SconeKV: Strongly CONsistEnt Key-Value Store

Extended Abstract

João Gonçalves
Advisors: Miguel Matos and Rodrigo Rodrigues
Instituto Superior Técnico, Universidade de Lisboa

***Abstract*—Relational databases provide a strong foundation for constructing applications due to their ACID properties, which come at the cost of synchronization. Modern distributed applications reached such a scale, both in terms of the amount of data and the number of concurrent clients, that traditional databases cannot sustain it, resulting in performance degradation. Other distributed storage solutions appeared that scaled horizontally by employing optimistic replication protocols that only guaranteed eventual consistency. Besides offering weaker consistency, many modern key-value stores have difficulty maintaining their guarantees at a large scale, with higher latency and dynamism of the membership. These systems offer better performance but provide a less stable foundation to build applications, as they allow for concurrent data updates whose conflicts need to be resolved by the application developers.**

In this thesis, we propose SconeKV: a distributed key-value storage system with strong consistency guarantees for large scale deployments. SconeKV offers serializable, distributed transactions. It leverages a membership layer with strong probabilistic guarantees and a design based on horizontal partitioning, reducing the synchronization required between nodes, while still employing consistent replication protocols and thus providing strong consistency to clients. Experimental results show that SconeKV performs better than CockroachDB in write heavy workloads whilst being competitive with Cassandra in all workloads.

I. Introduction

Distributed systems began at a much smaller scale than today. Initially, these systems were comprised of a few nodes connected to a centralized database for storage. Relational databases provide applications with a strong foundation, offering transactional support and strong consistency on client operations. Today the paradigm has changed. Users demand that systems are always available, leading to a shift to cloud computing. Traditional relational databases are able to scale vertically, but now systems require databases that scale horizontally, are always available and with low latency, anywhere in the world.

Previous to this, an initial approach to distributed scalable storage were peer-to-peer distributed hash tables [1]–[3] which provided the location of objects at a large scale, with data replication and fault tolerance but no consistency guarantees. Traditional DHTs employ only partial views, meaning that each node only knows a subset of the system, and because of that located objects in $O(\log n)$ hops. Modern key-value stores, namely Dynamo [4] and Cassandra [5], base themselves in some of the same principles as DHTs but with more robust guarantees. These systems combine high availability and reliability with low latency for requests across the globe. However, these properties come at the cost of optimistic replication protocols and weaker consistency guarantees when compared with relational databases. Moreover, the membership solutions employed by these systems do not maintain the same guarantees with the higher node turnover or churn. Concretely, Cassandra has an open ticket since 2015 [6] reporting difficulties in maintaining consistency guarantees during membership changes. These shortcomings are emphasized with the dynamism of large scale deployments.

Today, applications once again require strong consistency in data storage, but now at a global scale while still being highly available. Distributed relational databases rely on classical consensus, imposing a high complexity in the synchronization between nodes. To achieve strong consistency at a high scale, systems need to be designed in such a way that reduces the synchronization requirements.

The purpose of our work is to develop a distributed storage system with strong consistency guarantees at a large scale. The goal is to combine the scale of peer-to-peer and eventually consistent key-value stores, with the strong consistency and ACID properties of relational databases. To achieve this, a system should not rely on optimistic replication nor employ a weakly consistent membership solution, as these approaches scale well but at the cost of the overall consistency guarantees that the resulting solution is able to provide to target applications. In this thesis, we present SconeKV: a transactional key-value store with strong consistency guarantees even in large scale deployments. The main objectives behind the design are to: maintain a total view of the system with strong consistency guarantees, guarantee strong consistency on all operations, provide multi-object distributed transactions, guarantee the replication factor, and tolerate membership changes. We employed a scalable membership solution with strong probabilistic guarantees, horizontal partitioning, consistent replication and a two-phase agreement protocol, all while guaranteeing tolerance to membership changes. The system was evaluated in comparison with two state-of-the-art systems: Cassandra [5], a highly scalable, eventually consistent distributed key-value store; and CockroachDB [7], a distributed SQL database with ACID properties built on top of a transactional and strongly consistent key-value store. Experimental results show that SconeKV performs better than CockroachDB in write heavy workloads whilst being competitive with Cassandra in all workloads, proving that SconeKV scales whilst providing serializable distributed transactions.

II. Background and Related Work

In this section, we present a bottom-up view of a distributed storage system. Most distributed applications require the notion of a view, a list of correct nodes belonging to the system. This view can either be partial or total. Partial views can either be structured [1]–[3] (imposing specific linkage between nodes) or unstructured, as is the case of Peer Sampling Services (PSS) [8]–[10]. Total views imply that every node knows every other node in the system. Distributed solutions to this problem trade-off scalability and performance for view consistency, some offering weak consistency guarantees [11], [12] and others strong [13], [14]. PRIME [15] offers strong probabilistic guarantees at a large scale, leveraging EpTO [16], a total order communication algorithm based on gossip.

Replication protocols provide a similar trade-off between consistency and scalability. Solutions based on classical consensus [17] provide strong consistency but exhibit poor scalability. State Machine Replication [18]–[20] can offer better scalability if paired with horizontal partitioning (sharding). Many distributed storage systems opt to weaken their consistency guarantees in order to scale, employing optimistic replication protocols, as is the case with Dynamo [4] and Cassandra [5]. Google Spanner [21] is a highly scalable SQL database that shards data across many Paxos [17], [22] state machines, relying on GPS and atomic clocks to order transactions. It provides strong consistency at scale but requires specialized hardware to do so.

III. SconeKV

In this section we present SconeKV, a distributed key-value store designed to provide strong consistency guarantees at a large scale.

A. Overview

SconeKV is a complete system, with a fully modular architecture, constituted by four different layers, each one providing the required properties to the ones above. The bottom layer is a PSS (CYCLON [8]), followed by a Total Order Group Communication layer (EpTO [16]) and a Membership Manager (PRIME [15]). The top layer is the main focus of our work: a distributed key-value store with strong consistency guarantees, fit for large scale deployments.

