SconeKV: Strongly CONsistEnt Key-Value Store

João Tiago Alves Gonçalves

Thesis to obtain the Master of Science Degree in

Information Systems and Software Engineering

Supervisors: Prof. Miguel Ângelo Marques de Matos
Prof. Rodrigo Seromenho Miragaia Rodrigues

Examination Committee

Chairperson: Prof. João António Madeiras Pereira
Supervisor: Prof. Miguel Ângelo Marques de Matos
Member of the Committee: Prof. Sérgio Marco Duarte

November 2020

For my parents, and for Mel,

Acknowledgments

First of all, I would like to express my special gratitude to Professor Miguel Matos and Professor Rodrigo Rodrigues, for welcoming me as a student, for being my advisors, for supporting and guiding my work whilst providing great encouragement. Without their dedication, this thesis would not have been possible.

Thanks to all my colleagues and friends at IST. Especially David and Rodrigo for their support and comradery in this portion of my academic life.

A special thanks to my parents for their endless support in the pursuit of my objectives. Also my dog Mel. For the calmness but also the excitement she provides to the household, in this era of social isolation.

Lastly, a sincere thank you to Mafalda for being as extraordinary as she is.

Resumo

AAs bases de dados relacionais fornecem uma base sólida para a construção de aplicações devido às suas propriedades ACID, contudo, incorrendo em custos de sincronização. As aplicações distribuídas modernas atingiram tal escala, tanto em termos de quantidade de dados quanto de número de clientes simultâneos, que as bases de dados tradicionais não conseguem sustentá-los, resultando na degradação do seu desempenho. Outras soluções de armazenamento distribuído surgiram, horizontalmente escaláveis, mas empregando protocolos de replicação otimistas que garantem apenas consistência eventual. Além de oferecer uma consistência mais fraca, muitas bases de dados chave-valor modernas têm dificuldade em manter suas garantias em larga escala, com maior latência e maior dinamismo na sua filiação. Esses sistemas oferecem melhor desempenho, mas fornecem uma base menos estável para construir aplicações, permitindo atualizações concorrentes de dados cujos conflitos precisam de ser resolvidos pelos criadores de aplicações. Nesta tese, fornecemos uma visão geral do estado da arte, discutimos as principais limitações das soluções atuais e propomos o SconeKV: uma base de dados de chave-valor distribuída com garantias de consistência fortes mesmo em larga escala. O SconeKV oferece transações distribuídas serializáveis. Aproveita uma camada de filiação com garantias fortes e probabilísticas e um desenho baseado em particionamento horizontal, reduzindo a sincronização necessária, ao mesmo tempo que emprega protocolos de replicação consistentes, fornecendo, consistência forte aos clientes.

Os resultados experimentais mostram que o SconeKV tem uma performance bastante superior à do CockroachDB em escritas, ao mesmo tempo em que é competitivo com o Cassandra em todas as cargas de trabalho.

Palavras-chave: sistemas distribuídos, bases de dados, consistência, escalabilidade

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 provide an overview of the state-of-the-art, discuss the main limitations of current solutions, and 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.

Keywords: distributed systems, storage, consistency, scalability

Contents

List of Tables

List of Figures

Nomenclature

Greek symbols

α	Angle of attack.
β	Angle of side-slip.
κ	Thermal conductivity coefficient.
μ	Molecular viscosity coefficient.
ρ	Density.

Roman symbols

C_D	Coefficient of drag.
C_L	Coefficient of lift.
C_M	Coefficient of moment.
p	Pressure.
$\mathbf{u}$	Velocity vector.
u, v, w	Velocity Cartesian components.

Subscripts

∞	Free-stream condition.
i, j, k	Computational indexes.
n	Normal component.
x, y, z	Cartesian components.
ref	Reference condition.

Superscripts

*	Adjoint.
T	Transpose.

Chapter 1

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 provided 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 were able to scale vertically, but now systems require databases that scale horizontally, are also 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 [SMK+01, RD01, RFH+01] 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 locate objects in $O(\log n)$ hops. Modern key-value stores, namely, Dynamo [DHJ+07] and Cassandra [LM10], 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 [Apa] reporting difficulties in maintaining the consistency guarantees during membership changes. These shortcomings are emphasized with the dynamism of large scale deployments.

Today, applications once again require strong consistency on 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. In order to achieve strong consistency at a high scale, systems need to be designed in such a way that reduces the synchronization requirements. Google Spanner [CDE+13] is a highly scalable SQL database that shards data across many Paxos [Lam98, L+01] state machines, relying on GPS and atomic clocks to order transactions. It provides strong consistency at scale but requires specialized hardware to do so.

1.1 Objectives

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:

- **Maintain a total view of the system with strong consistency guarantees** — indispensable in order to guarantee the remaining properties, all nodes need to have knowledge of and be able to contact every other node in the system.
- **Guarantee strong consistency on all operations** — operations should be consistent across any number of concurrent clients.
- **Provide multi-object distributed transactions** — offer the ability of grouping operations in a single transaction performed atomically without any restrictions, the same transaction can contain operations performed on distinct keys, managed by distinct nodes in the system.
- **Guarantee the replication factor** — to provide availability, operations need to be consistently replicated according to the client's requirements.
- **Tolerate membership changes** — maintain the offered properties even in the case of failures, addition or removal of nodes from the system.

In other to accomplish these objectives, 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 [LM10], a highly scalable, eventually consistent distributed key-value store; and CockroachDB [Lab], 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 while providing serializable distributed transactions.

1.2 Contributions

This thesis makes the following contributions:

- Design of a strongly consistent storage solution that explores a new space in the spectrum of solutions, providing serializable transactions at a large scale.
- Implementation of a real prototype according to the design, containerized and deployed in real-world conditions.
- Evaluation and comparison with state-of-the-art storage solutions using realistic benchmarks.

1.3 Thesis Outline

The rest of this document is organized as follows. Chapter 2 provides a bottom-up view of a distributed storage system and its components. Chapter 3 describes the design of SconeKV and Chapter 4 details the implementation of the prototype. Chapter 5 presents the experimental evaluation of SconeKV, comparing it with state-of-the-art distributed key-value stores and discussing the results. Finally, Chapter 6 concludes this document by summarizing the main contributions and discussing possible enhancements of SconeKV in future work.

Chapter 2

Background and Related Work

As described in Chapter 1, the objective of our work is to develop a large scale storage system. In this chapter, we will present a bottom-up view of distributed storage systems, starting by describing the basics of gossip-based protocols, and how they offer good properties in terms of scalability for information dissemination in distributed systems (Section 2.1). Next, we discuss group communication, and how gossip concepts can be leveraged to achieve probabilistically reliable multicast whilst maintaining scalability and even ordering guarantees (Section 2.2). Following that, we explain the importance of group membership in distributed applications, distinguishing between partial and total views, comparing centralized and distributed membership management and also weak and strong guarantees when relating to distributed membership protocols (Section 2.3). Next, we discuss the topic of agreement and its role on constructing a coherent system even in the presence of failures, analyzing a couple different approaches (Section 2.4). Following that, we present some background on data consistency and how different levels can increase or decrease the concurrency of a system in favor of stronger or weaker guarantees (Section 2.5). Finally, we explore some distributed storage solutions and how they employ some of the previously explained concepts to achieve scalability whilst maintaining good performance (Section 2.6).

2.1 Gossip

The concept of gossip information dissemination emulates how gossip spreads between people [EGKM04]. Gossip dissemination is also sometimes referred to as epidemic dissemination, as the mathematical models it relies upon resemble those used to explain the spread of diseases. In a nutshell, gossip works as follows. When a node needs to disseminate a message it sends the information to a random set of peer processes of size k (fanout), which act accordingly, forwarding the message themselves to other nodes, and so forth. This presents a good solution for spreading information across a system, given it grows logarithmic with the number of nodes and is quite resistant to losses due to the redundancy in message transmission.

Gossip-based protocols can be classified into two classes, depending on how and when they disseminate a message, as described by Rivière and Voulgaris [RV11]. Rumor-mongering protocols disseminate information for a defined amount of cycles, the higher the number, the better the likelihood of all participants receiving the information. This class of protocols follows a Push-gossip approach, meaning that the communication is initiated by the sender. Anti-entropy protocols have nodes periodically choose a peer to gossip with, requesting a state exchange to determine the differences in messages received and resolve said differences. Anti-entropy protocols follow a Pull-gossip approach: when a node becomes aware of a message it did not receive, it explicitly requests that message from the peer.

2.2 Group Communication

Group communication, as described in [Kol04], addresses the problem of distributing information associated with multiple processes (senders and receivers). Group communication is mainly concerned with reliability, scalability and message ordering. Reliability is the capacity of a system to guarantee that events reach all destinations, such that it is able to tolerate faults. Scalability is the capability of a system growing in size with a cost proportional to the growth. Finally, the ordering of messages can range from none to causal or even total [CS93].

Group communication protocols can be divided into three main groups: reliable, unreliable and probabilistic. Reliable protocols guarantee that if a multicast reaches a destination then it will eventually reach all destinations. Regardless of the ordering guarantees offered, the synchronization needed to achieve this reliability limits scalability and even leads to unstable behavior when the system is under stress [CS93]. Unreliable protocols disseminate information on a best-effort basis, meaning if a participant discovers a failure an attempt is made to solve it but no guarantees are offered. This approach scales well and can even be reliable in ideal circumstances, but has an unpredictable behavior under perturbed conditions. Finally, probabilistic protocols guarantee that if a correct process delivers an event e, then with high probability all other correct group members will eventually also deliver event e. These approaches are typically based on gossip, avoiding the creation of hot-spots (typical of the hierarchical dissemination used in reliable protocols).

We now discuss probabilistic protocols in more detail. Bimodal Multicast [BHO$^+$99], also referred to as *pbcast*, is composed of two layers: an unreliable, hierarchical broadcast and an anti-entropy reconciliation protocol. The first layer delivers events on a best-effort basis, leveraging IP multicast or pseudo-randomly generated spanning trees to efficiently reach all destinations. The second layer detects message losses by randomly selecting members to gossip with, exchanging message digests and soliciting copies in order to converge to identical message histories. However, *pbcast* relies on a global view of the membership which is costly to maintain at scale.

Lightweight Probabilistic Broadcast [EGH$^+$03], *lpbcast* for short, is a completely decentralized information dissemination solution based on gossip and partial views. Nodes engage on unsynchronized periodical gossips, exchanging messages with both system events and membership updates. Each message contains system notifications received since the last gossip message, a digest of past notifications, a set of nodes that left the group and a set of nodes belonging to the group. As notifications are only gossiped once, the digest of past notifications enables the receiver to compare it with its own, possibly detecting a missing message and requesting the retransmission. Upon receiving a message, node p: (1) removes any node that left the group from its partial view; (2) randomly combines its current partial view with the received set of nodes, maintaining a view with size l; (3) delivers the received notification to the application.

Probabilistic Atomic Broadcast [FP02] (PABCast for short) disseminates messages through a sequence of asynchronous rounds (exemplified in Figure 2.1). In each round, node p can either broadcast a message or vote on another message (broadcasting a message is, in fact, adding a message to the list and voting on it). Each node p keeps a list of votes per round $list_p$ and periodically chooses a random subset of nodes to send it to. Each node q that receives the message updates the list: (a) if q already voted this round, it updates both lists so that they converge; (b) if q has not yet voted this round, it selects the message with lower count and casts its vote on it. Rounds are terminated by each node independently once they have collected $n - f$ votes for that round (f being the maximum number of processes that may fail).

EpTO [MMF$^+$15] is a total order epidemic dissemination algorithm with probabilistic reliabil-

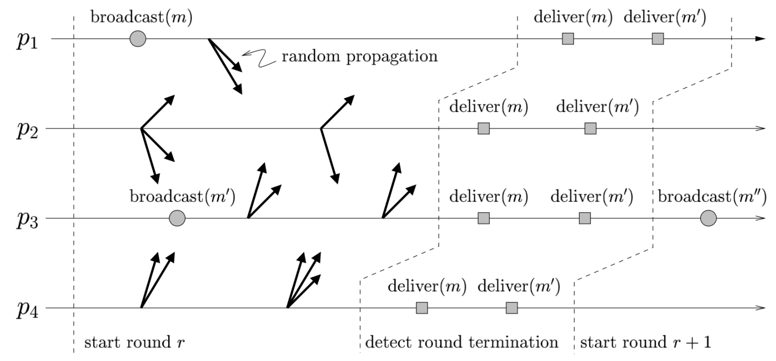

Figure 2.1: **PABCast** An example execution of the algorithm. Processes $p1$ and $p3$ broadcast m and m' respectively and both messages are delivered by all processes during round r. Process $p3$ can only broadcast a second message m'' in a following round $r+1$. This figure was obtained from [FP02].

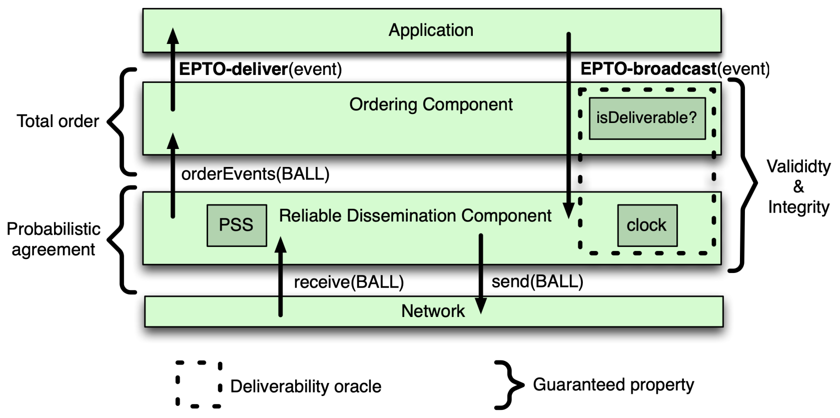

Figure 2.2: **EpTO** Architecture with component interactions. This figure was obtained from [MMF+15].

ity guarantees, meaning that processes eventually agree on a set of received events with high probability and deliver them to the application layer in total order. It follows the balls-and-bins model [Kol04] regarding information dissemination, mapping processes to bins and events to balls. It is divided into a dissemination and an ordering component (depicted in Figure 2.2). The dissemination component is responsible for delivering events to other nodes. If a node wants to broadcast an event, the event is timestamped, ttl set to 0 and added to *nextBall*, where ttl is the number of rounds an event has been gossiped and *nextBall* is a collection of the events currently being disseminated by a node. When a node receives a ball, events with $ttl < TTL$ are added to *nextBall*, where TTL is the maximum number of rounds an event is gossiped. Every round, node p increments the ttl of the events in *nextBall* and send it to K other processes. The ordering component is responsible for delivering the events to the application, satisfying the total order property. Every round it updates the list of received events with those in *nextBall*, increments their ttl and delivers the events that are ready ($ttl \geq TTL$).

2.3 Membership

Most distributed applications require the notion of a view, a list of correct nodes belonging to the system. Depending on the application's needs, the view can either be total (a list with all other nodes, that grows with the size of the system) or partial (a subset, with certain properties, of all correct nodes). Membership protocols provide processes belonging to the group with a list of non-faulty members, whilst also ensuring that the list is updated with new members joining or leaving (either by choice or due to faults).

2.3.1 Partial Views

Some applications work with each node only knowing a subset of other nodes, called neighbors. In peer-to-peer systems, nodes jointly maintain what is called an overlay network, a logical network imposed on top of the physical network, used to disseminate information. In terms of how the overlay structure is managed, systems can be divided into two groups: structured and unstructured overlays.

