeting replicas provides a much better load balance but can increase the percentage of aborted transactions, depending on the workload, as they can have slightly outdated versions of the values.

## 6 Evaluation

We evaluated SconeKV and compared it with two other state-of-the-art industrial systems, Cassandra [6] and CockroachDB [8], [9]. We selected those systems because they are mature and have a wide usage in the industry, and also because they represent different points in the consistency versus scalability spectrum: Cassandra provides eventual consistency and good scalability, while CockroachDB offers strong consistency but scales poorly as we will show.

To make a fair comparison with the other systems, Cassandra was configured to use quorums on both reads and writes, although this change is not enough to consider it strongly consistent given its optimistic replication protocol.

We selected two benchmarks: YCSB [12], as it is the *de facto* standard for evaluating cloud based data stores, and TPC-C [13], a standard OLTP benchmark. We evaluate each system according to the following metrics:

- **Throughput** - the number of operations/transactions performed per second.
- **Goodput** - because SconeKV and CockroachDB are transactional, not all operations are guaranteed to commit. Goodput is the fraction of the throughput which corresponds to the number of committed operations/transactions per second.

The evaluation is organized as follows. §6.1 presents the results for YCSB workloads with an increasing number of clients with a small cluster of 20 nodes. §6.2 studies the scalability of the systems running YCSB benchmarks with an increasing number of servers. §6.3 details the experiments performed using the TPC-C benchmark targeting transaction processing. §6.4 evaluates the fault recovery capabilities of the systems.

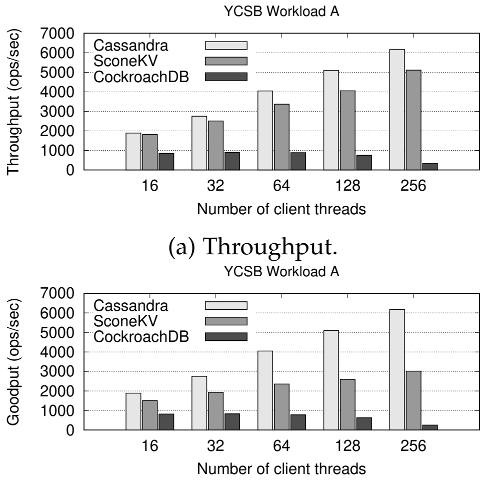
### 6.1 YCSB - Small cluster

The first phase of our experimental evaluation was performed using Docker containers on a cluster with 6 physical machines, with one machine dedicated to running the clients and the others dedicated to the servers. The machines were equipped with 40GB of RAM and 8 Core Intel Xeon E5506 2.13GHz processors.

Each system was deployed in a cluster of 20 nodes (containers) with a replication factor of 4. In the case of SconeKV this corresponds to 5 buckets of 4 nodes each. SconeKV uses all optimizations presented in §5.

We built an YCSB driver for SconeKV and extended the existing CockroachDB JDBC driver to provide transactional support. In both cases, each transaction corresponds to 5 operations grouped together. We selected the core workloads provided by YCSB, in detail: Workload A (50% read, 50% write), Workload B (95% read, 5% write), Workload C (100% read) and Workload F (read-modify-write). All workloads were run using a skewed *i.e.* zipfian distribution leading to 80% of the operations being performed on the *hotset* (20% of the keys), as this is more representative of real world workloads [12]. Each experiment (combination of workload, number of clients and number of servers) was run three times, with a duration of 300 seconds.

**YCSB Workload A** The throughput and goodput results for this write-intensive workload are shown in Figure 5a and Figure 5b, respectively. SconeKV performs in between the baselines, as expected. As it is possible to observe, Cassandra scales well with an increasing number of clients, while CockroachDB demonstrates that it does not handle well write heavy workloads. This is explained by their design based on classical consensus which is a costly primitive. As expected, SconeKV provides a good compromise between



(a) Throughput.

(b) Goodput.

Fig. 5: Throughput and Goodput for YCSB Workload A with an increasing number of clients.