SconeKV uses an identifier ring, dividing it into sections called buckets - horizontal partitioning. Both nodes and items are then assigned to buckets (rather than points in the space) using consistent hashing. This approach allows for reconfiguration of the system, for example, resizing buckets in order to better distribute data or request processing load. Each data item is present in a single bucket and is managed and replicated by the set of nodes that constitute that bucket. Each bucket works as in primary-backup, one node is the master whilst the remaining nodes are replicas. As the membership layer provides a consistent view of the system to all participants, there is no need for a leader election or another consensus variant to determine the master, it simply requires a deterministic function using the bucket members as input (for example, the node with the lowest identifier).

B. Protocol

SconeKV offers three different operations: `read(key)` returns the current value for that key, `write(key, value)` updates the key with the given value and `delete(key)` removes the key-value pair from the store. Values are arrays of bytes, opaque to the system, and each consequent write to the same key overwrites the value written by the previous one.

Each key-value pair is also associated with a version. Versions are returned by all operations and represent the moment in that object's life cycle where a given operation should be performed. A key's version is initially 0 and incremented by one with each write operation on that key.

Each operation corresponds to a request to a SconeKV node (master or replica, depending on the client configuration) inside the bucket corresponding to the key accessed in the operation, receiving the current version, and in the case of a read also the current value of the key. The client must perform operations inside a transaction, while the library maintains the read/write set with the versions observed for each key.

To externalize the operations, the client issues `commit` on a given transaction. This operation will be successful or unsuccessful, depending if the versions seen by the transaction (those in the read/write set) match the actual current versions for each key in the system at the time of the commit.

C. Distributed Transactions

Depending on the keys accessed and the system configuration, a transaction can span multiple buckets. Buckets function according to primary-backup, thus each transaction is agreed on by the masters of the buckets accessed. This agreement is achieved using a two-phase commit protocol led by the master with the lowest identifier - the transaction coordinator.

Figure 1 provides an example for the messages exchanged during a transaction spanning multiple buckets. At commit time, the client sends each master involved in the transaction a subset of the read/write set containing only the keys assigned to each bucket, and also a list of all buckets involved. Once the commit request is received, each master makes a local decision based on the versions required by each operation in a transaction, acquiring the required locks and replicating the potential writes (if the transaction was locally accepted) and the local decision. Once replicated, the local decision is communicated to the transaction coordinator, which corresponds to the first phase of the protocol (indicated in Figure 1).

When all masters communicate their local decisions to the transaction coordinator, they reach a global decision - the second phase of the protocol (also indicated in Figure 1). Following that, the global decision is replicated, the potential writes are committed or aborted, and all locks are released. According to the two-phase commit protocol, a transaction is only committed if all participants accept the transaction

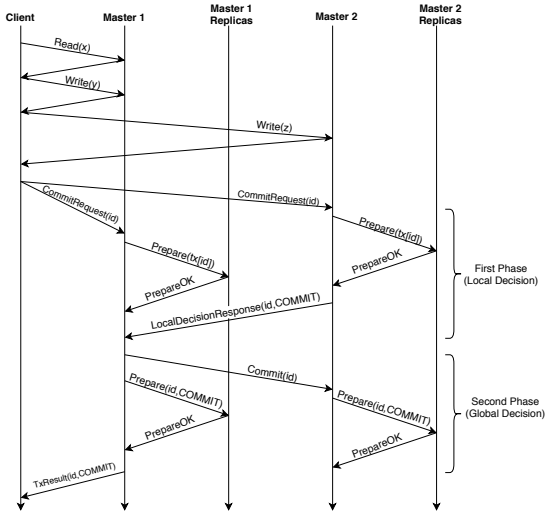

Fig. 1: Messages exchanged during a transaction involving 2 buckets. Master 1 is the transaction coordinator, both masters accept the transaction locally and thus it is committed. For simplicity, the replicas for each bucket are represented as a single entity each. The first phase of the protocol is defined in Algorithm 1, the second phase was omitted due to space constraints.

locally, whilst it takes a single local rejection to abort the entire transaction.

The 2PC protocol has a weak liveness property. We counteract that by having a membership layer with strong consistency guarantees. This means that it will detect a node failure and consistently inform all participants. We also replicate the state inside each bucket (explained further in Section III-E) so that any master may be replaced by a replica to complete the protocol.

Figure 2 exemplifies a numbered sequence of events that are triggered when the client attempts to commit a transaction, in order for SconeKV nodes to decide the transaction outcome. Once the client issues `commit`, the first phase of the protocol begins with each involved master receiving the request and triggering MAKELOCALDECISION (Algorithm 1, lines 1-23, some of the auxiliary functions are omitted due to space constraints). This verifies if the versions accessed are valid (lines 34-42), meaning the transaction is still serializable. It will then determine if it can acquire locks for all accessed keys (lines 4 and 5), queuing them all if any of them was already acquired by another transaction. Assuming it successfully acquired the locks, the local decision is propagated to the replicas for fault tolerance (the SMR portion of Figure 2) and, once it is consistently replicated, the local decision is communicated to the transaction coordinator (lines 25-32), finishing the first phase of the protocol.

Once the transaction coordinator receives local decision responses from all participating buckets (or at least one abort decision), it starts the second phase of the protocol, broadcasting the global decision to all participants. Every master involved will then receive it, replicated inside its bucket, and act accordingly, committing or aborting and responding to the client in the case of the transaction coordinator. In either case, the locks are released and MAKELOCALDECISION is triggered for the next transactions in the queues of the released locks.