Structured Overlays

Structured overlays impose a specific linkage between nodes. This can be done according to physical factors (latency, distance) to improve the routing of messages, but can also rely on system logic. Distributed hash tables (explained in further detail in Section 2.6) map items and nodes onto an identifier space and construct a structured overlay using routing tables. Content-Addressable Network [RFH$^+$01] maps items to a coordinate inside a d-dimensional Cartesian coordinate space. Nodes are responsible for hyper-rectangles called zones and keep routing tables containing their direct neighbors inside said space (Figure 2.3). Chord [SMK$^+$01] uses a one-dimensional identifier key space (where the largest identifier is followed by 0, wrapping around as in a ring) mapping nodes and items to that identifier space. Each node p maintains a so-called finger-table with $\log N$ entries, where the i^{th} entry corresponds to a node that succeeds p by at least 2^{i-1} in the identifier circle (Figure 2.4). Pastry [RD01] arranges items and nodes in a circle, as in Chord, with a 2^b identifier space while using a prefix-based forwarding scheme. Nodes construct routing tables (example in Table 2.1) with $\log_{2^b} N$ rows, each with $n = 2^{b-1}$ entries. For each node p, row i has n peers which share the first i identifier digits with p.

In summary, structured overlays offer efficient routing of messages, however, they are highly susceptible to failures and churn as the overlay needs to be rebuilt when those occur [RGR$^+$04].

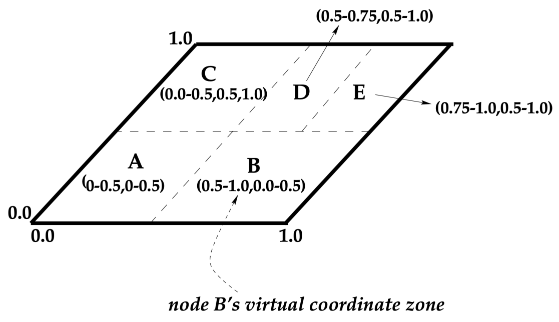

Figure 2.3: **CAN** Example of a 2-dimensional coordinate overlay with 5 nodes. The coordinate space wraps (not illustrated here). This figure was obtained from [RFH+01].

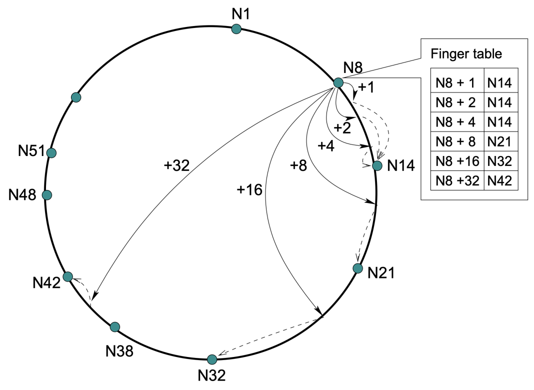

Figure 2.4: **Chord** Finger-table entries for node 8. This figure was obtained from [SMK+01].

Shared prefix	0	1	2	3
—	01230	13320	22222	—
3	**3**0331	**3**1230	—	**3**3123
32	**32**012	—	**32**212	**32**301
321	—	**321**10	**321**21	**321**31
3210	**3210**0	—	**3210**2	**3210**3

Table 2.1: **Pastry** Example of a routing table for node 32101 in a 2^3 identifier space. Some entries are omitted as they match the node identifier.

Unstructured Overlays

Unstructured overlays do not impose a specific linkage between nodes, links are created randomly. To ensure the connectivity of the overlay, links need to be established with some redundancy. As there is no explicit criteria to form links, with enough redundancy, unstructured overlays are less susceptible to failures and churn as there is no need to rebuild the overlay, making them a good solution for information dissemination in highly dynamic environments. Typically, these are constructed with every node maintaining a small set of neighbors with a fixed size or one that grows logarithmically with the size of the system. To achieve this, systems employ so-called Peer Sampling Services [JVG+07], which are fully decentralized solutions for building unstructured overlays. These services can either be proactive or reactive.

In a proactive PSS (also referred to as cyclic PSS) processes exchange views with their peers periodically regardless of failures occurring or not. CYCLON [VGVS05] introduces the concept of age, the number of cycles since a specific entry was added to a node's view. Nodes shuffle neighbors with each other each cycle. Each time a node p shuffles with a node q, p selects a random set of its neighbors and drops its oldest entry, sending the set to q along with an entry pointing to p with age 0. This way correct nodes will introduce new references to themselves each cycle, remaining in other nodes' views. A failed node will not shuffle with others, meaning it will not create new entries with age 0 pointing to itself and eventually all references to it will be dropped by the other nodes. Thus, the protocol reduces the amount of time a failed node remains in views when compared to a completely random solution.

With a reactive PSS, processes views are only modified when failures occur or new processes join the system. SCAMP [GKM01] updates partial views using a subscription protocol, triggered when new nodes join the system. With nodes leaving the system (by unsubscribing or simply failing) others might become isolated. Nodes periodically send heartbeats to all other nodes on their partial view, and if one considers itself isolated it joins the system again (by subscribing).

HyParView [LPR07] combines both approaches, maintaining a large passive view and a smaller active view. The passive view is not used for message dissemination and it is updated by periodically shuffling with other peers. The active view, as the name implies, is the one used to send messages and is updated when a failure is detected, removing the failed node from the active view and promoting an entry from the passive view.

In summary, the two approaches present a clear trade-off between view freshness and network bandwidth usage.

2.3.2 Total Views

Some applications require each node to know every other node in the system. The most obvious solution to tackle the problem of managing a membership group would be to centralize it. Electing a leader, for example, works well for a few nodes but creates an obvious bottleneck and compromises availability. A better approach is to centralize the membership management logically but not physically, using a replicated coordination service. Norbert [Con] leverages Apache ZooKeeper [HKJR10], a service for coordination of distributed processes, to achieve just this. ZooKeeper provides the abstraction of data nodes (called znodes) organized in a hierarchical name space, forming a tree-like structure, presenting application developers with an API that enables them to create their own specific coordination mechanisms. For membership coordination, the group is represented by a znode z_g with a well-known location, and group members create ephemeral znodes under z_g to join the membership. Ephemeral znodes have the special property that, while they are still created and deleted as regular znodes, they are automatically removed by the system if the session that created them terminates. The fact that the znodes used for joining the group are ephemeral ensures that, if a process fails, it will eventually leave the group. Members can receive changes by registering for updates on the children of z_g In this specific case the proposed solution does not scale, due to ZooKeeper's implementation. This is because all ZooKeeper updates are forwarded to the leader which then atomically broadcasts the change using Zab [RJ08]. When an update in the membership occurs, ZooKeeper replicas inform the observers that their information about the group is outdated, requiring each group member to re-register after every callback to continue to receive updates. This presents two problems. First it means that N requests are sent to ZooKeeper after every JOIN and LEAVE to re-register for updates, limiting the scalability of the solution. Secondly, as ZooKeeper only informs members that their membership is outdated with the first change, following changes will not be transmitted to group members until they update their membership information and re-register for updates, meaning it can lead to inconsistent views. Generally, centralizing the membership management works well for retrieving group information as, depending on configuration, replicas can respond to requests locally. Nevertheless, when an update occurs replicas still need to synchronize with each other, achieving some sort of consensus which is well known to scale poorly [KJ10].

Distributed membership can scale but have the problem of maintaining view consistency. Views are considered consistent if all members have the same view and apply the same changes to it over time. Distributed solutions can generally offer scalability at the cost of view consistency or base performance and resource usage, depending on the proposed guarantees, which we discuss next.

Weak Guarantees

SWIM [DGM02] is a protocol that leverages gossip techniques for information dissemination to offer scalability and good performance with weaker guarantees in terms of view consistency. Every cycle, each node p selects a random node q from its member list to `ping`. A successful `ping` results in q answering `ack` to p. If q does not respond within a defined timeout, p initiates an indirect probe (`ping-req`), contacting k other members to `ping` q on p's behalf. If no response, either directly or indirectly, is received, p marks q as suspected and disseminates it. In the future, if p or another member successfully `ping` q, or even if q learns about the suspicion, an alive message can be disseminated, clearing previous suspicions. Nevertheless, if after a timeout p does not receive a message revoking the suspicion, p removes q form its local membership and disseminates an update stating q as faulty. Updates to the membership are piggybacked on to failure detection messages (`ping`, `ping-req`, `ack`), effectively propagating through gossip. Each message is piggybacked at most $\lambda \times \log(n)$ times, where n is the size of the system and λ is a configurable parameter.

Memberlist [DPC18] is an extension of SWIM used by HashiCorp on Serf [Has]. The most notable differences are:

- In addition to the gossip messages piggybacked on the failure detection messages, it also sends regular gossip messages, allowing the gossip rate to be tuned independently of the failure detection rate, possibly leading to faster convergence.
- To improve the consistency of the views, nodes will periodically do a full state sync over TCP.
- Lamport clocks [Lam78] are attached to all messages, allowing the ordering of messages delivered out of order. Most importantly, they are used in case of out of order JOIN and LEAVE messages regarding the same node p. Before, if some node q received LEAVE before JOIN, it would add p to its view, generating an incorrect state.
- Local Health Aware Probe, a node will adjust its probe and timeout periods if it suspects about its own health. A node p can suspect about itself by comparing the number of received acknowledgments (positive ack or negative $nack$) with the number of `ping` and `ping-req` messages sent, as well as needing to refute a suspicion about itself. With this mechanism, if a node considers itself slow it will take more time to suspect of others, reducing the number of false positives.
- Local Health Aware Suspicion, the suspicion timeout for each node p starts higher than normal, but it is reduced proportionately to the number of nodes that suspect of p. This mechanism allows for higher tolerance of temporary slow responses.

Strong Guarantees

On the other end of the spectrum, we have systems that strive for consistency. Rapid [SMG+18] combines gossip-based dissemination with a fast-path consensus agreement to achieve that. In Rapid, nodes are organized using an expander graph, forming K pseudo-random rings. With this approach, each node monitors K nodes (subjects) and is monitored by K others (observers). $REMOVE$ and $JOIN$ are alerts that an observer disseminates when a subject is to leave or join the membership, respectively. These alerts are irrevocable, meaning there are no conflicting reports from the same source about a specific node. As each node is observed by K others, multiple alerts are generated about the same subject. Monitoring incurs in $O(K)$ overhead per cycle, while nodes joining/leaving lead to $2K$ edge changes. These alerts are handled at every node independently. For each observer/subject pair, each node p maintains $M(o,s)$ that represents whether an alert from o about s was received. Node p decides on node s by using $tally(s) = sum(M(*,s))$, deferring on a decision until the report is considered stable. A report is to be considered unstable if $tally(s)$ is above a low watermark L (threshold to take noise into account) but below a high watermark H. Node p delays proposing a configuration change until it has at least one node in stable report mode ($tally(s) >= H$) and no node in unstable report mode. A view-change occurs using Fast-Paxos [Lam06], a bitmap of votes is generated and disseminated for each unique proposal. It reaches a decision if a quorum of $3/4$ of the membership is formed. If there is no fast agreement, the view-change process continues, achieving consensus through classical Paxos [Lam98, L^{+}01] (described in greater detail in Section 2.4).

PRIME [San18] is a membership service that leverages EpTO (described in Section 2.2) to achieve probabilistic consistency in views. PRIME consists of three main components: the Membership Manager maintains the view of the system, the Hole Manager handles possible missing view updates at each node (caused by EpTO's probabilistic agreement) and a Failure Detector that handles detection and decision on failed nodes. The membership is updated when a new node p joins the group by sending a ⟨DISCOVERY⟩ message to any node q already belonging to the system. Node q will then broadcast ⟨JOIN | p⟩ to the rest of the group, each node will update their view accordingly and verify if they are p's first-guardian (node that monitors p and has the lowest identifier, further explain bellow), and send p a view copy. Upon receiving a view copy, p initializes the Failure Detector and starts acting like a normal node. The implementation of EpTO used in Prime may deliver out-of-order events instead of dropping them. The only problematic case is a ⟨JOIN⟩ after ⟨LEAVE⟩ for the same node. This is solved by tracking out-of-order ⟨JOIN⟩ messages and dropping them if they correspond to an already delivered ⟨LEAVE⟩. As stated earlier, EpTO's probabilistic agreement may fail to deliver certain updates, generating holes in a node's view. To handle this, each update has also the hash of the current view and a version (timestamp and unique node identifier). To fix a hole, a node will decreasingly go from the timestamp of the update to zero, asking another node for missing updates. Failure detection is accomplished by organizing the nodes in a ring, sorted according to the unique node identifiers. Each node p monitors K nodes (Protegees) and is monitored by K others (Guardians). Whenever a guardian's Failure Detector considers node p as failed, it sends ⟨SUSPICION | p⟩ messages to the other guardians until the suspicion is refuted or more than half of the guardians agree. A suspicion is refuted if a guardian sends an $\langle \textsc{Alive} \mid p \rangle$ message to the other guardians, in the case it detected that p had not failed.

2.4 Consensus

Agreement, also referred to as consensus, is one of the most fundamental problems of distributed computing. Achieving it is key, perhaps even indispensable, when building reliable, consistent, fault-tolerant distributed systems. Consensus algorithms aim to solve this problem, allowing a set of processes to reach an agreement whilst tolerating the failures of some members. Paxos [L+01] is a well-known solution to this problem and many others derive from it.

The protocol is defined for a single decision but can be extended, combining multiple instances of the protocol, to achieve a series of decisions - multi-Paxos. The algorithm ensures that a single value is chosen from a collection of proposed values. It is performed by three different agents: proposers, acceptors and learners (although in practice a single process can act as more than one agent). In the first phase of the protocol, a proposer wishes to propose a value v. It creates a unique identifier n and sends $\langle \textsc{Prepare} \mid n \rangle$ to at least a majority of acceptors. When an acceptor receives a $\langle \textsc{Prepare} \mid n \rangle$ message it replies with $\langle \textsc{Promise} \rangle$ if and only if n is the greatest identifier it has received. If a proposer receives $\langle \textsc{Promise} \rangle$ message from a majority of acceptors, then it starts the second phase of the protocol, sending $\langle \textsc{Accept} \mid n, v \rangle$ to all correct acceptors. Upon receiving $\langle \textsc{Accept} \mid n, v \rangle$, an acceptor accepts the proposal (unless it has already promised a number greater than n), sending $\langle \textsc{Accepted} \mid n, v \rangle$ to the proposer and all learners (which effectuate the decision). This simplified solution omits some fault tolerance mechanisms and possible optimizations, in sum it solves the problem of reaching an agreement, selecting a single value consistently across a distributed set of processes. As is, this solution does not scale, mostly due to the number of phases and the message complexity of each one. The consensus problem has other, more concrete, approaches that with a few assumptions and different applications can achieve better message complexity, and thus better scalability.

2.4.1 State Machine Replication

A state machine consists of set variables and commands, that encode and transform its state respectively [Sch90]. Commands are executed atomically and have a deterministic output. A distributed system behaves as a state machine if all node agree on each action taken, performing each action consistently across all nodes, if and only if that action was agreed on before.

State machine replication [Sch90], from here on referred to as SMR, offers a t fault tolerant state machine, implemented as a distributed log where each node acts as a replica of the state machine. This requires all non-faulty replicas to start in the same initial state and to receive and process the same sequence of commands, implying that every non-faulty replica has to receive every command - agreement - and process it in the same relative order.