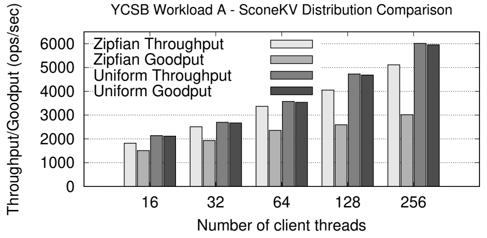
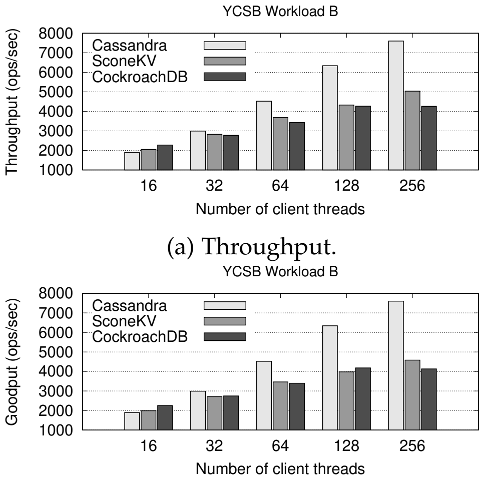


Fig. 6: Throughput and Goodput for SCONEKV in YCSB Workload A, with *zipfian* and uniform distributions.

the raw performance of Cassandra and the strong consistency guarantees of CockroachDB. A closer look comparing the throughput and goodput of SCONEKV shows a significant transaction abort rate, around 40% at its highest (256 concurrent clients). This is justified by the distribution of the requests as the highly skewed workload inevitably leads to an extremely high number of concurrent updates on the same keys, which cannot be serialized. Interestingly, CockroachDB reaches an abort rate of 22% with 256 concurrent clients, but by only committing 254 operations per second, thus further illustrating that consensus-based systems scale poorly. In fact, SCONEKV achieves 11 times more goodput than CockroachDB (and 15 times more throughput). To further study this behaviour in SCONEKV, we ran an additional workload with an uniform distribution (writes and reads are evenly distributed across all keys). The results are depicted in Figure 6. As it is possible to observe, the throughput and goodput are almost identical due to the lower write contention that leads to fewer aborts.

**YCSB Workload B** The throughput and goodput results for this read-intensive workload are shown in Figure 7a and Figure 7b, respectively. The results show that SCONEKV scales, although at a lower rate than Cassandra. This can be explained by the fact that, from a transactional standpoint, SCONEKV does not differentiate writes from reads, applying the same protocol to decide the transactions' outcomes. It is noteworthy that CockroachDB's performance stagnates after 128 concurrent clients, demonstrating that even a 5% update rate is enough to negatively affect its scalability. The abort rate on this workload is substantially lower than for Workload A due to the low rate of concurrent updates. Therefore, for all systems, the achieved goodput is very close to the throughput, as can be observed in Figure 7b.



(a) Throughput.

(b) Goodput.

Fig. 7: Throughput and Goodput for YCSB Workload B with an increasing number of clients.

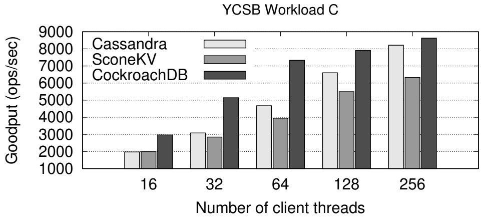
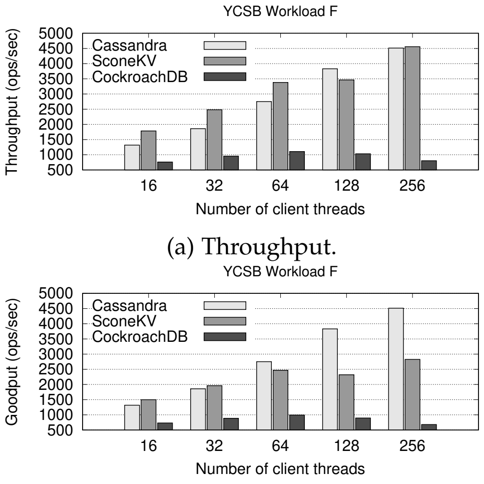


Fig. 8: Goodput for YCSB Workload C with an increasing number of clients.