D. Avoiding Distributed Deadlocks

The design presented thus far was simplified and may lead to distributed deadlocks. Assume that two transactions, T_A and

Algorithm 1 Local Decision - First Phase

```
1:  upon event ⟨master, MakeLocalDecision| txID⟩ do
2:      if txs[txID].state ≠ aborted then
3:          if CheckValidTransaction(txID) then        ▷ validate the versions used in the tx
4:              owners ← GetLockOwners(txID)
5:              if owners = ∅ then
6:                  AcquireLocks(txID)
7:                  txs[txID].state ← prepare-commit
8:                  trigger ⟨Prepare|txs[txID]⟩
9:              else
10:                 QueueLocks(txID)
11:                 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
12:                     for each t ∈ owners do
13:                         otherCoord ← GetCoord(txs[t].nodes)
14:                         send ⟨RequestRevertLocalDecision| otherTxID⟩ to otherCoord
15:                     end for
16:                 end if
17:             end if
18:         else
19:             txs[txID].state ← prepare-abort
20:             trigger ⟨Prepare|txs[txID]⟩
21:         end if
22:     end if
23: end event
24:
25: upon event ⟨master, SendLocalDecision| txID⟩ do
26:     txCoord ← GetCoord(txs[txID].nodes)
27:     if txs[txID].status = prepare-commit then
28:         send ⟨LocalDecisionResponse|txID,commit⟩ to txCoord
29:     else
30:         send ⟨LocalDecisionResponse|txID,abort⟩ to txCoord
31:     end if
32: end event
33:
34: function CheckValidTransaction(txID)
35:     for each (key, _, version, _) ∈ txs[txID].rwSet do
36:         currentVersion ← GetVersion(key)
37:         if currentVersion ≠ version then
38:             return False
39:         end if
40:     end for
41:     return True
42: end function
43:
44: function GetLockOwners(txID)
45:     owners ← ∅
46:     for each (key, _, _, _) ∈ txs[txID].rwSet do
47:         lockOwner ← GetLocker(key)
48:         if lockOwner ≠ NULL ∧ lockOwner ∉ owners then
49:             owners ← owners ∪ lockOwner
50:         end if
51:     end for
52:     return owners
53: end function
54:
```

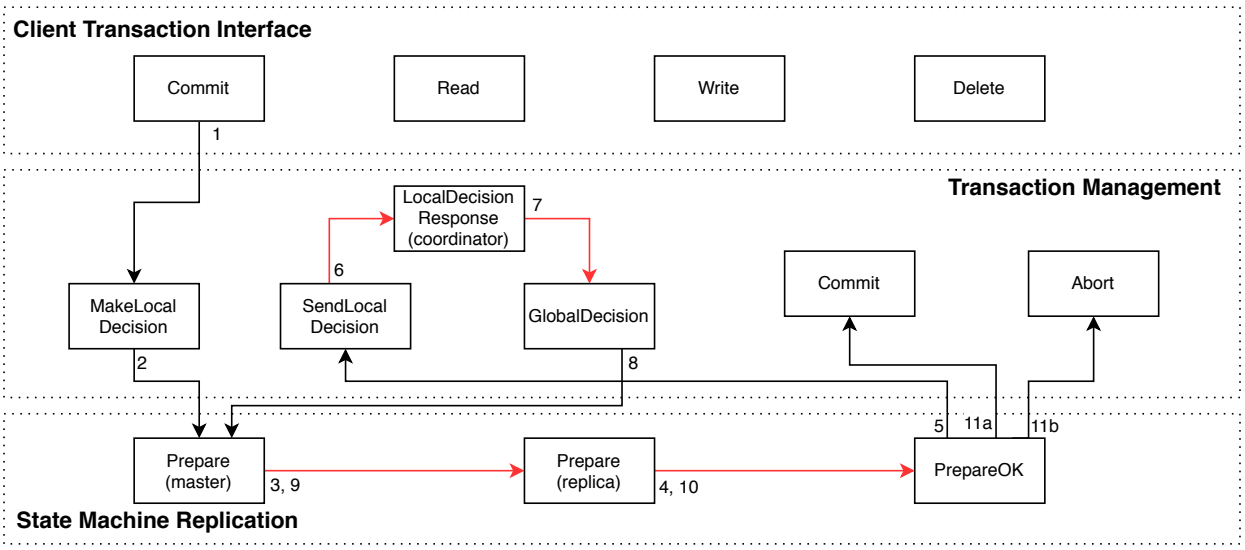

Fig. 2: Diagram showing the sequence of events to commit a transaction in the general case. The red arrows represent that the next event was triggered in a remote node.

T_B, perform writes on keys x and y, which belong to buckets B_1 and B_2 respectively. If the two transactions attempt to commit concurrently, it is possible that the master of B_2 (M_2) receives T_A first, whilst the master of B_2 (M_1) receives T_2 first. In that case, M_1 would lock a for T_A and add T_B to the queue, while M_2 would lock b for T_B and add T_A to the queue. This would generate a distributed deadlock, as neither T_A nor T_B would achieve a global decision, each requiring another local decision.

This is a simple example, in this case one of the transactions would need to abort as they are conflicting. Nevertheless, there are other, more complex examples where both transactions could commit and respect serializability if ordered correctly. Figure 3 exemplifies how SconeKV avoids such a scenario, giving priority to the transaction with the lowest identifier. As such, a master that locally accepted a transaction T_B may ask the transaction coordinator to revert its local decision in order for another transaction T_A with a lower identifier to acquire the required locks. This will be granted if and only if T_B has not yet been agreed on by all its participants.

The protocol is triggered during MAKELOCALDECISION, back in Algorithm 1. If a transaction fails to acquire the locks, it queues them and determines if it should be processed before all the transactions currently owning any of those locks (Algorithm 1, lines 11-17). Assuming it should, the coordinators for the reverted transactions will confirm that those have not reached a global decision yet, accepting the request if that is the case. The requesting node would then propagate the reversion of the decision to the rest of the bucket and, once it is consistently replicated, release the locks, eventually processing MAKELOCALDECISION once more for the transaction that triggered the protocol.