Viewstamped replication [OL88, LC12], otherwise referred to as VR, is an replication protocol that tolerates failures. It ensures reliability and availability if no more than f nodes fail, assuming a crash-failure model, through the use of replica groups of $2f + 1$ nodes. The system works using the primary-backup model, requests (which comprise the SM commands) are routed through a primary replica, ensuring an order that is simply accepted by the other replicas (backups). Comparing it with Paxos, this design principle eliminates the need for the first phase of the protocol, with only one *proposer* there is no need for *promises*. This does unfortunately introduce a single point of failure. The solution is for backup nodes to monitor the primary node the trigger a succession protocol in case of a primary failure (an adaptation of which is presented in Section 3.6). The state of each replica is comprised of a set of variables, including, but not limited to, a *op-number* assigned to the most recent request, a *log* containing the requests received so far (in order) and a *commit-number* corresponding to the largest (most recent) *op-number* committed. Under normal operation, when the primary receives a request it advances the *op-number*, adds the request to the end of the *log* and sends $\langle \textsc{Prepare} \mid m, n, k \rangle$ to all backup nodes, m being the request, n the assigned *op-number* and k the *commit-number*. When a backup receives a $\langle \textsc{Prepare} \rangle$, it waits until it has entries in its log for all previous *op-numbers*, increments its *op-number*, adds the request to the end of its log and sends $\langle \textsc{PrepareOK} \mid n, i \rangle$ to the primary, with i being the backup identifier. Once the primary receives f $\langle \textsc{PrepareOK} \rangle$ from different backups, it considers n and all previous entries as committed. Backups are informed of the commit in a following $\langle \textsc{Prepare} \rangle$ messages (the k argument represents the latest committed entry). It then executes in order all operations in the log from the previous committed entry up to k.

Raft [OO14] aimed to reach a more understandable solution for the agreement problem, when compared to the widely known Paxos. It is similar to the previously described Viewstamped Replication, as it also only allows for log entries to flow from the master to the backups, simplifying the algorithm, and has backups monitor the current master, triggering a leader election if a failure is detected. It handles membership changes differently, allowing for the cluster to continue operating during changes in the configuration. This requires decisions to overlap majorities of two different configurations (the old and the new), forming a joint consensus. In comparison, VR does not accept client requests during a view change, meaning Raft provides better availability during membership changes, even if at the cost of extra coodination.

2.4.2 Sharding

Another workaround to achieve agreement at a higher scale, as presented in [GHSV19], is sharding, otherwise referred to as horizontal partitioning. This is an approach highly used in databases: the idea is to divide the system state into disjointed sections, each running a separate agreement protocol, at a smaller scale.

To achieve this, many distributed storage solutions rely on consistent hashing [KLL+97]. This consists on mapping both nodes and items to the same bounded space, typically represented as a ring, in such a way that items are not associated directly to a specific node, but rather to the node closest to it in the space. Hashing items directly to indexes makes the assignment dependent on the current size of the system, thus changes to the system composition require items to be re-hashed and re-assigned to buckets. Consistent hashing ensures that nodes joining and leaving the system will only affect nodes directly adjacent to them in the space. When using sharding, this is especially relevant if the system is design to adjust the number of shards according to the environment conditions.

Sharding alone is not sufficient to ensure that the whole system stays consistent, meaning additional protocols are required to guarantee cross-shard coordination. An example of such protocol is two-phase commit (2PC), commonly used to commit transactions than span across multiple shards, used for example in [CDE+13]. The protocol works using a group of participants, one of which is considered the coordinator, and, as the name implies, has two phases: voting phase and a decision phase. During the voting phase, the coordinator requests all participants to prepare to commit, collecting their responses (*yes* or *no* according to whether or not the can commit the transaction). In the decision phase, the coordinator informs the participants of the global decision: commit if all participants voted *yes* or abort if any participant voted *no*. After voting *yes*, a participant can neither commit nor abort until the coordinator informs them of the global decision, acting accordingly once it occurs. 2PC has weak liveness properties, it blocks in case of failures, leaving all other participants waiting indefinitely. This problem can be reduced using a membership system with strong consistency guarantees and replicating the state of each participant, ensuring that in case of a failure the system is repaired autonomously and the protocol resumes.

2.5 Consistency

Consistency is one of the key concepts of distributed systems. The idea is for a distributed system is to appear as a single unit, despite the distribution, whilst employing replication techniques that ensure it offers better availability and reliability than a centralized system. One of the problems behind the replication of system state is the need for it to stay consistent across replicas, especially in the case of data stores (which we will discuss further in Section 2.6). Consistency models, in the database world referred to as isolation levels, defines the agreement between the data store and its users with regards to the consistency it offers for accesses to its system state (the data stored by the client).

Consistency comes at a cost of the level of concurrency and throughput a system is able to provide. The possibility of providing differing levels of consistency for applications led to the creation of the ANSI/ISO SQL-92 specifications, later refined by [BBG$^+$95, ALO00]. These specifications were constructed based on anomalies (incorrect behaviours), where each model defines which anomalies it prevents. The strongest consistency possible is strict serializability: it requires all transactions to be performed atomically, totally ordered and respecting real-time. Meaning if a transaction T_A completes before T_B begins, then T_A should precede T_B in the serialization order. Serializability weakens those guarantees, removing the real-time requirement.

Snapshot Isolation [BBG$^+$95] was defined to provide strong consistency but allow more concurrency and less contention than serializability. Transactions operate in independent snapshots of the database, progressing without any contention and only making their changes visible at commit time, atomically. SI requires that if a transaction T_A has modified an object x, and another transaction T_B committed a write to x after T_A's snapshot began and before T_A has committed, then T_A must abort. In comparison to serializability, SI dos not enforce total order of transactions, allowing them to interleave with one another. This may lead to write skews: transactions T_A and T_B concurrently reading an overlapping data set (x and y), concurrently making disjoint updates (T_A writes to x whilst T_B writes to y), and finally concurrently committing, without either transaction having seen the update performed by the other.

Finally, some systems employ even weaker levels of consistency, benefiting from the increase performance they allow for. Eventual consistency [DHJ$^+$07, LM10] simply states that updates will eventually reach all replicas of a database, giving no guarantees regarding client observations. Causal consistency [ALR13] requires that causally dependent operations should appear in the same order to all processes, even if they disagree on the order of independent operations.

2.6 Distributed Storage

Distributed storage surfaced with the objective of providing applications with four main guarantees: availability, performance, durability and scalability. Centralized storage solutions have a clear single point of failure for both availability and durability reasons, whilst the hardware capabilities of a single system will cap the performance and the total amount of data it is able to store. The idea behind distributed storage is that dividing the data (replicated) amongst a set of processes increases the amount of data a system can store, gives better guarantees in terms of availability and durability (especially if the data is geo-replicated), and also distributes the work amongst multiple machines, improving performance.

Some of the first approaches to distributed storage were distributed hash tables (DHTs). CAN [RFH+01], Chord [SMK+01] and Pastry [RD01] are different solutions for the same problem: distributed storage system built on top of a structured overlay using a routing mechanism that ensures $O(\log n)$ hops for each query (as explained in Section 2.3). DHTs, in general, provide a similar programming model: objects (opaque to the system) that are indexed by a key and replicated for availability, organized in a flat namespace. In sum, DHTs present a solution for highly distributed storage but give very little guarantees in terms of consistency of the objects, as there are is no ordering of request nor conflicting resolution mechanism when concurrent updates on the same object take place.

Some systems leveraged the overlay and routing techniques employed by DHTs to offer better properties regarding object organization (hierarchical namespaces) and conflict resolution for concurrent updates. OceanStore [KBC+00] is a geo-distributed persistent storage system that employs a routing mechanism based on Tapestry [ZHS+04]. The two main goals behind OceanStore are as follow: it needs to be constructed on top of an untrusted infrastructure and data should be nomadic (meaning data will need to be available in different servers at different times). To achieve these goals, OceanStore employs data redundancy, cryptographic techniques (outside the scope of this work) and *promiscuous* caching (trade-off consistency for availability). The Tapestry routing mechanism employed in OceanStore works similarly to the DHTs described earlier. It is prefix-based as Pastry but it works in two phases. First, it will probabilistically attempt to find a replica of the object on a machine nearby, only if that first attempt fails will it enter the second phase, deterministically finding the object using a lookup mechanism (once again, similar to the one employed by Pastry). OceanStore objects can either be *active*, meaning modifiable, or *archived* meaning they are read-only. For modifiable objects, conflict resolution is done by ordering the updates. As OceanStore does not trust its infrastructure and strives to tolerate byzantine failures, this is not done by a single master machine. A *primary tier* of machines will use an agreement protocol to determine the order of the updates, whilst a *secondary tier* disseminates the updates using an epidemic protocol. Once the *primary tier* agrees on an order, this is multicast down to the other replicas and the update is applied.

2.6.1 Distributed Key-Value Stores

Distributed key-value stores provide a key to object abstraction, similar to DHTs, and work with a total view of the system, offering one-hop routing of requests. Dynamo [DHJ+07] is a highly available and scalable distributed key-value store with eventual consistency. It supports incremental scalability, symmetry (all nodes have the same responsibilities) and heterogeneity (work is distributed proportionately to the capability of each individual node). Developed by Amazon to be used in its e-commerce platform, Dynamo is highly focused on guaranteeing the Service Level Agreements related to response latency and reliability. As most key-value stores, it offers two main operations: `get(key)` locates object replicas corresponding to that key and returns an object (or a list of objects if there are conflicting versions), `put(key,context,object)` determines the responsible replicas based on th key and writes the object to disk. Dynamic partitioning of the data is ensured by consistent hashing, nodes are assigned to multiple points in the ring (relative to their capacity), and updates to objects are routed through a coordinator node (the one closest to the object in the ring) that replicates the object to its successors in the ring. Requests can reach any Dynamo node and, depending on the key, are forwarded to the corresponding coordinator which handles the request. Objects have versions (vector clocks) attributed when an update occurs to determine causality. As Dynamo aims for high write throughput, writes are never rejected and conflict resolution is pushed to the read operations. Most times a syntactic resolution occurs, meaning the system reconciled the updates by itself, but if there are conflicting versions of an object the client is required to collapse the updates (semantic reconciliation), for example merging the objects. Objects are replicated to N nodes and write and read operations require the participation of W and R nodes respectively. These parameters are configurable according to the workload necessities, possibly reducing the latency of the desired operation. If $W + R > N$ it yields a quorum-like system, achieved not between the first N nodes following the object in the ring, but between the first N correct nodes. As these groups may not be the same this is considered a *sloppy quorum*.

Cassandra [LM10] is an open-source adaption of Dynamo with some differences:

- Integrates the BigTable [CDG+08] data model, organizing items in tables with rows, columns, column families and super columns.

- Has three operations: `insert(table, key, columnName)`, `get(table, key, columnName)` and `delete(table, key, columnName)`.

- To account for heterogeneity, instead of nodes having multiple positions on the ring, Cassandra moves nodes along the ring to alleviate others.

- Replication is configurable, Rack Unaware works as in Dynamo (assigns to the first $N - 1$ successors) whilst Rack Aware and Datacenter Aware, as the names imply, choose successors based on their physical location.

Cassandra's authors also specify that their membership is managed using Scuttlebut [VRDGT08] and that failure detection is done according to the Φ-accrual failure detector [HDYK04]. Scuttlebut is a reconciliation protocol for anti-entropy gossip-based dissemination, used to maintain system membership as nodes exchange views. The Φ-accrual failure detector, in contrast with other failure detectors, does not emit a boolean value stating if a node has failed or not, rather emits a value Φ that represents the confidence a node has about another node having failed (the higher the Φ, the higher the confidence). Dynamo also employs a membership protocol and failure detection mechanism, but neither was specified by the authors in the paper.

2.6.2 Geo-Replication

Providing distributed storage in geo-replicated environments allows systems to provide client locality, reducing the latency for applications used throughout the world. Unfortunately, it does also introduce new challenges: the occurrence of faults is a given, latency is increased in communications and network partitions may occur, blocking the system depending on its design. As per the CAP theorem [Bre00], as system cannot, at the same time, guarantee consistency, availability and tolerance to network partitions. Some systems opted to weaken their consistency guarantees in order to achieve the desired performance and availability. ChainReaction [ALR13] employs chain-replication [VRS04] and offers *causal+* consistency on operations. Chain-replication organizes replicas in a chain topology, with writes being routed to the head and propagated down the chain until they reach the tail, which acknowledges the write to the client. Reads are routed to the tail, where writes are linearized and guaranteed to have been propagated to all other replicas, thus providing strong consistency. ChainReaction weakens these guarantees in order to leverage the rest of the chain in order to promote load balancing for read operations. *Causal+* consistency allows for more parallelism than linearizability, while still respecting causal dependencies between operations. In comparison with classic causal consistency, it guarantees that replicas may diverge due to concurrent updates but will eventually converge. Their approach relies on the fact that if some node x is causally consistent to some client observation, then all predecessors in the chain are also causally consistent. ChainReaction also reduces the constraint regarding writes reaching the tail before replying to the client, instead requiring them to reach some configurable node k, propagating from k to n asynchronously. As nodes are organized topologically, this k value does not need to represent a *quorum*, as the first k represent always the same nodes and not any k nodes. This allows the system to provide even greater load distribution of read operations, without negatively affecting the latency of write requests.

As discussed in Chapter 1, weaker consistency levels do not provide a strong enough foundation for many applications, delegating to developers the responsibility of dealing with their shortcomings and consistency anomalies. Some systems have appear offering strong consistency even at the geo-replicated level, most notably Google Spanner [CDE$^+$13], a scalable SQL database that shards data across many Paxos state machines. It is a temporal, multiversion database, data is stored in schematized semirelational tables and each version is automatically timestamped with its commit time, allowing applications to read data at old timestamps. Replication is configurable and can be dynamically controlled: applications can specify which datacenter holds which data, how it is replicated geographically and even dynamically move data between datacenters to ensure load balance. Most importantly, Spanner offers this whilst still guaranteeing externally consistent reads and writes and globally consistent read across the entire database for a specific timestamp. The is accomplished by assigning globally meaningful commit timestamps to all transactions, reflecting a serialization order and providing external consistency. These meaningful timestamps are obtained using the TrueTime API, which exposes clock uncertainty and guarantees timestamps depend on configurable bounds, slowing down to wait out the uncertainty if necessary. This is especially useful when paired with GPS and atomic clocks which keep uncertainty at less than 10ms.

CockroachDB [Lab] 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 transactional and strongly-consistent key-value store. It divides the keyspace into ranges and assigns ranges to nodes according to the replication factor selected, each range consistently replicated using Raft [OO14], with one node per range being the Raft leader for write requests. It replaces Google's use of TrueTime, atomic clocks and GPS, which requires dedicated hardware, with a software solution relying on hybrid logical clocks [KDM$^+$14] (HLC). 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 Cockroach still requires replicas to be within a configurable offset (default 500ms) to work correctly, shutting down nodes that get out of sync with 80% of the cluster.

2.7 Discussion

As discussed in Section 1, currently available solutions for distributed storage do not provide the necessary consistency guarantees in highly dynamic environments. DHTs construct a structured overlay where each node can only directly communicate with its neighbors and are unable to provide meaningful consistency guarantees, there are multiple routes for the same nodes and no ordering of requests nor conflict resolution for concurrent updates. Dynamo and Cassandra provide scalability with eventual consistency guarantees in a datacenter environment. Cassandra uses Scuttlebutt for managing its membership, which is known to misbehave when membership changes are too frequent. A Cassandra ticket [Apa] (open since 2015) states that, when joining the system, nodes can have overlapping coordination key ranges which in turn leads to inconsistencies (multiple coordinators for the same object).