**YCSB Workload C** This is a read-only workload and, therefore, we only present the goodput (Figure 8), as it is identical to the throughput since there are no updates. Interestingly, for this workload, CockroachDB exhibits better performance than Cassandra. This can be explained in two ways: on one hand, CockroachDB's does not need to acquire any locks for read-only transactions, and, on the other hand Cassandra was configured to use a quorum of reads instead of a single read, to provide better consistency guarantees. SCONEKV does not achieve the same raw performance as either Cassandra or CockroachDB, but it scales with an increasing number of clients. We attribute this difference to the fact that the other systems have been highly optimized over the years, but with additional engineering effort it should be possible to improve SCONEKV's raw performance.

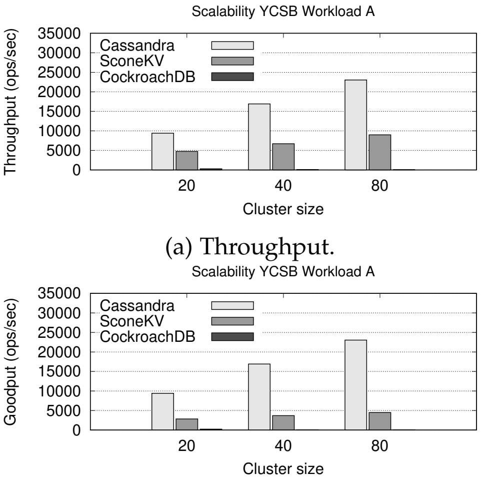
**YCSB Workload F** This workload selects a key following the distribution of requests, and reads, modifies, and writes to it. The results for the throughput and goodput are depicted in Figure 9a and Figure 9b, respectively. Once more, we observe that CockroachDB does not scale with update-intensive workloads. SCONEKV shows better performance than Cassandra in terms of throughput. This can be explained by the fact that SCONEKV is a transactional data store and thus, if inside the same transaction a client performs multiple operations on the same key, only the first operation results in an external request to retrieve the



(a) Throughput.

(b) Goodput.

Fig. 9: Throughput and Goodput for YCSB Workload F with an increasing number of clients.



(a) Throughput.

(b) Goodput.

Fig. 10: Throughput and Goodput for YCSB Workload A with an increasing system size.

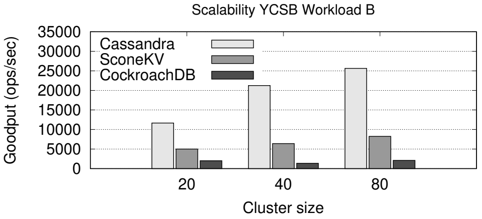


Fig. 11: Goodput for YCSB Workload B with an increasing system size.

version (all others will be handled by the client library, without the need for extra RTTs). However, the highly skewed distribution of requests leads to a high abort rate as can be observed in the goodput results (Figure 9b).

### 6.2 YCSB - Scalability

Due to resource constraints, and the need for more servers, we ran the next set of experiments in the Google Cloud Platform rather than in our premises. We used `e2-highmem-4` instances (each with 4 vCPUs and 32 GB of memory), deploying 4 nodes per instance. For each system, we deployed 20, 40 and 80 nodes using update-intensive (Workload A) and read-intenstive (Workload B) workloads. We start with a keyspace of 1M keys and 256 concurrent clients for a cluster of 20 nodes (similar to the deployment used in the previous sections), and increased the workload proportionately with the size of the system, maintaining a replication factor of 4.

The results for the write-intensive workload are shown in Figure 10. As it is possible to observe, for both Cassandra and SconeKV, goodput increseases as number of nodes increases (albeit SconeKV does so with a lower slope), whereas CockroachDB's performance not only is much worse than the other two systems but it also degrades as the system size grows. This stems from the cost of consensus which gets more expensive as the number of nodes increases. As before, it is possible to observe a significant abort rate for SconeKV, reflected in the goodput results (Figure 10b) due to the skewed nature of the workload.

The goodput results for the read-intensive workload are shown in Figure 11. We observe the same pattern as before, Cassandra and SconeKV are able to scale but CockroachDB performance degrades as the system grows due to the costly synchronization primitives.