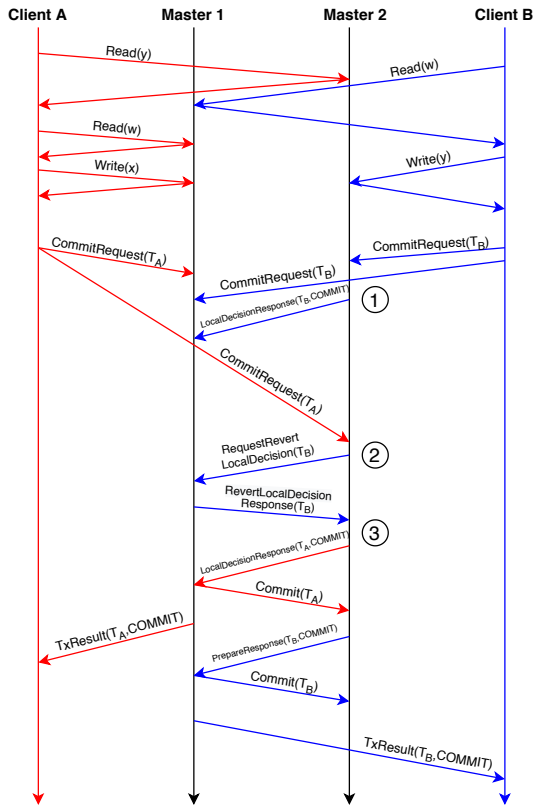

Fig. 3: Messages exchanged during two concurrent transactions that need to acquire locks for the same objects. (1) Master 2 begins by locking object y for transaction T_B, (2) later receives transaction T_A (which also requires the lock for y and has a lower identifier) and thus asks to revert the first decision, in order to release the lock. The request is accepted (3) and both transactions end up being committed. Replication was omitted for simplicity.

E. Replication

In this section, we address our penultimate objective: guaranteeing the replication factor. As stated earlier, inside each bucket, the system works according to a primary-backup scheme. Requests that change the state of the system - commits - are routed through the master and persisted to the replicas for fault tolerance and durability. Masters serialize transactions and thus client operations that observe and/or modify the system state are consistent and their results deterministic. With this behaviour, the system has the properties of a state machine.

The solution was then to apply a state machine replication algorithm inside each bucket, guaranteeing not only that the replication factor is maintained but also that it is done in a consistent manner. Our replication protocol is a concretization of Viewstamped Replication [18], [19]. In summary, entries flow from the master to the replicas. To replicate a log entry, the master issues PREPARE. Once received, each replica processes the entry if and only if it has processed all previous entries, issuing PREPAREOK. When the master receives f PREPAREOK's, the entry is consistently propagated and can be committed.

This protocol assures reliability and availability 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. Note that replication does not guarantee the preservation of data in the case of catastrophic failures, such as all replicas crashing at the same time. In Section IV we discuss how SconeKV provides durability guarantees.

F. View Changes

To conclude the system design, we address our final objective of tolerating membership changes. The membership layer - PRIME - monitors nodes and updates the view accordingly, consistently across all correct members. This by itself does not guarantee that our system exhibits correct behaviour in the presence of faults. For example, if the master of a bucket participating in a transaction fails before communicating its local decision, all other master will wait indefinitely for its response without reaching a global decision nor releasing the locks they acquired. To address this, we once more adapted the Viewstamped replication [18], [19] view-change algorithm to work in this scenario.

First, view-changes implicate buckets independently, meaning if a node n was added or removed from bucket i in a membership update, this does not affect any bucket j, where $j \neq i$.

Next, as we have a dedicated membership layer, we can remove that behaviour from the state machine replication component. Moreover, as this membership layer provides consistent views across all members, we do not need a leader election to determine the master of the bucket, we simply require a deterministic function such that any entity (a node from inside or outside the bucket or a client) can determine the master of a bucket simply by knowing its participants. This eliminates the need for another agreement between the nodes and facilitates the communication with the client: we maintain one hop access without the need for extra synchronization between client and cluster. Clients are not part of the cluster membership, thus when a client starts it requests a view from any node in the system. This view is only updated in case of a timeout contacting a node (resulting in a request of a new view from another node) or if the client sends a request to an incorrect node (wrong bucket and/or incorrect master), in which case the node proactively responds with an updated view of the cluster.

Finally, inside of a specific bucket, a view change can either maintain or change the current master. If the master remains the same and a replica left, the system continues working normally (always with the assumption that there are $f + 1$ correct nodes in the bucket). If the master remains and a new node joins, then this new node is a replica and there needs to be a state exchange to bring it up to date. This occurs once a node receives a PREPARE with an *opNumber* MAXOPNUMBERHOLE entries after its current most recent entry in the log. Now if the current master changes, either because the previous one failed or our function determines that a new node is the master, then we run the view change algorithm (omitted due to space constraints) to guarantee that the next master is consistently chosen and has the most up-to-date log, continuing to provide strongly consistent operations inside that bucket. During the view change, a bucket can be temporarily unavailable, since to guarantee the serializability of transactions it is required to be working in stable conditions.

IV. IMPLEMENTATION

SconeKV was implemented using Java 13. It depends on a Java implementation of PRIME [15] and Kotlin implementations of EpTO [16] and CYCLON [8]. To closely emulate the behaviour presented in III, SconeKV nodes are implemented using an even-based architecture. Every time an event is triggered, it is added to an event queue and processed by one of a configurable number of worker threads. The communication is done via TCP using ØMQ[1] for an asynchronous message queue abstraction and Cap'N Proto[2] for message serialization. Durability is provided in case of catastrophic failures. Updates are batched and persisted to disk using RocksDB[3] and a configurable period, using a similar approach to [4], [5].

SconeKV provides a client library with a simple API, exemplified in Listing 1.

A. Optimizations

a) Fast Aborts: The algorithm presented in Section III-C can lead to long lock queues on frequently accessed keys, especially when running skewed workloads. All transactions on the queue for a specific key expect the current version to be the latest. Thus, if a write occurs, increasing the version of that key, all transactions waiting in the queue can be immediately aborted.