Spanner, as a scalable SQL database that offers strong consistency at a geo-replicated level, somewhat contradicts the motivation presented in Chapter 1. This, however, comes with a major caveat: Google achieved this by leveraging their enormous and highly reliable network infrastructure whilst also relying on GPS and atomic clocks, reducing clock skew to the barest minimum and allowing them to lean on real time to order distributed transactions in an efficient and scalable manner. In the case of Cockroach, which mimics Spanner's design but without relying on dedicated hardware, performance degrades quickly under write heavy workloads, as we detail in Chapter 5.

To create a data store solution that maintains its consistency guarantees even in a geo-replicated deployment, we will need to employ a membership component with strong view consistency guarantees across nodes. Norbert provides logically centralized membership management with strong consistency in theory, but in practice, it does not scale to the number of nodes intended and the mechanism used for updates could lead to temporary inconsistencies. Rapid provides distributed consistent membership but consumes too many resources and does not maintain view consistency above 250 nodes [San18]. PRIME provides probabilistic consistency in views at the scale intended and thus satisfies one of the properties required for our system.

In the next chapter we will describe our proposal for a distributed large scale data store, and how a membership component providing PRIME's guarantees whilst maintaining scalability can be a foundation for a system that offers strong consistency on operations in large scale deployments with dynamic environments.

Chapter 3

SconeKV

In this chapter, we present SconeKV, a distributed key-value store designed to provide strong consistency guarantees at a large scale. The main objectives of SconeKV are as follows:

- **Maintain a total view of the system with strong consistency guarantees** — indispensable in order to guarantee the remaining properties, all nodes need to have knowledge of and be able to contact every other node in the system.
- **Guarantee strong consistency on all operations** — operations should be consistent across any number of concurrent clients.
- **Provide multi-object distributed transactions** — offer the ability of grouping operations in a single transaction performed atomically without any restrictions, the same transaction can contain operations performed on distinct keys, managed by distinct nodes in the system.
- **Guarantee the replication factor** — to provide availability, operation need to be consistently replicated according to the client's requirements.
- **Tolerate membership changes** — maintain the offered properties even in the case of failures, addition or removal of nodes from the system.

3.1 System Overview

SconeKV is a complete system constituted by four different layers, as depicted in Figure 3.1. The main purpose behind these layers is that each one provides the required properties to the layers above. Each layer represents a component with concrete implementation and guarantees, but the architecture is fully modular, allowing the use of other implementations in the future as long as they provide the same guarantees. The bottom layer is a Peer Sampling Service (discussed in Section 2.3.1), meaning it provides a list of nodes (neighbors) to the layers above that is updated over time. We use CYCLON, a proactive PSS. On top of the PSS we have a Total Order Group Communication component. As the name implies, it should guarantee the layers above that messages are disseminated to all nodes and delivered to the application in total order. We use an implementation of EpTO, a total order epidemic dissemination algorithm (described in Section 2.2). Above the group communication layer we have the Membership Manager that provides our application with strongly consistent views across all nodes in the form of a ring. As described in Section 2.7, PRIME is the only currently available solution that provides strong consistency guarantees at our intended scale, and with it we accomplish our first main objective for the system. The top layer is the main focus of our work: a Distributed Key-Value Store with strong consistency guarantees, fit for large scale deployments.

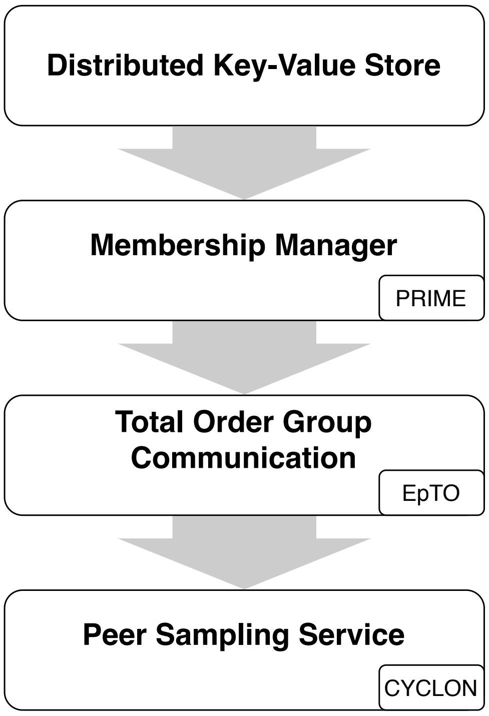

Figure 3.1: SconeKV architecture layer stack.

Focusing now on the top layer, SconeKV uses an identifier ring, using horizontal partitioning of the key space - sharding - to divide the ring into section called buckets. 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 leader election or another consensus variant in order to determine the master, it simply requires a deterministic function using the bucket members as input (for example, the node with the lowest identifier). Figure 3.2 presents the organization of the nodes in the cluster and the connections established between them.

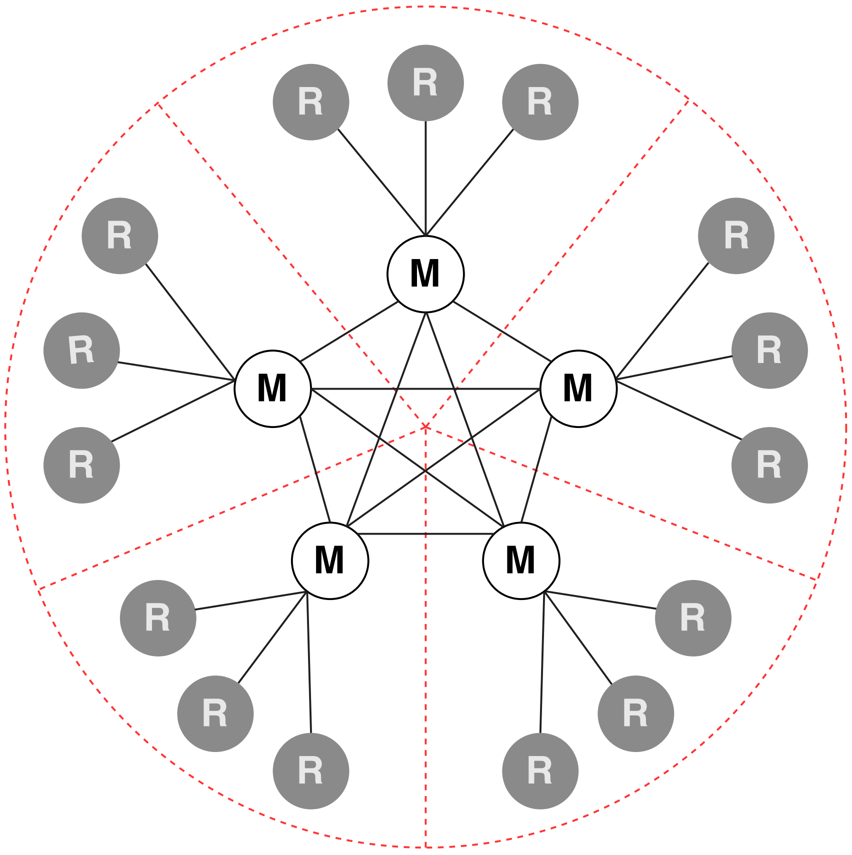

Figure 3.2: SconeKV node organization for a 20 node, 5 bucket deployment. This is the same topology used in the evaluation presented in Chapter 5. The red dotted lines divide the identifier space into buckets, M represents the bucker master, R represents a replica and the black lines represent the connections established between the nodes.

3.2 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 objects life cycle where a given operation should be performed. A key's version is initially 0 and incremented by one with each write operation to that key.

All operations must be performed inside a transaction. The SconeKV Client Library offers a `newTransaction` method that returns a new transaction. The client can then perform operations inside that transaction, while the library maintains the read/write set with the versions observed for each 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.

Algorithm 1 Node Initialization and Client Transaction Interface

```
upon event ⟨Init⟩ do
    $txs \leftarrow \varnothing$    ▷ map with all transactions, indexed by the transaction ID, containing rwSet, nodes involved, current transaction state and local decision responses
end event

upon event ⟨$master$, Write | $key$⟩ do
    $version \leftarrow$ GetVersion($key$)
    return $version$
end event

upon event ⟨$master$, Delete | $key$⟩ do
    $version \leftarrow$ GetVersion($key$)
    return $version$
end event

upon event ⟨$master$, Read | $key$⟩ do
    $value, version \leftarrow$ Get($key$)
    return $value, version$
end event

upon event ⟨$master$, CommitRequest | $txID, rwSet, nodes$⟩ do
    $tx[txID].rwSet \leftarrow rwSet$
    $tx[txID].nodes \leftarrow nodes$
    $tx[txID].state \leftarrow$ received
    trigger ⟨MakeLocalDecision|txID⟩
end event

```

SconeKV nodes do not maintain any transaction state during those transactions, as defined in Algorithm 1. The client triggers events on the master of the bucket responsible for the key, receiving the current version, and in the case of a read also the current value of the key. READ, WRITE and DELETE events can be performed on a replica instead of the master, depending on the client configuration. In Section 4.6 we explain this further, discussing the trade-off it presents.

To externalize the operations, the client issues `commit` for a given transaction. This operation will be successful or unsuccessful, depending if 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. Committing one or more requests and one response, depending on the buckets accessed in the transaction (explained in further detail in Section 3.4). The COMMITREQUEST event needs to be triggered the masters of the buckets involved on a transaction and not the replicas, in order to guarantee the serializability of transactions. The library also offers the option to `abort` a given transaction, which prevents it from ever being committed in the future.

The `read`, `write` and `delete` requests represent 1 RTT. This can be optimized in case of multiple operations in the same transaction accessing the same key. For example, if a `read(keyX)` is followed by a `write(keyX, valueY)`, the read operation is replaced by the write in the read/write set, maintaining the version observed earlier and omitting the additional request and corresponding RTT. A `commit` request, in the best case, represents 3 RTTs or 4 RTTs depending on whether it is single or multi bucket respectively. However, concurrency can lead to contention between transactions, thus affecting the latency of each request.

3.3 Consistency

In this section, we address our second objective of guaranteeing strong consistency on all operations. SconeKV was designed to be a transactional database, regardless of the keys accessed inside each transaction (distributed transactions will be further explained in Section 3.4). Before introducing the algorithms in detail, we first provide a high-level intuition on how the guarantees it provides are achieved. As explained in Section 3.2, keys are versioned, incrementing with each write operation on that key and a transaction is a set of operations, each performed on a specific key for a specific version. Transactions are executed atomically, once a commit request arrives at a master, it verifies that each version matches the current version of each key, and tries to acquire locks for all accessed objects atomically (each object is associated with a lock). If it is successful, only then are the values and versions updated for the write operations. If it fails to acquire any lock, the decision is delayed and the process restarts. If at any point the version for a given operation does not match the version for that key in the system, the transaction is aborted. This was designed to provide SconeKV with serializable isolation between transactions, guaranteeing that transactions can be ordered serially and obtain the current system state at any point during execution. According to Adya's formalism [ALO00], serializability prohibits the following anomalies, with w denoting a write, r read, and the subscripts indicating the transaction in which they were executed. The notation "$\ldots$" indicates any series of operations excluding a commit or abort while P refers to a predicate.

- P0 (Dirty Write): $w_1(x) \ldots w_2(x)$ - prevent T_1 from modifying x and T_2 further modifying x before T_1 commits or aborts.
- P1 (Dirty Read): $w_1(x) \ldots r_2(x)$ - prevents T_2 from reading an uncommitted write of T_1, which may later abort.
- P2 (Fuzzy Read): $r_1(x) \ldots w_2(x)$ - prevents the overwriting of an object being read by an uncommitted transaction.
- P3 (Phantom): $r_1(P) \ldots w_2(y \in P)$ - similar to P2, prevents T_2 from from performing a write that satisfies predicate P after an uncommitted transaction performed a read with the same predicate.

SconeKV's design avoids all previously presented anomalies:

- P0: the values of keys and the corresponding versions are only updated when the transaction is committed
- P1: the value written by T_1 in x only exists if T_1 commits, otherwise T_2 will read the previous version of x.
- P2: T_1 aborts if it tries to commit after T_2, as T_2 will increase x's version.
- P3: does not apply to SconeKV, as it does not support predicates.

3.4 Distributed Transactions

3.4.1 Simplified Solution

In this section we address our third objective of providing multi-object 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 3.3 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 3.3).

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

As stated in Section 2.4.2, 2PC 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 3.5) so that any master may be replaced by a replica to complete the protocol.

Figure 3.4 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 2, lines 1-23, some of the auxiliary functions are defined at the end of this chapter in Algorithm 11). 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 3.4) 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 (Algorithm 3). Every master involved will then receive it, replicated inside its bucket and act accordingly, committing (Algorithm 4, lines 25-39) or aborting (lines 41-47) and responding to the client in the case of the transaction coordinator. In either case, the locks are released (lines 49-62) and MAKELOCALDECISION is triggered for the next transactions in the queues of the released locks.

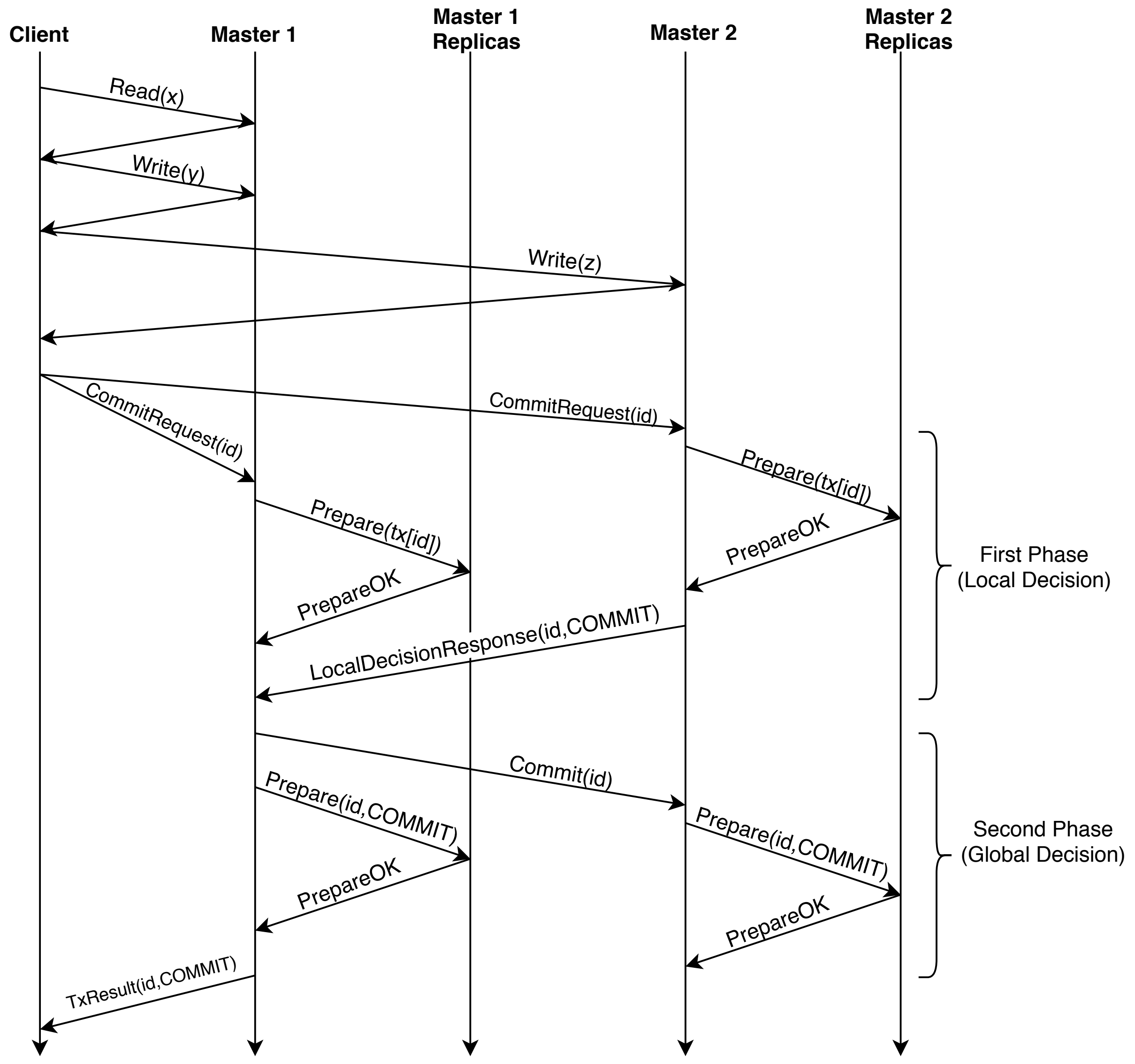

Figure 3.3: Messages exchanged during a transaction involving 2 buckets. Master 1 is the transaction coordinator, both masters accept the transaction locally and thus it is committed. For simplicity, the replicas for each bucket are represented as a single entity each. The first phase of the protocol is defined in Algorithm 2, the second phase is defined in Algorithm 3 and Algorithm 4.