### 6.3 TPC-C

Next, we evaluated the 3 systems using TPC-C, the industry standard OLTP benchmark. Traditionally this is a SQL benchmark, however, as our system does not support SQL, we used an in-house implementation that uses `read` and `write` operations following the guidelines of the benchmark described in [13]. We used the same experimental setup described in §6.2, starting with 256 concurrent clients and 768 warehouses (3× clients) for a cluster of 20 nodes, increasing the number of clients and warehouses proportionately with the size of the system. We maintained the replication factor of 4 in all experiments. Each experiment was ran 3 times and accounted for 300 seconds (discarding warm-up and cool-down).

The results are shown in Figure 12. Similarly to the results shown in the previous section, we observe that SconeKV and Cassandra are able to scale while CockroachDB is not (achieving negligible throughput with the maximum system size). It is worth noting that TPC-C transactions are much larger than those of our transactional YCSB implementation (15-30 operations per transaction in comparison with 5 operations per transaction). Nevertheless, SconeKV still scales while also using the completely naive partitioning function described in §4 which frequently results in transactions spanning all buckets. A partitioning function fit for the TPC-C workload could improve SconeKV's performance even more. Nonetheless, we consider this orthogonal to the present work.

As a final note, the results shown here for CockroachDB are not aligned with those published in [9]. The reason for this discrepancy is twofold. First, the experiments performed in [9] had much lower contention, increasing system size 5× while increasing the number of warehouses by 10×. Second, the authors do not specify the number of concurrent client threads used in their workload. For those reasons, we

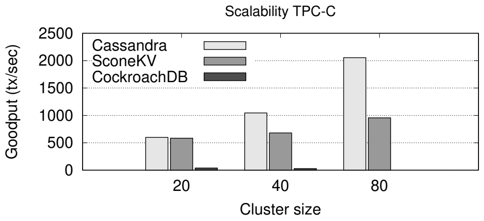


Fig. 12: Goodput for TPC-C with an increasing system size.

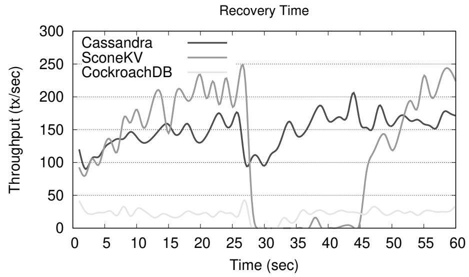


Fig. 13: Recovery time after a fault running TPC-C in a 20 node cluster.

do not consider those results comparable to ours.

### 6.4 Fault Recovery

Finally, we evaluated the fault recovery period of the three systems. We set up a 20 node cluster with a replication factor of 4 and submitted them to a light TPC-C workload with 32 concurrent clients. After warm-up, we crashed a primary node in SconeKV, or a leaseholder in CockroachDB. We omit replica failures as they do not visibly affect SconeKV or CockroachDB results. In Cassandra all nodes are uniform as there is no notion of a primary hence we simply crashed one node at random. Each experiment was run 5 times and Figure 13 shows the average throughput over time.

Cassandra is the least affected due to its design as there is no primary node for a given partition. Both SconeKV and CockroachDB's throughput drops to zero while the membership layer generates a new view. For SconeKV, the performance drop is explained by the fact that the system is small and transactions are large and hence all transactions are likely to touch all the buckets. In a larger system or with smaller transactions, the performance impact would be limited to the set of transactions accessing the faulty bucket. It is worth noting that this is, purposely, a worst case scenario. A failure to any replica would have negligible impact in performance regardless of the time it took for the membership layer to update the view accordingly.

Nevertheless, the majority of the time is spent waiting for a new view to be propagated throughout the cluster (∼16 seconds), after which performance increases to the levels displayed before the fault occurred. If there was another membership component that delivered the new view in less time, it could be integrated into SconeKV and severely reduce the impact of faults to primary nodes. CockroachDB displays a quicker recovery time (∼7 seconds), but, as always, while providing much lower performance.

### 6.5 Discussion

Overall the conducted experiments reveal that SconeKV is able to scale as the load and size of the system increases while still retaining strong consistency guarantees. This contrasts with Cassandra, which scales at the cost of consistency, and CockroachDB, which is not able to scale. In fairness, neither system scales perfectly. Nevertheless, their goodput increases as we increase system size and load, presenting a lower-than-perfect slope. In sum, SconeKV's results support our argument that programmers do not necessarily have to choose between consistency and scalability.