[1]https://zeromq.org
[2]https://capnproto.org
[3]https://rocksdb.org

b) Read-Write Locks: Once more addressing the issue of long lock queues for popular keys, the introduction of read-write locks allows for greater parallelism in read-heavy workloads.

c) Client Request Modes: The SconeKV API allows for configuration of the request mode: clients can select which nodes they which to connect to. Commit requests need to be routed through the masters of the buckets accessed, while read, write, and delete requests can target the master, replicas, or completely randomize their selection. Targeting replicas provides a much better load balance but can increase the percentage of aborted transactions, depending on the workload, as they can be slightly outdated regarding the versions of the keys (in comparison with the master that always has the most up-to-date version of all keys).

d) Garbage Collection: After transactions are committed and the key-value store is updated accordingly, the transaction data becomes mostly superfluous. Data of completed transactions is thus garbage collected in a configurable interval and according to an also configurable transaction TTL.

```
// Initiate the client
SconeClient client = new SconeClient();
// Creating a transaction
Transaction tx = client.newTransaction();
try {
    // Reading a key 'x' and obtaining its
    // value 'valueX'
    byte[] valueX = tx.read(x);
    // Writing a value 'valueY' on key 'y'
    tx.write(y, valueY);
    // Deleting key 'z'
    tx.delete(z);
    // Attempting to commit the transaction
    tx.commit();
catch (CommitFailedException e) {
    // In case the transaction aborted,
    // manage it accordingly
    (...)
}
```

Listing 1: Usage example of the SconeKV client API

V. Evaluation

We evaluated SconeKV and compared it with two other state-of-the-art distributed storage systems, Cassandra and CockroachDB, one at each end of the consistency spectrum. Cassandra [5] represents the weaker end of the consistency spectrum. It is a highly available and highly scalable distributed key-value store with eventual consistency. It scales both in terms of cluster size and the number of concurrent clients. To make the comparison fairer with the other systems, it 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.

CockroachDB [7] represents the stronger side of the consistency spectrum. It is a distributed SQL database with ACID properties, built on top of a transactional and strongly-consistent key-value store.

The evaluation considered the following metrics:

- **Throughput** - the number of operations processed per second; the main metric to access the scalability of data stores.
- **Goodput** - because SconeKV and CockroachDB are transactional, not all operations are guaranteed to commit. Goodput represents the number of committed operations per second.
- **Consistency** - derived from the design and guarantees provided by the three systems.
- **Resource usage** - CPU, memory, network, and disk utilization.

A. Experimental Setup

The system's experimental evaluation was performed using Docker [4] containers running on a 6 physical machine cluster, with one machine dedicated to running the client benchmark and the others dedicated to the servers. The machines were equipped with 40GB of RAM and 8 Core Intel Xeon E5506 2.13GHz processors.

The systems were deployed in 20 node clusters with a replication factor of 4, representing 5 buckets of 4 nodes in the case of SconeKV. Both the nodes and the client ran in Docker containers for ease of deployment and communication. It is also worth noting that, SconeKV was deployed using all optimizations presented in Section IV-A.

YCSB [23] was selected as the benchmark for our experiments as it is a standard for cloud based data store comparison. We extending it to provide support for transactions (of 5 operations) in the case of SconeKV and Cockroach. The workloads were selected from part of the YCSB core workloads: A (50% read, 50% write), B (95% read, 5% write), C (100% read) and F (read-modify-write). All workloads were run using a skewed distribution - *zipfian* - leading to 80% of the operations being performed on the *hotset* (20% of the population), as this is more representative of real world workloads [23]. Each experiment (combination of workload and number of concurrent clients) was ran three times, with a duration of 300 seconds.

B. Throughput and Goodput

a) Workload A: This is an update heavy workload, the throughput and goodput of the systems are shown in Figure 4 and Figure 5, respectively. SconeKV performs in between the baselines, as expected. Cockroach demonstrates that it does not handle well write heavy workloads, which is explained by their design based on classical consensus. A closer look comparing 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 it is highly skewed and leads to an extremely high number of concurrent updates on the same keys, which cannot be serialized. Interestingly, Cockroach 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. To further study this, we ran an

[4]https://www.docker.com

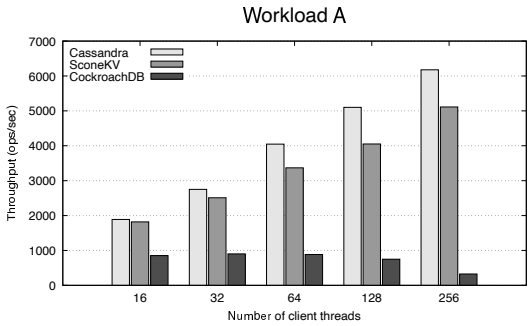

Fig. 4: Throughput for the three systems using YCSB Workload A.

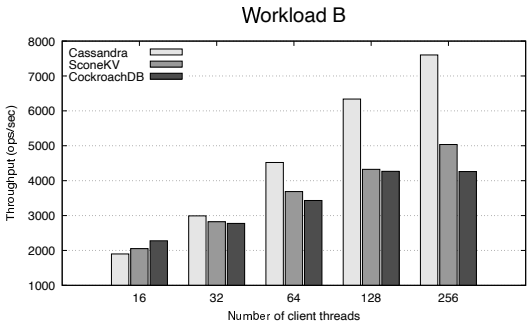

Fig. 7: Throughput for the three systems using YCSB Workload B.

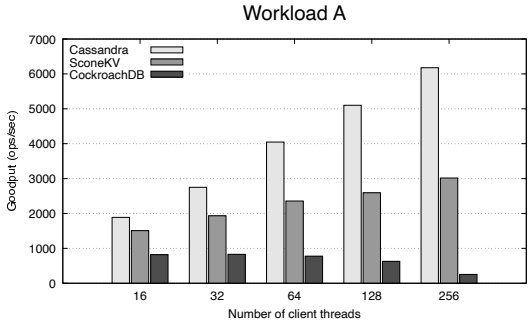

Fig. 5: Goodput for the three systems using YCSB Workload A.