Algorithm 2 Local Decision - First Phase

```
upon event ⟨$master$, MakeLocalDecision | $txID$⟩ do
    if $txs[txID].state \neq$ aborted then
        if CheckValidTransaction($txID$) then        ▷ validate the versions used in the tx
            $owners \leftarrow$ GetLockOwners($txID$)
            if $owners = \varnothing$ then
                AcquireLocks($txID$)
                $txs[txID].state \leftarrow$ prepare-commit
                trigger ⟨Prepare|txs[txID]⟩
            else
                QueueLocks($txID$)
                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
                    for each $t \in owners$ do
                        $otherCoord \leftarrow$ GetCoord($txs[t].nodes$)
                        send ⟨RequestRevertLocalDecision| otherTxID⟩ to otherCoord
                    end for
                end if
            end if
        else
            $txs[txID].state \leftarrow$ prepare-abort
            trigger ⟨Prepare|txs[txID]⟩
        end if
    end if
end event

upon event ⟨$master$, SendLocalDecision | $txID$⟩ do
    $txCoord \leftarrow$ GetCoord($txs[txID].nodes$)
    if $txs[txID].status =$ prepare-commit then
        send ⟨LocalDecisionResponse|txID,commit⟩ to txCoord
    else
        send ⟨LocalDecisionResponse|txID,abort⟩ to txCoord
    end if
end event

function CheckValidTransaction(txID)
    for each $(key, \_, version, \_) \in txs[txID].rwSet$ do
        $currentVersion \leftarrow$ GetVersion($key$)
        if $currentVersion \neq version$ then
            return $False$
        end if
    end for
    return $True$
end function

function GetLockOwners(txID)
    $owners \leftarrow \varnothing$
    for each $(key, \_, \_, \_) \in txs[txID].rwSet$ do
        $lockOwner \leftarrow$ GetLocker($key$)
        if $lockOwner \neq NULL \land lockOwner \notin owners$ then
            $owners \leftarrow owners \cup lockOwner$
        end if
    end for
    return $owners$
end function
```

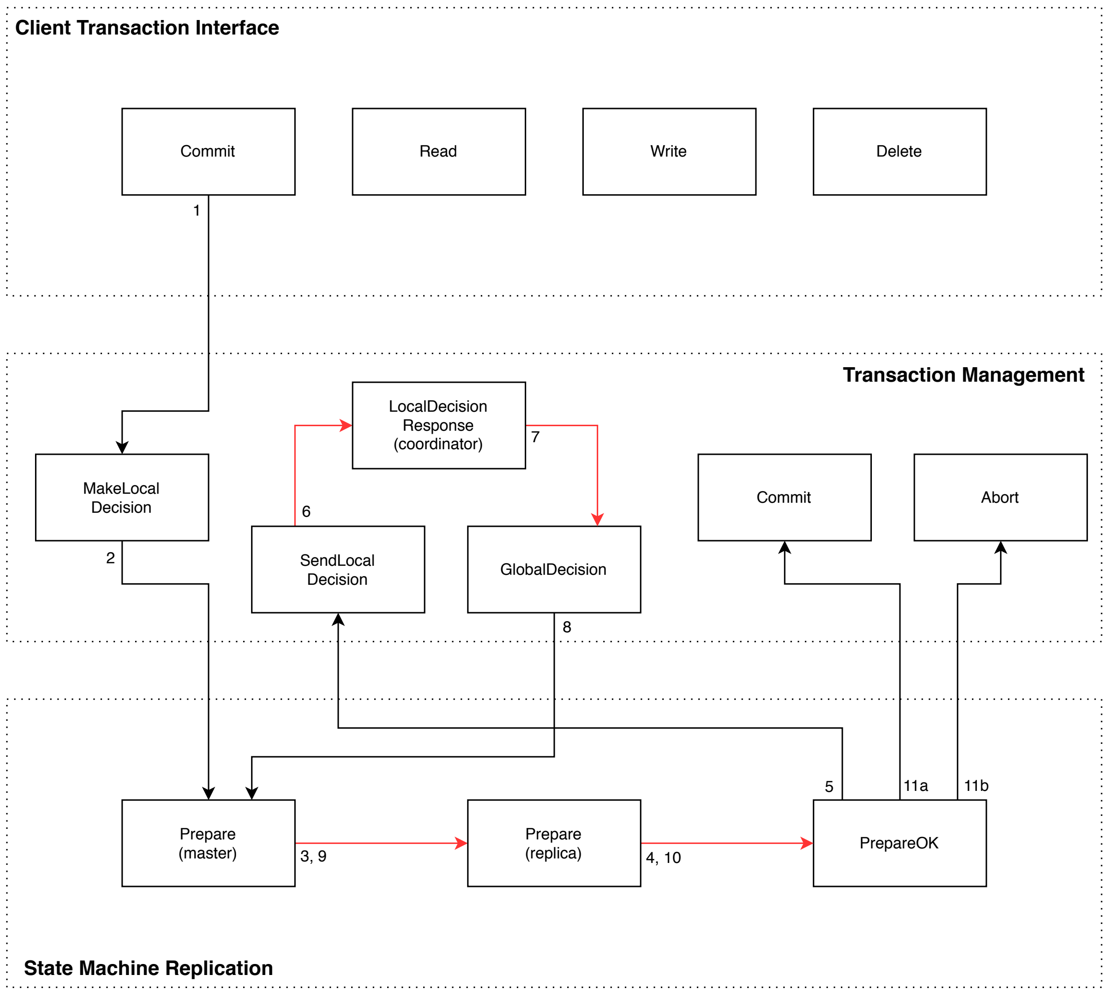

Figure 3.4: Diagram showing the sequence of events to commit a transaction in the general case. The red arrows represent that the next event was triggered in a remote node.

Algorithm 3 Global Decision - Second Phase

```
1:  upon event ⟨$coordinator$, LocalDecisionResponse | $txID, response$⟩ do
2:      if $response = \text{abort} \wedge txs[txID].state \neq \text{aborted} \wedge txs[txID].state \neq \text{to-abort}$ then
        ▷ transaction was rejected locally but not yet globally
3:          for each $node \in txs[txID].nodes$ do
4:              send ⟨GlobalDecision|txID,abort⟩ to node
5:          end for
6:      else                                        ▷ transaction was accepted locally
7:          $txs[txID].responses \leftarrow txs[txID].responses + 1$
8:          if $txs[txID].responses = \#txstxID].nodes$ then
9:              for each $node \in txs[txID].nodes$ do
10:                 send ⟨GlobalDecision|txIDcommit⟩ to node
11:             end for
12:         end if
13:     end if
14: end event
15:
```

Algorithm 4 Global Decision - Second Phase (continuation)

```
upon event ⟨$master$, GlobalDecision | $txID$, $decision$⟩ do
    if $decision$ = commit then
        $txs[txID].state \leftarrow$ to-commit
    else if $decision$ = abort then
        $txs[txID].state \leftarrow$ to-abort
    end if
    trigger ⟨Prepare|txs[txID]⟩
end event

upon event ⟨$master$, Commit | $txID$⟩ do
    for each $(key, value, version, type) \in txs[txID].rwSet$ do
        if $type = WRITE$ then
            $newVersion \leftarrow version + 1$
            Put($key, value, newVersion$)
        else if $type = DELETE$ then
            Delete($key$)
        end if
    end for
    ReleaseLocks($txID$)
    $txs[txID].state \leftarrow$ committed
    if IsCoordinatorTx($txID$) then
        send ⟨TxResult|txID,commit⟩ to txID.client
    end if
end event

upon event ⟨$master$, Abort | $txID$⟩ do
    ReleaseLocks($txID$)
    $txs[txID].state \leftarrow$ aborted
    if IsCoordinatorTx($txID$) then
        send ⟨TxResult|txID,abort⟩ to txID.client
    end if
end event

function ReleaseLocks(txID)
    $restartTx \leftarrow \varnothing$
    for each $(key, \_, \_, \_) \in txs[txID].rwSet$ do
        if UnlockKey($key, txID$) then
            $nextInQueue \leftarrow$ GetNextInLockQueue($key$)
            if $nextInQueue \neq NULL \land nextInQueue \notin restartTx$ then
                $restartTx \leftarrow restartTx \cup nextInQueue$
            end if
        end if
    end for
    for each $tx \in restartTx$ do
        trigger ⟨MakeLocalDecision|tx⟩
    end for
end function
```

3.4.2 Avoiding Distributed Deadlocks

The design presented thus far was simplified and may lead to distributed deadlocks. Assume that two transactions, T_A and 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 x for T_A and add T_B to the queue, while M_2 would lock y 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 needs to abort as they are both conflicting. Nevertheless, there are other, more complex examples where both transactions could commit and respect serializability if ordered correctly. Figure 3.5 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 sequence that occurs inside a SconeKV in the case of a revert of a local decision is exemplified in Figure 3.6. The protocol is triggered during MAKELOCALDECISION, back in Algorithm 2. If a transaction fails to acquire the locks, it queues them and determines if it should be processed before all transactions currently owning any of those locks (Algorithm 2, lines 11-17). Assuming it should, the coordinators for the reverted transactions will confirm that those have not reached a global decision yet (Algorithm 5, lines 1-6), 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 (lines 8-11) and, once it is consistently replicated, release the locks (lines 13-16), eventually processing MAKELOCALDECISION once more for the transaction that triggered the protocol.

Algorithm 5 Transactions - Revert decision events

```
1:  upon event ⟨$coordinator$, RequestRevertLocalDecision | $txID$⟩ do
2:      if $txs[txID].state \neq$ to-commit $\vee$ $txs[txID].state \neq$ committed then
3:          $txs[txID].responses \leftarrow txs[txID].responses - 1$
4:          send ⟨RevertLocalDecisionResponse|txID⟩ to node
5:      end if
6:  end event
7:
8:  upon event ⟨$master$, RevertLocalDecisionResponse | $txID$⟩ do
9:      $txs[txID].state \leftarrow$ revert
10:     trigger ⟨Prepare|txs[txID]⟩
11: end event
12:
13: upon event ⟨$master$, RevertLocalDecision | $txID$⟩ do
14:     $txs[txID].state \leftarrow$ received
15:     ReleaseLocks($txID$)
16: end event
17:
```

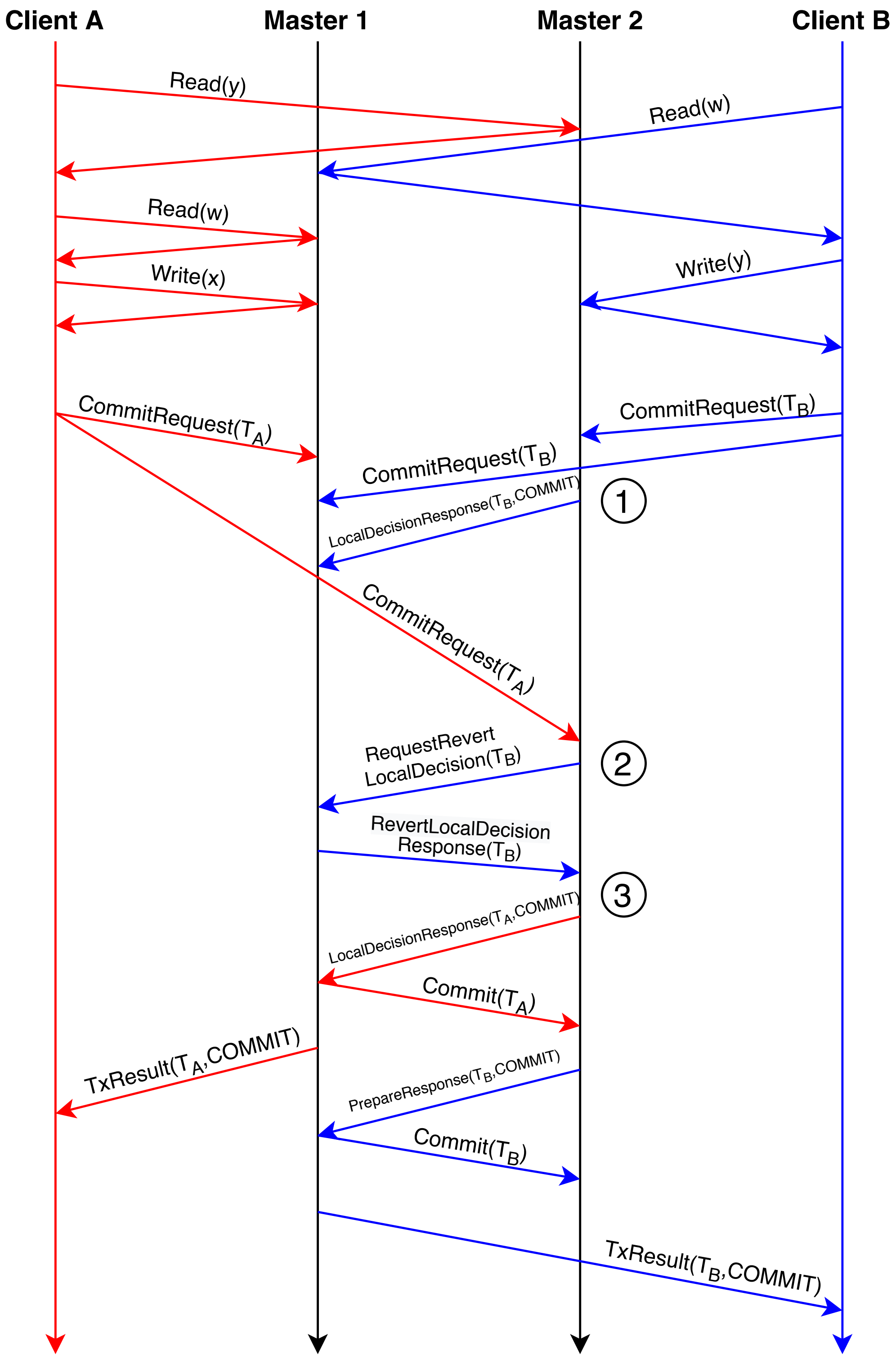

Figure 3.5: Messages exchanged during two concurrent transactions that need to acquire locks for the same objects. (1) Master 2 begins by locking object y for transaction T_B, (2) later receives transaction T_A (which also requires the lock for y and has a lower identifier) and thus asks to revert the 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.