Due to space constraints we omit resource usage results. Briefly, SconeKV did not exhaust the CPU (in contrast with Cassandra during the scalability experiments) and its memory requirements are comparable to CockroachDB (which is written in Go, typically less demanding in terms of memory usage when compared to Java). CockroachDB generally required less resources than the others systems, but also provided much less performance, especially in the scalability experiments.

## 7 Conclusion

In this paper, we aimed at demonstrating that recent advances in distributed computing can lead to interesting new trade-offs in the longstanding tension between consistency and scalability when selecting a key-value store. Our key insight is that by building on top of recent work on scalable and consistent membership services, the fundamental data management aspects of a key-value store, such as data partitioning, replication and transaction processing can be substantially simplified, resulting in a leaner and more scalable design. The resulting system, SconeKV provides a scalable key-value store with strong consistency guarantees. Our comparison with two industrial state-of-the-art systems, Cassandra and CockroachDB, shows that SconeKV is able to scale in all workloads, as opposed to CockroachDB, a database with strong consistency guarantees, and has performance competitive with Cassandra, a database that only ensures eventual consistency - while still offering strong consistency to the application.

## Acknowledgments

This work was partially supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under projects UIDB/50021/2020, PTDC/CCI-INF/6762/2020 and project Lisboa-01-0145-FEDER-031456 (Angainor).

## References

[1] I. Stoica *et al.*, "Chord: A scalable peer-to-peer lookup protocol for Internet applications," *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, pp. 17–32, 2003.

[2] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 2218, no. November 2001, pp. 329–350, 2001.

[3] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content-Addressable Network," Tech. Rep., 2001.

[4] B. Y. Zhao *et al.*, "Tapestry: A resilient global-scale overlay for service deployment," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, pp. 41–53, 2004.

[5] J. Kubiatowicz *et al.*, "OceanStore," *ACM SIGARCH Computer Architecture News*, vol. 28, no. 5, pp. 190–201, 2000.

[6] A. Lakshman and P. Malik, "Cassandra - A decentralized structured storage system," in *Operating Systems Review (ACM)*, vol. 44, no. 2, 2010, pp. 35–40.

[7] Apache. [cassandra-9667] strongly consistent membership and ownership. [Online]. Available: https://issues.apache.org/jira/browse/CASSANDRA-9667

[8] C. Labs. Cockroachdb. [Online]. Available: https://github.com/cockroachdb/cockroach

[9] R. Taft *et al.*, "CockroachDB: The Resilient Geo-Distributed SQL Database," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, vol. 20, 2020, pp. 1493–1509.

[10] L. Suresh, D. Malkhi, P. Gopalan, I. P. Carreiro, and Z. Lokhandwala, "Stable and consistent membership at scale with rapid," *Proceedings of the 2018 USENIX Annual Technical Conference*, pp. 387–399, 2020.

[11] F. Santos, "Prime : Probabilistic membership – large scale membership and consistency," Master's thesis, Instituto Superior Técnico, 2018.

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, 2010, pp. 143–154.

[13] TPC. Tpc-c benchmark. [Online]. Available: http://www.tpc.org/tpcc/

[14] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for internet-scale systems," *Proceedings of the 2010 USENIX Annual Technical Conference*, pp. 145–158, 2019.

[15] LinkedIn. Norbert is a cluster manager and networking layer built on top of zookeeper. [Online]. Available: https://github.com/rhavyn/norbert

[16] A. Das, I. Gupta, and A. Motivala, "SWIM: Scalable Weakly-consistent Infection-style process group Membership protocol," in *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, 2002, pp. 303–312.

[17] A. Dadgar, J. Phillips, and J. Currey, "Lifeguard: Local Health Awareness for More Accurate Failure Detection," *Proceedings - 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*, pp. 22–25, 2018.

[18] L. Lamport, "Fast Paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.

[19] ——, "The Part-Time Parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

[20] ——, "Paxos Made Simple," *ACM SIGACT News*, vol. 32, no. 4, pp. 51–58, 2001.