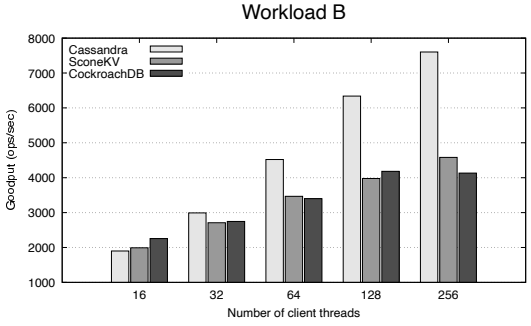

Fig. 8: Goodput for the three systems using YCSB Workload B.

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 in this configuration, throughput and goodput are almost identical, due to the lower write contention that leads to fewer aborts.

b) Workload B: This is a read heavy workload. The results displayed in Figure 7 and Figure 8 show that SconeKV still scales, although at a lower rate than Cassandra. This can be explained, as SconeKV does not differentiate writes from reads, applying the same protocol to decide the transaction outcome. It is noteworthy that Cockroach stagnates when reaching hundreds of concurrent clients, demonstrating that

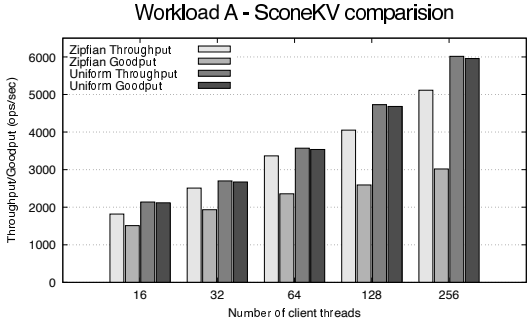

Fig. 6: Comparison of throughput and goodput for SconeKV, depending on the workload distribution (*zipfian* vs uniform)

even a 5% update rate is enough to reduce its scalability. The abort rate with this workload is severely lower for either system in comparison with the previous workload, as can be observed in Figure 8. This is expected, due to the low rate of concurrent updates.

c) Workload C: This is a read only workload. For that reason, we only present the goodput of the three systems (Figure 5), as it is identical to the throughput, given there are no updates to keys. SconeKV does not achieve the same raw performance as its baselines, but still shows constant growth, demonstrating its scalability. Recall that this is only a prototype and thus is less optimized than its competitors. Cockroach exhibits even better performance than Cassandra, although not as scalable. This can be explained in two ways: the fact that the read operations do not acquire locks, according to Cockroach's design; and the fact that Cassandra was configured to use a quorum of reads instead of a single read, to provide better consistency guarantees.

d) Workload F: This workload selects keys according to the distribution of requests, reads, modifies the value, and writes to the same key. Once more, we demonstrate that Cockroach does not scale with update heavy workloads. SconeKV displays even better performance than Cassandra in terms of throughput (Figure 10), however, Figure 11 once more shows that update heavy workloads with a highly skewed distribution result in a high transaction abort rate. The raw throughput can be explained as follows: SconeKV is a transactional data

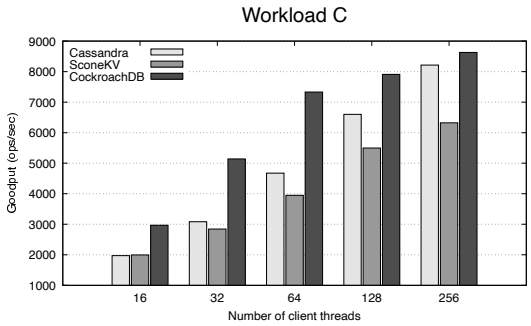

Fig. 9: Goodput for the three systems using YCSB Workload C.

store and thus, if inside the same transaction a client performs multiple operations on the same key, only the first operation will result in an external request to retrieve the version (all others will be handled by the client library, without the need for extra RTTs).

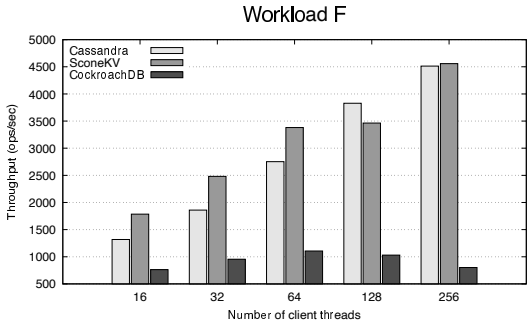

Fig. 10: Throughput for the three systems using YCSB Workload F.

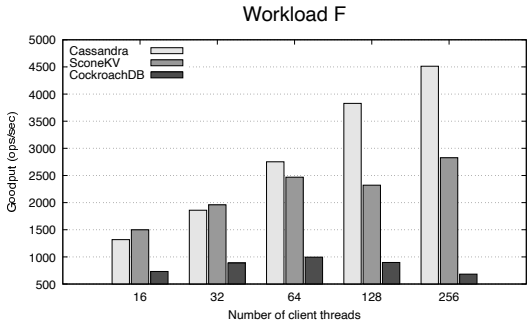

Fig. 11: Goodput for the three systems using YCSB Workload F.

C. Resource Usage

In this section, we analyze the the resource usage of the three systems, namely, CPU, memory, network, and disk. Every metric will be presented during an update-heavy workload (YCSB Workload A) and a read-only workload (YCSB Workload C). These metrics were captured using `dstat` in each of

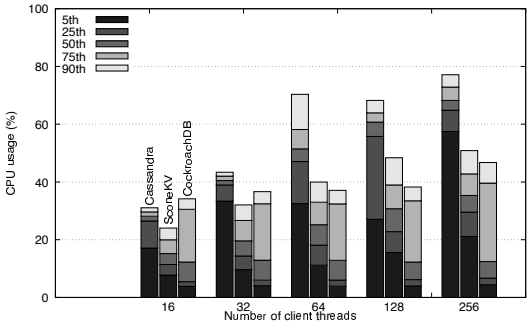

Fig. 12: CPU usage during update heavy Workload A.