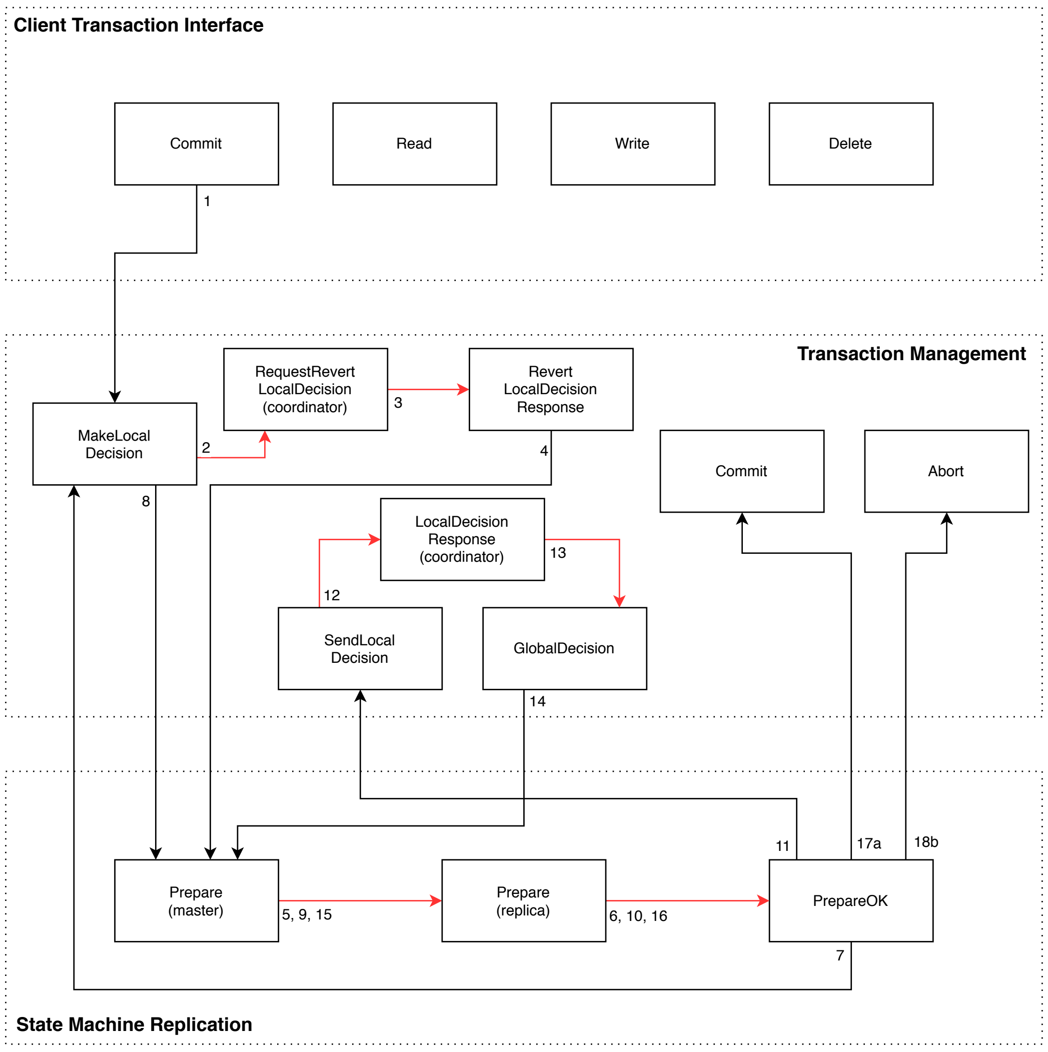

Figure 3.6: This example assumes that some transaction j was processed before and is awaiting a global decision. Now, the system receives the commit request for transaction i that accesses some of the same keys. Assuming that $i < j$, the diagram presented shows the sequence of events to revert the decision for j and commit/abort i.

3.5 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 system state are consistent and their result 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. In Section 2.4.1 we described two similar solutions to this problem: Raft [OO14] and Viewstamped Replication [OL88, LC12]. Both algorithms provide similar guarantees, we opted to adapt VR to our context for two main reasons: no need for leader election and the ease to adapt the view change algorithm to our context (both explained further in Section 3.6).

Our replication protocol is a concretization of the VR protocol described in Section 2.4.1. Its definition is separated, simply for space constraints, into Algorithm 6, Algorithm 7 and Algorithm 8. In summary, entries flow from the master to the replicas. To replicate a log entry, the master issues PREPARE (Algorithm 6, lines 15-23). Once received, each replica processes the entry if and only if it has processed all previous entries (Algorithm 7, lines 25-43), issuing PREPAREOK. When the master receives f PREPAREOK's, the entry is consistently propagated and can be committed (along with all previous entries, Algorithm 7, lines 45-67).

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 preservation of data in the case of catastrophic failures, such as all replicas crashing at the same time. In Chapter 4 we discuss how SconeKV provides durability guarantees.

Algorithm 6 Replication

```
1:  upon event ⟨Init⟩ do
2:      $view \leftarrow \cdots$                      ▷ current view provided by the membership layer
3:      $status \leftarrow \cdots$                    ▷ current status, either normal, view-change or master-after-view-change
4:      $bucketIdx \leftarrow \cdots$                 ▷ current bucket index in the DHT
5:      $currentMaster \leftarrow \cdots$             ▷ current state machine master
6:      $term \leftarrow \cdots$                      ▷ last view change version
7:      $log \leftarrow \cdots$                       ▷ current log of operations
8:      $opNum \leftarrow \cdots$                     ▷ largest operation number seen
9:      $commitNum \leftarrow \cdots$                 ▷ largest operation number committed
10:     $futureViewVersion \leftarrow \varnothing$    ▷ largest view version received
11:     $startViews \leftarrow 0$                     ▷ collection of StartView events
12:     $doViews \leftarrow \varnothing$              ▷ collection of DoView events
13: end event
14:
15: upon event $\langle master, \text{Prepare} \mid tx \rangle$ do
16:     if $status = normal$ then
17:         $log \leftarrow log \cup tx$
18:         $opNum \leftarrow opNum + 1$
19:         for each $node \in view.bucket[bucketIdx]$ do
20:             send ⟨Prepare|tx, opNum, commitNum⟩ to node
21:         end for
22:     end if
23: end event
24:
```

Algorithm 7 Replication (continuation)

```
upon event $\langle replica, \text{Prepare} \mid tx, entryOpNum, masterCommitNum\rangle$ do
    if $status = normal$ then
        if $entryOpNum \neq opNum + 1$ then
            $pendingEntries[entryOpNum] \leftarrow pendingEntries \cup \{tx, commitNum\}$
            if $entryOpNum - opNum \geq$ MaxOpNumberHole then
                send ⟨GetState|self, currentVersion, opNum⟩ to master
            end if
        else
            $log \leftarrow log \cup entry$
            $opNum \leftarrow entryOpNum$
            $commitNum \leftarrow masterCommitNum$
            if $opNumber + 1 \in pendingEntries$ then
                $pendingEntry, pendingCommitNum \leftarrow pendingEntries[opNum + 1]$
                trigger ⟨Prepare|pendingEntry, opNum + 1, pendingCommitNum⟩
            end if
            send ⟨PrepareOK|self, opNum⟩ to master
        end if
    end if
end event

upon event $\langle master, \text{PrepareOK} \mid replica, replicaOpNum\rangle$ do
    if $status \neq$ view-change then
        for each $i \in [commitNum + 1, replicaOpNum[$ do
            $tx \leftarrow log[i]$
            $tx.oks \leftarrow tx.oks \cup replica$
            if $\#tx.oks \geq f$ then
                $commitNum \leftarrow i$
                if $tx.state =$ prepared-commit $\vee$ $tx.state =$ prepared-abort then
                    trigger ⟨SendLocalDecision|tx.txID⟩
                else if $tx.state =$ to-commit then
                    trigger ⟨Commit|tx.txID⟩
                else if $tx.state =$ to-abort then
                    trigger ⟨Abort|tx.txID⟩
                else if $tx.state =$ revert then
                    trigger ⟨RevertLocalDecision|tx.txID⟩
                end if
            end if
        end for
        if $status =$ master-after-view-change $\wedge$ $commitNum = opNum$ then
            $status \leftarrow normal$
        end if
    end if
end event

upon event $\langle master, \text{GetState} \mid replica, replicaViewVersion, replicaOpNum\rangle$ do
    if $currentVersion = replicaViewVersion$ then
        $logFragment \leftarrow \varnothing$
        for each $i \in ]replicaOpNum, opNum]$ do
            $logFragment \leftarrow logFragment \cup log[i]$
        end for
        send ⟨NewState|logFragment, opNum, commitNum⟩ to replica
    end if
end event

```

Algorithm 8 Replication (continuation)

```
79: upon event $\langle replica, \text{NewState} \mid missingEntries, masterOpNum, masterCommitNum\rangle$ do
80:     $log \leftarrow log \cup missingEntries$
81:     $opNum \leftarrow masterOpNumber$
82:     $commitNum \leftarrow masterCommitNum$
83: end event
84:
```

3.6 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 [OL88, LC12] 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 (any 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 request 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 this node receives a PREPARE event with an *opNumber* MAXOPNUMBERHOLE entries after its current most recent entry in the log (as seen before in Algorithm 7).

Now if the current master has changed, either because the previous one failed or our function determines that a new node is the master, then we run Algorithm 9 and Algorithm 10 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 serializability of transactions it is required to be working in stable conditions. The transactions between view change states are depicted in Figure 3.7.

Algorithm 9 View change

```
upon event ⟨ViewUpdate | $newView$⟩ do          ▷ triggered by the membership layer
    $view \leftarrow newView$
    if $bucketIdx \neq \text{GetBucket}(view, self)$ then
        $bucketIdx \leftarrow \text{GetBucket}(view, self)$
        $currentMaster, term, log, opNum, commitNum \leftarrow \bot$
    end if
    if $futureViewVersion < view.version$ then
        $futureViewVersion \leftarrow view.version$
        $startViews \leftarrow 0$
        $doViews \leftarrow \varnothing$
    end if
    $master \leftarrow \text{GetMaster}(view.bucket[bucketIdx])$
    if $view.version = futureViewVersion \land master \neq currentMaster$ then
        $status \leftarrow$ view-change
        for each $node \in view.bucket[bucketIdx]$ do
            send ⟨StartViewChange|view.version⟩ to node
        end for
    end if
end event

upon event ⟨StartViewChange | $newViewVersion$⟩ do
    if $newViewVersion > futureViewVersion$ then
        $futureViewVersion \leftarrow newViewVersion$
        $startViews \leftarrow 1$
        $doViews \leftarrow \varnothing$
    else if $newViewVersion = futureViewVersion$ then
        $startViews \leftarrow startViews + 1$
        if $newViewVersion = view.version \land startViews \geq f$ then
            $status \leftarrow$ view-change
            $master \leftarrow \text{GetMaster}(view.bucket[bucketIdx])$
            $state \leftarrow \{log, term, opNum, commitNum\}$
            send ⟨DoViewChange|view.version, state⟩ to master
        end if
    end if
end event

```

Algorithm 10 View change (continuation)

```
upon event $\langle master, \text{DoViewChange} \mid newViewVersion, otherState\rangle$ do
    if $newViewVersion > futureViewVersion$ then
        $futureViewVersion \leftarrow newViewVersion$
        $startViews \leftarrow 0$
        $doViews \leftarrow \{otherState\}$
    else if $newViewVersion = futureViewVersion \wedge newViewVersion \neq term$ then
        $doViews \leftarrow doViews \cup \{otherState\}$
        if $newViewVersion = view.version \wedge \#doViews \geq f + 1$ then
            $currentMaster \leftarrow self$
            UpdateLog( )
            for each $node \in view.bucket[bucketIdx]$ do
                $state \leftarrow \{log, term, opNum, commitNum\}$
                send ⟨StartView|self, view.version, state⟩ to node
            end for
            $status \leftarrow$ master-after-view-change
        end if
    end if
end event

upon event $\langle replica, \text{StartView} \mid master, newTerm, newState\rangle$ do
    $term \leftarrow newTerm$
    $log \leftarrow newState.log$
    $opNum \leftarrow newState.opNum$
    $commitNum \leftarrow newState.commitNum$
    $currentmaster \leftarrow master$
    $status \leftarrow$ view-change
    send ⟨PrepareOK|self, opNum⟩ to currentMaster
end event

function UpdateLog( )
    //Updates the log according to the events stored in doViews.
    //Selects as the new log the one with the largest term (largest opNum in case of a tie).
    //Set the opNum to the topmost of the new log.
    //Sets the commitNum to the largest one received.
end function
```

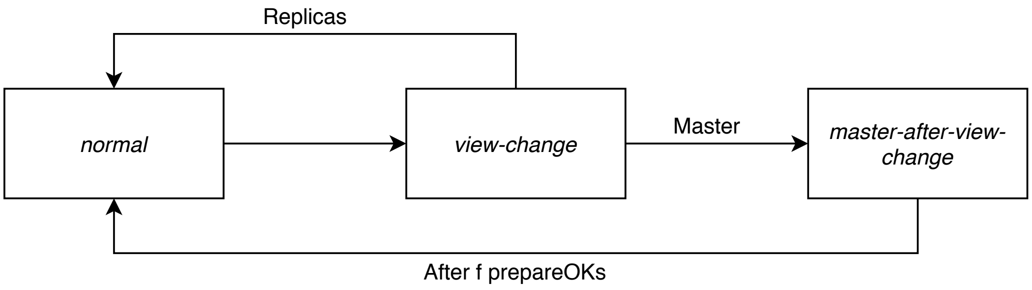

Figure 3.7: Possible state machine *status* and transitions between them.

Algorithm 11 Functions to access the underlying storage

```
function Put(key, value, version)
    //Sets object associated with the key to value and updates version.
end function

function Get(key)
    //Returns current object and version associated with the key.
end function

function Delete(key)
    //Removes the key and associated object from the store.
end function

function GetVersion(key)
    //Returns current object version associated with the key.
end function

function GetCoord(masters)
    //Returns the transaction coordinator from a set of master, implemented as Min.
end function

function GetLocker(key)
    //Returns the identifier of the transaction currently locking object referenced by key.
end function

function LockKey(key, txID)
    //Locks object associated with key for the transaction txID.
end function

function UnlockKey(key, txID)
    //If txID holds the lock on the object associated with key, the lock is released and returns true, otherwise returns false.
end function

function QueueLock(key, txID)
    //Adds txID to the lock queue for key.
end function

function GetNextInLockQueue(key)
    //Returns lowest txID in the lock queue for key.
end function

function AcquireLocks(txID)
    for each $(key, \_, \_, \_) \in txs[txID].rwSet$ do
        LockKey($key, txID$)
    end for
end function

function QueueLocks(txID)
    for each $(key, \_, \_, \_) \in txs[txID].rwSet$ do
        QueueLock($key, txID$)
    end for
end function
```

3.7 Summary

In this chapter we presented SconeKV, a distributed key-value store designed to provide strong consistency guarantees at a large scale. To summarize, SconeKV achieves its main objectives as follows:

- **Maintain a total view of the system with strong consistency guarantees** — leveraging a membership layer with strong consistency guarantees at scale - PRIME.
- **Guarantee strong consistency on all operations** — the design of our transaction decision algorithm that guarantees serializability.
- **Provide multi-object distributed transactions** — the design of our client protocol and distributed decision algorithm.
- **Guarantee the replication factor** — applying a SMR protocol inside each bucket, ensuring consistent replication.
- **Tolerate membership changes** — the combination of the membership layer detecting the failures and the view change algorithm ensuring that each bucket remains in a consistent state.