[21] M. Matos, H. Mercier, P. Felber, R. Oliveira, and J. Pereira, "EpTO: An epidemic total order algorithm for large-scale distributed systems," *Proceedings of the 16th Annual Middleware Conference*, pp. 100–111, 2015.

[22] G. DeCandia *et al.*, "Dynamo: Amazon's highly available key-value store," in *Operating Systems Review (ACM)*, 2007, pp. 205–220.

[23] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger semantics for low-latency geo-replicated storage," *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, pp. 313–328, 2013.

[24] ——, "Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, 2011, pp. 401–416.

[25] S. Almeida, J. Leitão, and L. Rodrigues, "ChainReaction: A causal+ consistent datastore based on chain replication," *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 85–98, 2013.

[26] J. C. Corbett *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems*, vol. 31, no. 3, 2013.

[27] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," *Proceedings of the 2014 USENIX Annual Technical Conference*, pp. 305–319, 2019.

[28] S. Kulkarni *et al.*, "HLC: Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases," *Opodis*, 2014.

[29] M. Brooker, T. Chen, and F. Ping, "Millions of tiny databases," in *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*, 2020, pp. 463–478.

[30] K. Sonbol, O. Özkasap, I. Al-Oqily, and M. Aloqaily, "Edgekv: Decentralized, scalable, and consistent storage for the edge," *Journal of Parallel and Distributed Computing*, vol. 144, pp. 28–40, 2020.

[31] S. Peluso, P. Romano, and F. Quaglia, "SCORe: A scalable one-copy serializable partial replication protocol," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7662 LNCS, 2012, pp. 456–475.

[32] A. Hentschel, D. Shirley, L. Lafrance, and M. Zamski, "Flow: Separating consensus and compute–execution verification," *arXiv preprint arXiv:1909.05832*, 2019.

[33] Y. Hassanzadeh-Nazarabadi, A. Küpçü, and O. Özkasap, "Lightchain: Scalable dht-based blockchain," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 10, pp. 2582–2593, 2021.

[34] Y. Hassanzadeh-Nazarabadi and S. Taheri-Boshrooyeh, "A consensus protocol with deterministic finality," in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2021, pp. 1–2.

[35] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, Apr. 1988.

[36] D. Karger *et al.*, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web," in *Conference Proceedings of the Annual ACM Symposium on Theory of Computing*, 1997, pp. 654–663.

[37] B. M. Oki and B. H. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, 1988, p. 8–17.

[38] B. Liskov and J. Cowling, "Viewstamped Replication Revisited," *IEICE Transactions on Information and Systems*, no. MIT-CSAIL-TR-2012-021, pp. 1—-14, 2012.

[39] N. A. Lynch, *Distributed algorithms*. Elsevier, 1996.

[40] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, 1985.

**João Gonçalves** is a PhD student in Computer Science, a Teaching Assistant at Instituto Superior Técnico, University of Lisbon, and a Junior Researcher at INESC-ID Lisbon. His research interests are distributed systems, specifically group membership, scalable storage, and bug detection and reproducibility. Currently, he is researching Persistent Memory with a focus on crash-consistency testing.

**Miguel Matos** is a professor at Instituto Superior Técnico (ULisboa) and a researcher at INESC-ID. His research interests lie in the area of distributed systems, in the subjects of scalability, performance, correctness and systems evaluation. In particular, he is conducting research in blockchain and related problems, consistency and scalability in large scale databases, systems evaluation under faults and experimental reproducibility. Miguel has been involved in several EU, national, and industry funded research projects. He has publications in venues such as SOSP, TPDS, JPDC, Eurosys, ICDCS, DSN, IPDPS, Middleware and SRDS.

**Rodrigo Rodrigues** is a professor at the Instituto Superior Tecnico (ULisboa) and a researcher at INESC-ID since 2015. Previously, he held faculty positions at the Nova University of Lisbon, and the Max Planck Institute for Software Systems (MPI-SWS), where he led the Dependable Systems Group. He received his PhD degree from the Massachusetts Institute of Technology (MIT) in 2005, under the supervision of Prof. Barbara Liskov. He won several fellowships and awards, including a best paper award at SOSP, a special recognition award from MIT's Department of EECS, an ERC starting grant, and a Google faculty research award.