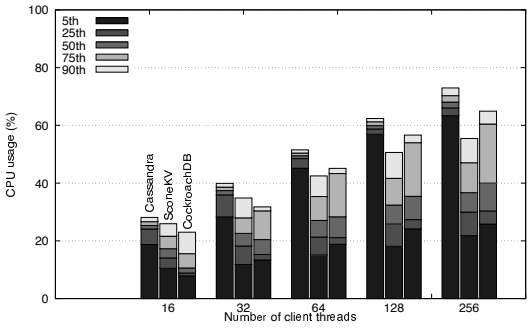

Fig. 13: CPU usage during read-only Workload C.

the 5 physical machines running the containers for the storage nodes, to minimize the impact of these measurements. The results displayed represent the percentiles for those metrics.

a) CPU Usage: Figure 12 and Figure 13 depict CPU usage over time inside each physical machine for benchmarks of 16, 32, 64, 128, and 256 clients. We used stacked bars with decaying shades of gray to represent the distribution using a set of percentiles. For example, the medium shade gives the median value, while the lighter shade gives the 90^{th} percentile, meaning 90% of the time nodes have a lower CPU utilization. Despite Cassandra utilizing more CPU time than the others, none of the three systems exhausted their CPU resources. These results are acceptable given that this is a primary service and thus, in most cases, is deployed in its own dedicated machines. We can also conclude that the CPU will not represent a bottleneck in the scalability of SconeKV.

b) Memory Usage: Figure 14 and Figure 15 show the memory usage over time in the physical machines in the form of stacked bars representing the percentiles. We can conclude that Cassandra requires much more memory than the other two systems. Cassandra is implemented in Java which helps to explain the high memory requirements. Cockroach uses less memory than SconeKV or Cassandra in the update heavy workload, but it also provides much less throughput (shown in Figure 4). With a read heavy workload (in which it surpassed the other two systems in terms of throughput) it required more memory than SconeKV but still less than Cassandra. This can be justified by its implementation being in Go, which is typically less demanding than Java in terms of memory.

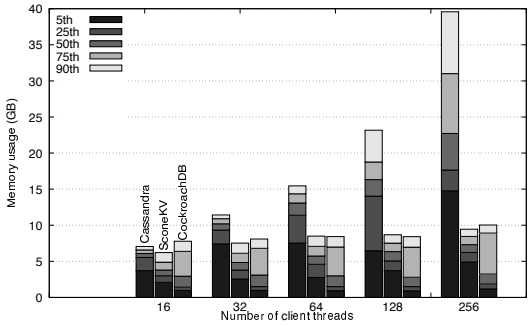

Fig. 14: Memory usage during update heavy Workload A.

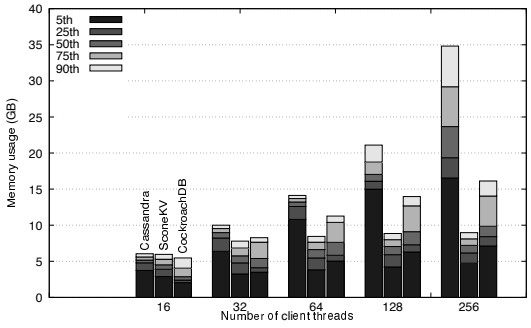

Fig. 15: Memory usage during read-only Workload C.

c) Bandwidth Usage: For this metric, we measured the bytes sent and received over time in the physical machines, once again constructing percentile graphs. Figure 16 and Figure 17 depict the bandwidth usage in terms of upload, while Figure 18 and Figure 19 show it in terms of download. We can conclude that SconeKV's bandwidth usage is on par with Cassandra's, for both upload and download. Cockroach exhibits less median usage for both upload and download for the update heavy workload, which is understandable given it is responding to much fewer requests per second. In the read-heavy workload it uses much more bandwidth than its competitors, both in terms of upload and download, for similar reasons. It is also important to note that classic consensus-based protocols have quadratic complexity, leading to a higher number of messages exchanged between nodes.

d) Disk Usage: Regarding disk I/O, we only display measurements for disk writes because, since the keyspace fits entirely into memory, disk reads are negligible. Figure 20 helps explain the poor performance of Cockroach in write heavy workloads when compared to SconeKV and Cassandra. It is important to note that these values, of KB/s written to disk, were captured while the system failed to achieve 1000 operations per second. In the case of SconeKV, it only writes to disk for durability in case of catastrophic failures, and it uses RocksDB which is highly optimized. These writes are based on a configurable period (10sec for these experiments), and thus the impact is negligible. For read heavy workloads (Figure 21) Cockroach still writes much more to disk than the alternatives, but an order of magnitude less than in the

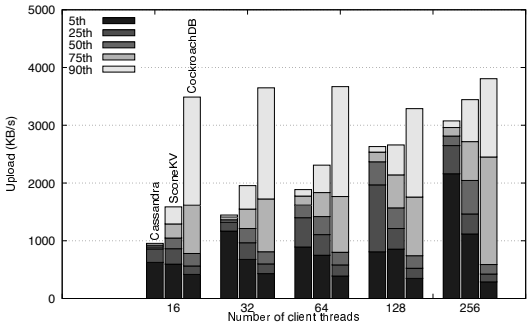

Fig. 16: Bandwidth usage during update heavy Workload A, upload.