Chapter 4

Implementation

In this chapter we discuss the implementation details of the SconeKV prototype. It was developed using Java 13, mainly because PRIME[San18] was also developed in Java using implementations of EpTO [MMF$^+$15] and CYCLON[VGVS05] written in Kotlin. Our implementation also depends on ØMQ[1] for TCP communication, Cap'n Proto[2] for message serialization and RocksDB[3] for durability. The rest of this chapter is organized as follows: in Section 4.1 we present an overview of a SconeKV node, its components, and how they interact; next, in Section 4.3 we discuss the durability guarantees provided by the system; following that, in Section 4.4 we address the client library and the SconeKV API; in Section 4.5 we analyze the bootstrap process for a node; and finally in Section 4.6 we discuss some optimizations made during the implementation process.

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

4.1 Node Overview

In this section, we describe a SconeKV node, its components, and how they interact. To facilitate understanding, we present a diagram of these components in Figure 4.1.

In order to resemble more closely the event-based algorithms presented in Chapter 3, we also opted to employ an event based architecture in our implementation. There is a FIFO event queue in which events are pushed and popped by a configurable number of worker threads. This design choice greatly simplified the implementation, especially given that a single event might trigger multiple others and even show a recursive behaviour.

The workers remove events from the queue, performing interactions with the various components: the Membership Manager - PRIME - which triggers view update events; the in-memory Key-Value Store (which also interacts with RocksDB to persist updates, explained in Section 4.3); the State Machine Manager that coordinates replication according to our adaptation of VR; and a Communications Manager that handles TCP connections (presented in Section 4.2).

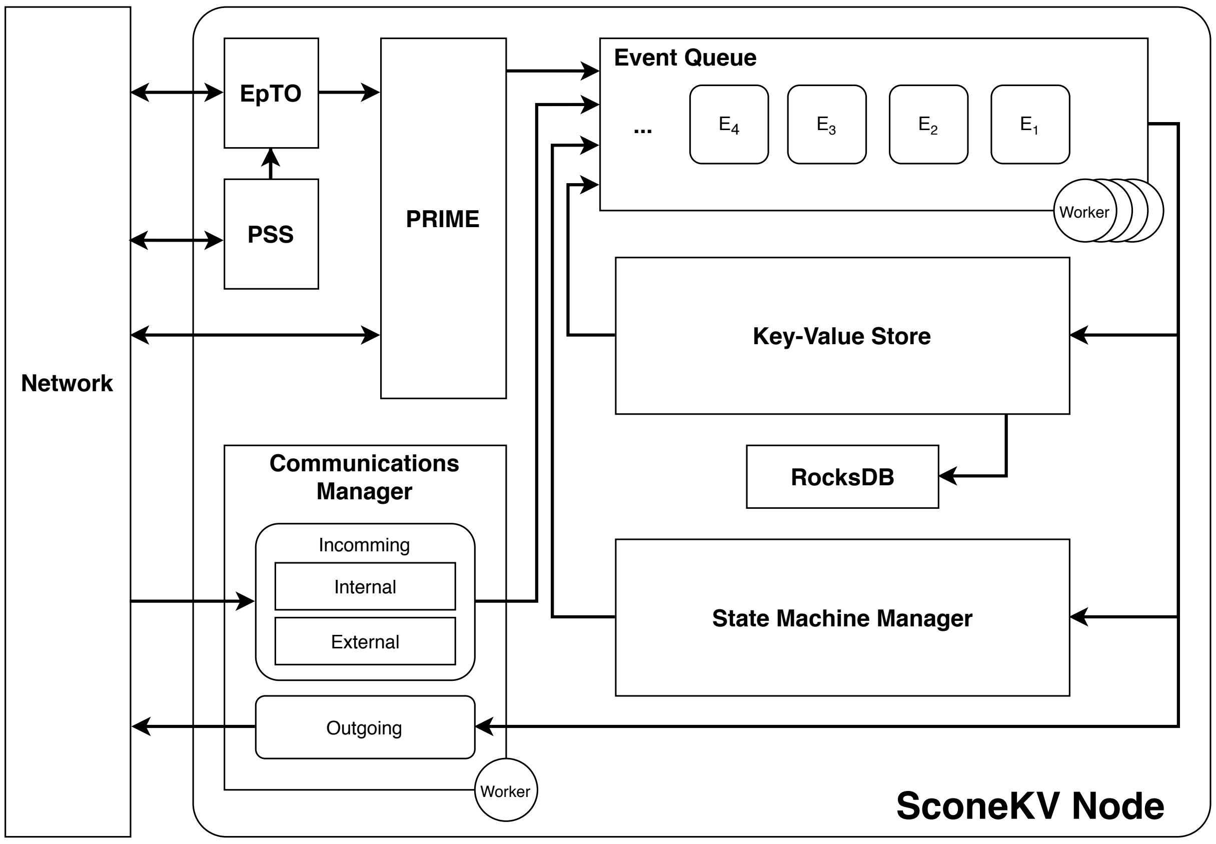

Figure 4.1: Components inside a SconeKV Node and their interactions

4.2 Communication

The most important decision regarding communication was whether to use UDP or TCP. We decided on TCP for both internal (between nodes) and external (with clients) communications. Being a database, clients usually establish a connection that spans for a relatively extended period of time, justifying the TCP overhead. In the case of internal communications, as described in Figure 3.2 connections are only established inside of a bucket (between replicas and master) and between masters. Therefore, we expect the number of connections for each node to grow logarithmically with the size of the system, for most deployments.

The decision to use ØMQ for communication was justified by its abstraction of an asynchronous message queue, contrasting with the typical synchronous behaviour of TCP connections, allowing for application logic to progress while ØMQ's middleware interacts with the physical resources. External and internal communications are routed through different sockets, allowing for prioritization if so desired. Unfortunately, ØMQ's does not handle well multithreaded access to its resources. For this reason, the communication manager has a dedicated thread that receives network messages, creates events, and pushes them to the event queue and also receives messages from other threads to be sent via the network to clients or other nodes. The asynchronous behaviour allows this thread to skip the wait for I/O operations, assuring it does not become a bottleneck for the system.

For message construction, we decided to use Cap'n Proto for message serialization for its efficiency and the expressiveness of the schema language, allowing for easy and clean distinction between message types. Cap'n Proto does not require a platform specific encoding/decoding, arranging data as a struct: with fixed widths, fixed offsets, and proper alignment, embedding variable-sized elements as pointers, with these being offset-based rather than absolute, making messages position-independent. This favors processing efficiency in detriment of space efficiency. If in the future bandwidth becomes a bottleneck for SconeKV, this could be solved applying a general purpose compression algorithm as all the extra padding are zeros.

4.3 Durability

For simplicity of implementation, SconeKV's prototype was developed with the assumption that the keyspace for each bucket fits entirely in memory. State machine replication (in our case VR) ensures reliability, consistency of the data, and availability of the system if no more than f replicas per bucket are faulty. Catastrophic failures (above f nodes per bucket), could result in loss of client data, which is especially relevant in geo-replicated deployments. To mitigate this problem, we employed RocksDB as an efficient solution to persist key-value store updates to disk. As this behaviour is only required in case of catastrophic events, updates are batched for a configurable amount of time and only then persisted to disk, using a similar approach to that of Dynamo [DHJ+07] or Cassandra [LM10].

In practice, this means that every write to the key-value store does not represent a write to disk, multiple updates to a single key can be performed resulting in a single write of the most recent version, depending on the parameterized period. Values are also only read from the disk if the key does not exist in memory. As SconeKV assumes the whole keyspace fits in memory, it does not dump keys to disk for memory efficiency, thus making this behaviour only relevant in the case of a reboot after failing.