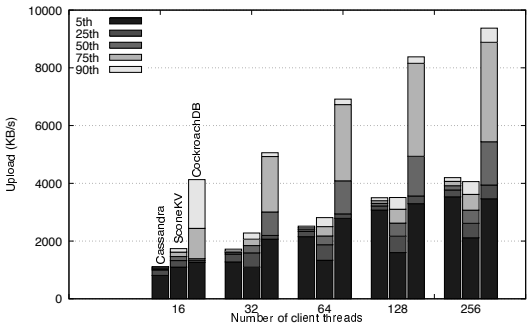

Fig. 17: Bandwidth usage during read-only Workload C, upload.

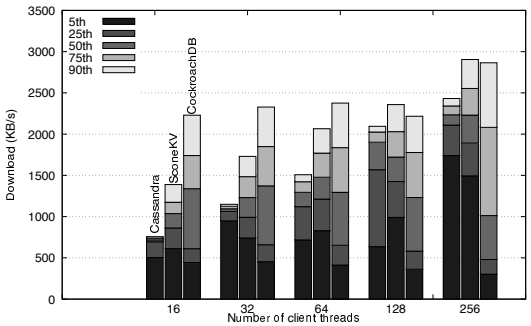

Fig. 18: Bandwidth usage during update heavy Workload A, download.

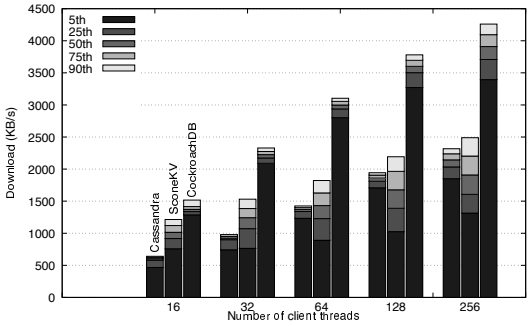

Fig. 19: Bandwidth usage during read-only Workload C, download.

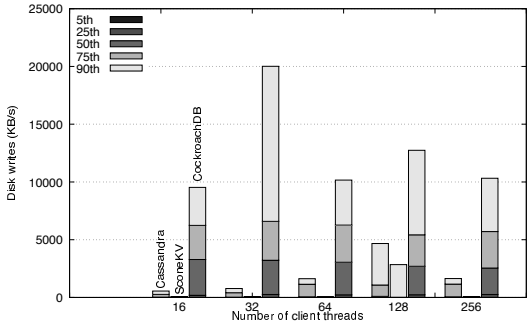

Fig. 20: Disk write usage during update heavy Workload A.

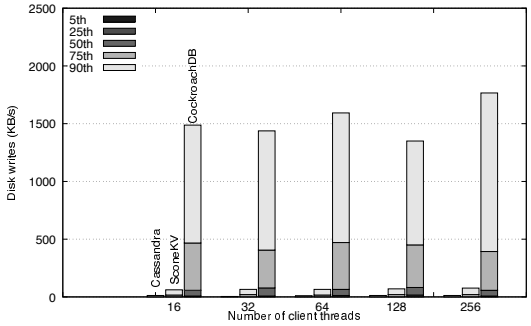

Fig. 21: Disk write usage during read-only Workload C.

update heavy workload and provides much more throughput. The high variance of the measurements of Cockroach can be explained with the distribution of requests. This leads some ranges (hot-spots) to have more entries in the log and thus the nodes in charge of those ranges have more data to persist to disk.

VI. Conclusion

This work describes SconeKV: a scalable, strongly consistent key-value store with support for distributed transactions. The system employs horizontal partitioning and an agreement protocol based on 2PC to provide serializable, multi key, distributed transactions at scale. It guarantees the replication factor using state machine replication, providing consistency and fault tolerance. It leverages a scalable membership layer with strong probabilistic consistency guarantees - PRIME. The experimental results show that SconeKV performs better than CockroachDB in write heavy workloads whilst being competitive with Cassandra in all workloads. This opens a new space in the spectrum of solutions with promising results: SconeKV scales whilst providing serializable distributed transactions.

References

[1] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[2] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2001, pp. 329–350.

[3] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, *A scalable content-addressable network*. ACM, 2001, vol. 31, no. 4.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *ACM SIGOPS operating systems review*, vol. 41, no. 6. ACM, 2007, pp. 205–220.

[5] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

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

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

[8] S. Voulgaris, D. Gavidia, and M. Van Steen, "Cyclon: Inexpensive membership management for unstructured p2p overlays," *Journal of Network and systems Management*, vol. 13, no. 2, pp. 197–217, 2005.

[9] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié, "Scamp: Peer-to-peer lightweight membership service for large-scale group communication," in *International Workshop on Networked Group Communication*. Springer, 2001, pp. 44–55.

[10] J. Leitão, J. Pereira, and L. Rodrigues, "Hyparview: A membership protocol for reliable gossip-based broadcast," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. IEEE, 2007, pp. 419–429.

[11] A. Das, I. Gupta, and A. Motivala, "Swim: Scalable weakly-consistent infection-style process group membership protocol," in *Proceedings International Conference on Dependable Systems and Networks*. IEEE, 2002, pp. 303–312.

[12] A. Dadgar, J. Phillips, and J. Currey, "Lifeguard: Local health awareness for more accurate failure detection," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2018, pp. 22–25.

[13] C. Conrad. Norbert. [Online]. Available: https://github.com/rhavyn/norbert

[14] L. Suresh, D. Malkhi, P. Gopalan, I. P. Carreiro, and Z. Lokhandwala, "Stable and consistent membership at scale with rapid," in *2018 USENIX Annual Technical Conference USENIX ATC 18*. ACM, 2018, pp. 387–400.

[15] F. Santos, "Prime : Probabilistic membership – large scale membership and consistency," 2018.

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

[17] L. Lamport *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[18] 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, pp. 8–17.

[19] B. Liskov and J. Cowling, "Viewstamped replication revisited," 2012.

[20] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, 2014, pp. 305–319.

[21] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, pp. 1–22, 2013.

[22] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998.

[23] 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*. ACM, 2010, pp. 143–154.