4.4 Client Library

SconeKV provides a client library with simple API, exemplified in Listing 4.1. The `SconeClient` manages the client's transactions and connections with the cluster, performing the requests and storing the respective metadata (operations, respective keys, and versions).

```
// Initiate the client, which manages the connections
SconeClient client = new SconeClient();
// Creating a transaction associated with the client
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 4.1: Usage example of the SconeKV client API

The library needs a view of the system in order to route requests correctly without the need for hopping. To achieve this, the client library request a current view during bootstrap, using GETVIEW, contacting any node that might be on the system. As previously discussed in Section 3.6, this view will only be updated if the client makes a request to an incorrect node, in which case it receives a new view in response.

Transaction identifiers (`txID`) are a tuple $\langle \textit{clientID}, i \rangle$. The library generates a *clientID* upon the creation of a `SconeClient` using Java's `UUID` class. The i is an internal counter of transactions, and thus in combination with the guarantees provided by `UUIDs`, it ensures that *txID* are unique as required. Note that, the ordering of transactions to avoid distributed deadlocks (presented in Section 3.4.2) is not fair, as `txIDs` do not have any relation with transaction creation time or commit time. In Section 6.1 we discuss a possible approach to change this behaviour, making transaction ordering fairer.

4.5 Bootstrap

SconeKV's bootstrap depends on PRIME and the deployment selected. The number of buckets and minimum size of each bucket are system parameters. Nodes wait for PRIME to provide a view with the required size, before accepting any connections. After the bootstrap the cluster size can increase or decrease and the system adapts accordingly.

Recall our membership layer - PRIME - requires a messaging layer - EpTO - which in turn relies on a PSS - CYCLON. During the bootstrap process, the PSS needs to be fed a list of possible neighbors and the membership layer needs to determine if it is the first node or not, in order to decide if it should start a new cluster or join an existing one. To bootstrap the system from scratch, we rely on a Python based tracker which is a simple REST application that collects all nodes that contact it, responding with a list of all previous connections, giving no guarantee on whether they are still correct or have failed. Therefore, after starting, each node contacts the tracker and obtains a list of other nodes already in the system, if there are any. Note that the tracker is only used for bootstrapping and never in the regular operation of SconeKV nodes. The client library also uses the tracker during its bootstrap in order to obtain a list of possible servers, selecting one at random and requesting a current view of the system until it is fulfilled. During the evaluation process, discussed in Chapter 5, this became a bottleneck at the start of benchmarks running hundreds of concurrent clients. The solution was to implement a multithreaded version of the tracker, although, in a real world scenario, clients could probably use a list of well known server addresses and skip this entirely.

4.6 Optimizations

We now discuss some system optimizations to the algorithms presented in Chapter 3.

Fast Aborts The algorithm presented in Section 3.4 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 (the MAKELOCALDECISION event guarantees that much, Algorithm 2, line 3). Thus, if a write occurs, increasing the version of that key, all transactions waiting in the queue can be immediately aborted, without needing to run MAKELOCALDECISION once more.

Read-Write Locks Once more addressing the issue of the long lock queues for popular keys, the introduction of read-write locks allows for greater parallelism in read-heavy workloads. This is a system parameter, allowing system administrators to select the desired behaviour (read-write or mutex), as it can further increase the latency of commits for transactions that write to frequently read keys.

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 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 inside that bucket).

Garbage Collection After transactions are committed and the key-value store is updated accordingly, transaction data becomes mostly superfluous, excluding the possibility of a master failure and the need to communicate with the transaction coordinator in order to learn the global decision of a specific transaction, or even a coordinator failing and its replacement asking previous participants of a given transaction if they managed to reach a global decision. Data of completed transactions is thus garbage collected in a configurable interval and according to an also configurable transaction TTL.

Chapter 5

Evaluation

In this chapter, we evaluate SconeKV and compare it with two other state-of-the-art distributed storage systems, Cassandra and CockroachDB, one at each end of the consistency spectrum.

Two of the main objectives in the design of our system (described in Chapter 3) were to provide multi-object distributed transactions whilst guaranteeing the replication factor with strong consistency on client observations. Our objective is to show that SconeKV's design allows it to scale in the number of concurrent clients whilst still offering strongly consistent operations, in the form of distributed serializable transactions.

The evaluation will consider 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.

The chapter is organized as follows: first we present the baselines and the rationale behind them (Section 5.1), following that we detail the setup used for the experiments and introduce the selected benchmark (Section 5.2) and finally we show the results (Section 5.3) and discuss them (Section 5.4).

5.1 Baselines

SconeKV was designed to be a distributed data store that scales whilst maintaining strong consistency guarantees. Thus, we decided to compare it with two distinct distributed key-value stores: one scalable and weakly consistent and another transactional and strongly consistent.

Cassandra [LM10] (discussed in Section 2.6.1) 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, though this change is not enough to considered it strongly consistent given its optimistic replication protocol. Note that, as Cassandra is not transactional, its throughput and goodput values are equal as all operations are considered as committed.

CockroachDB [Lab] (discussed in Section 2.6.2) 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. For simplicity, the system was deployed in insecure mode (without certificates and SSL connections) as this is outside the scope of our work and does not negatively affect any of the metrics we considered.

5.2 Experimental Setup

The system's experimental evaluation was performed using Docker [1] 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, running on the cloudTM[2] GSD cluster hosted at IST-DSI.

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 (the same deployment shown before, in Figure 3.2). Both the nodes and the client ran in Docker container for ease of deployment and communication. In the case of SconeKV, the tracker application ran on the same physical machine as the client, without it affecting the results given it is only used for bootstrap purposes. It is also worth noting that, SconeKV was deployed using all the optimizations presented in Section 4.6.

[1]https://www.docker.com
[2]http://www.cloudtm.eu

YCSB [CST+10] was selected as the benchmark for our experiments as it is a standard for cloud based data store comparison. We employed the existing Cassandra binding [3], whilst for Cockroach we extended the functionality of the JDBC binding [4], providing it with support for transactions with a configurable number of operations. We implemented a YCSB binding for SconeKV that utilizes the client API (shown in Section 4.4), and that also supports the configuration for the size of transactions. As Cassandra does not support transactions it was ran as a regular NoSQL database with independent operations, both SconeKV and Cockroach ran using transactions of 5 operations. 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 ran 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 [CST+10].

Each experiment (combination of workload and number of concurrent clients) was ran three times, with a duration of 300 seconds. The throughput and goodput metrics were collected using the YCSB client output, with the resources being captured using `dstat` [5] with a 1 second delay, running on each physical machine.

5.3 Results

5.3.1 Throughput and Goodput

Workload A This is an update heavy workload, the throughput and goodput of the systems is shown in Figure 5.1 and Figure 5.2, 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 additional work-

[3] https://github.com/brianfrankcooper/YCSB/tree/master/cassandra
[4] https://github.com/brianfrankcooper/YCSB/tree/master/jdbc
[5] https://linux.die.net/man/1/dstat

load with an uniform distribution (writes and reads are evenly distributed across all keys). The results are depicted in Figure 5.3. 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.

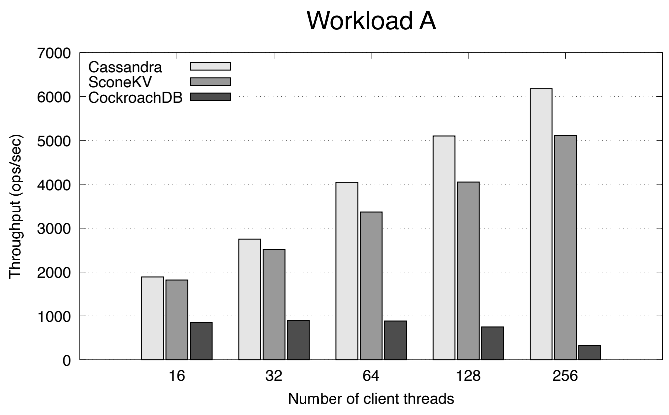

Figure 5.1: Throughput for the three systems using YCSB Workload A.

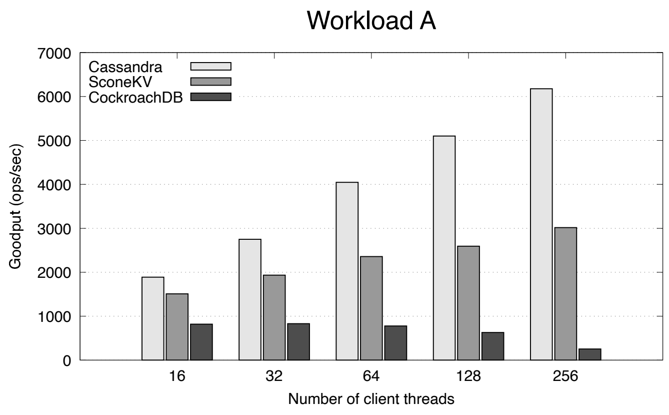

Figure 5.2: Goodput for the three systems using YCSB Workload A.

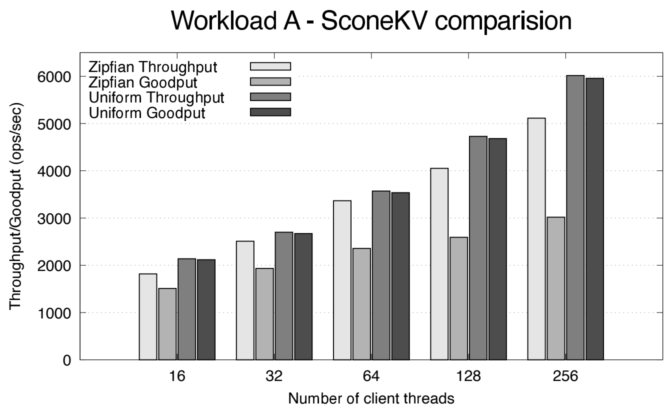

Figure 5.3: Comparison of throughput and goodput for SconeKV, depending on the workload distribution (*zipfian* vs uniform)

Workload B This is a read heavy workload. The results displayed in Figure 5.4 and Figure 5.5 show that SconeKV still scales, though at a lower rate than Cassandra. This can be explained, as SconeKV does not differentiate writes from reads, applying the same protocol to decide transactions outcomes. It is noteworthy that Cockroach stagnates when reaching hundreds of concurrent clients, demonstrating that 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 5.5. This is expected, due to the low rate of concurrent updates.

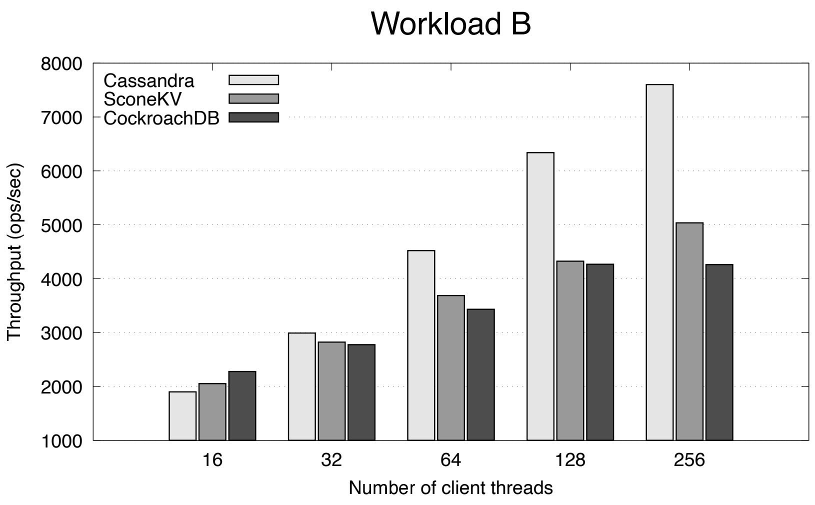

Figure 5.4: Throughput for the three systems using YCSB Workload B.

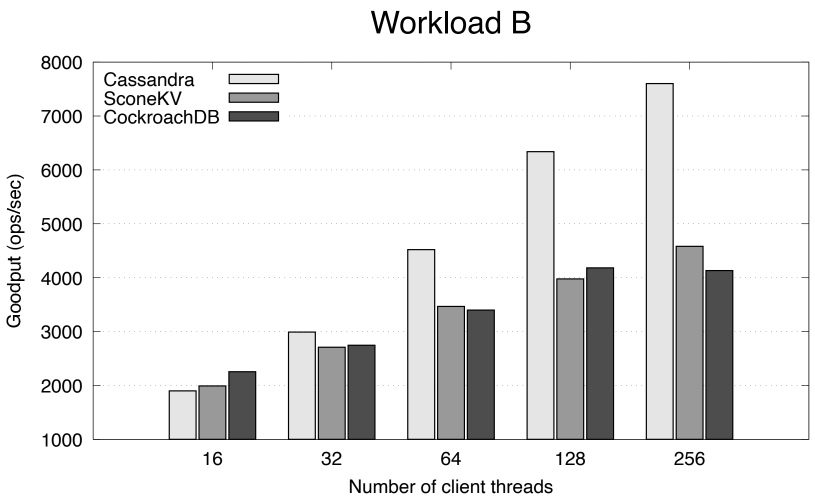

Figure 5.5: Goodput for the three systems using YCSB Workload B.

Workload C This is a read only workload. For that reason we only present the goodput of the three systems (Figure 5.2), 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 explained in two ways: the fact 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, in order to provide better consistency guarantees.

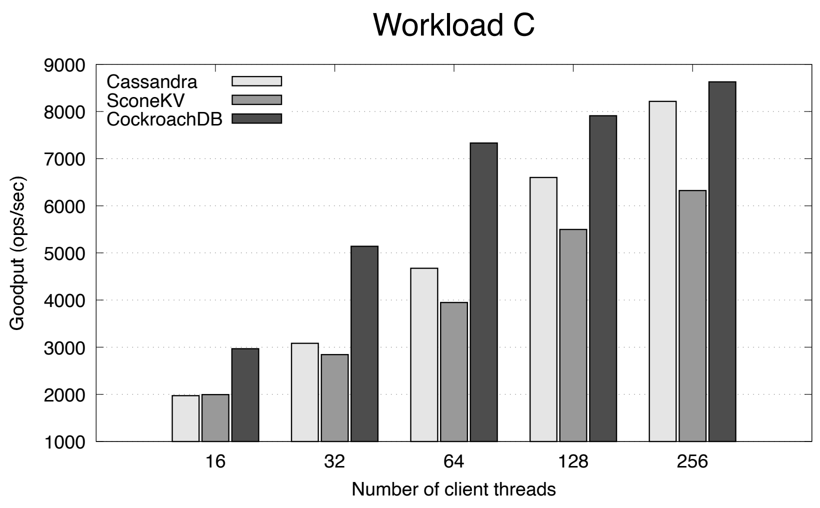

Figure 5.6: Goodput for the three systems using YCSB Workload C.

Workload F This workload selects keys, according to the distribution of requests, and 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 5.7), however Figure 5.8 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 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).

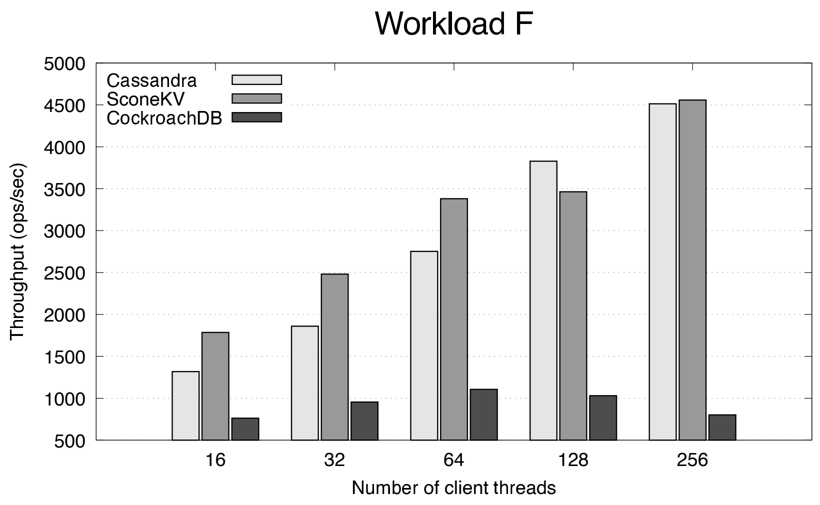

Figure 5.7: Throughput for the three systems using YCSB Workload F.

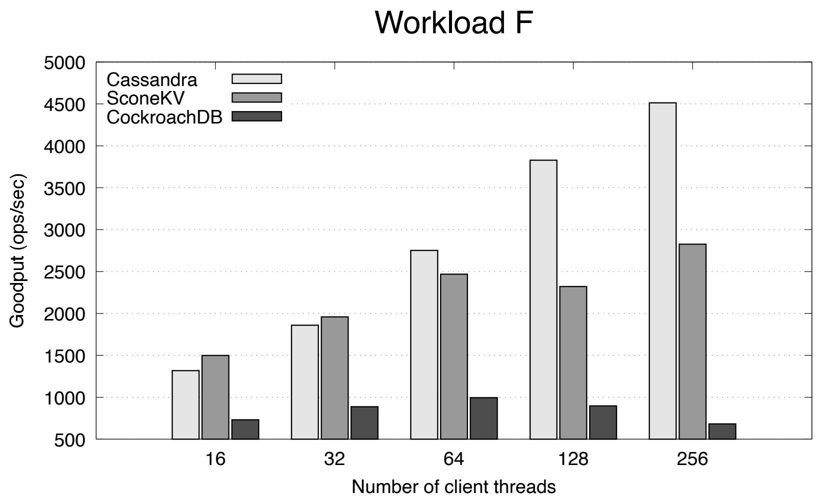

Figure 5.8: Goodput for the three systems using YCSB Workload F.

5.3.2 Resource Usage

In this section, we analyze the three systems with regards to their resource usage, namely CPU, memory, network and disk. Every metric will be presented during a update-heavy workload (YCSB Workload A) and a read-only workload (YCSB Workload C). These metrics were captured using `dstat` in each of the 5 physical machines running the containers for the storage nodes, in order to minimize the impact of these measurements. The results displayed represent the percentiles for those metrics.

CPU Usage Figure 5.9 and Figure 5.10 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 90th 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.

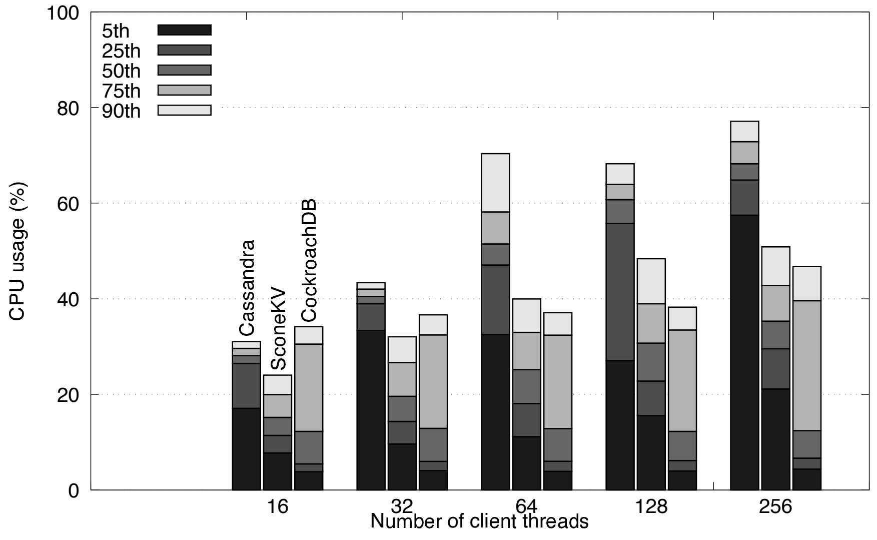

Figure 5.9: CPU usage during update heavy Workload A.

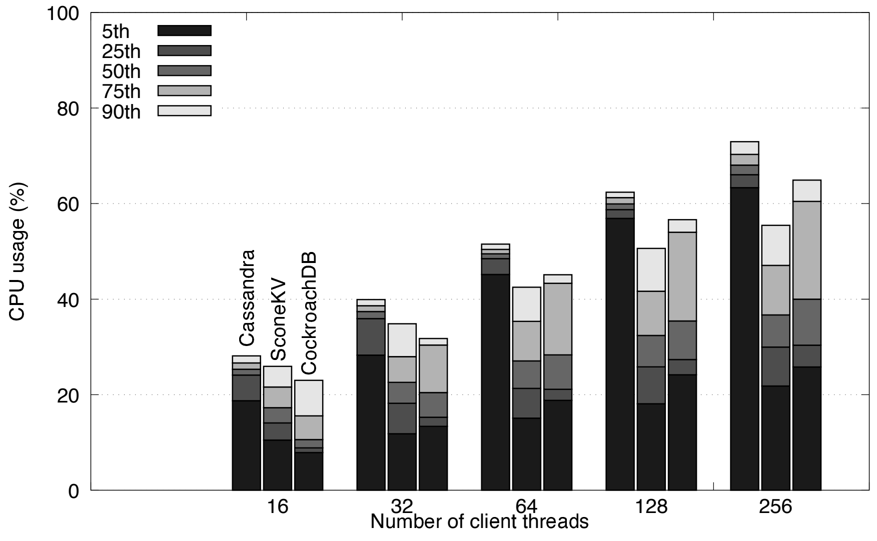

Figure 5.10: CPU usage during read-only Workload C.

Memory Usage Figure 5.11 and Figure 5.12 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 provided much less throughput (shown in Figure 5.1). With a read heavy workload (in which it surpassed the other two system 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.

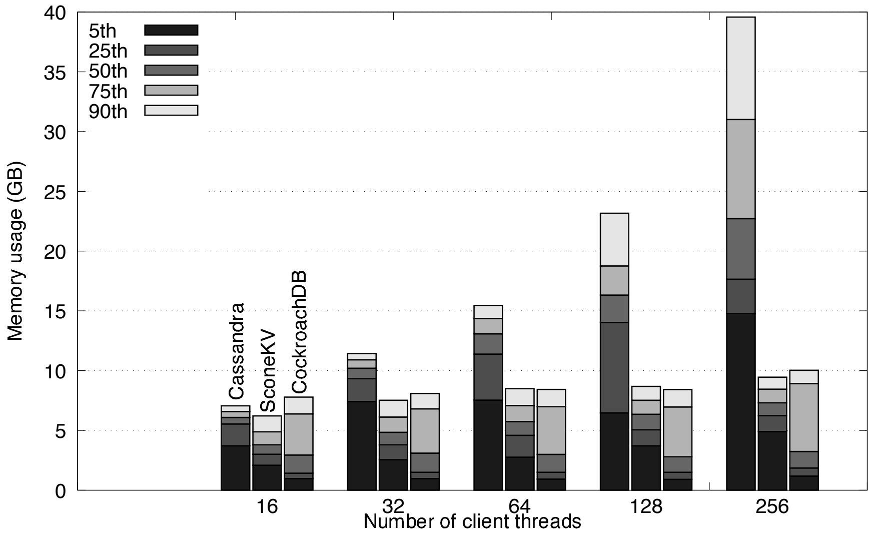

Figure 5.11: Memory usage during update heavy Workload A.

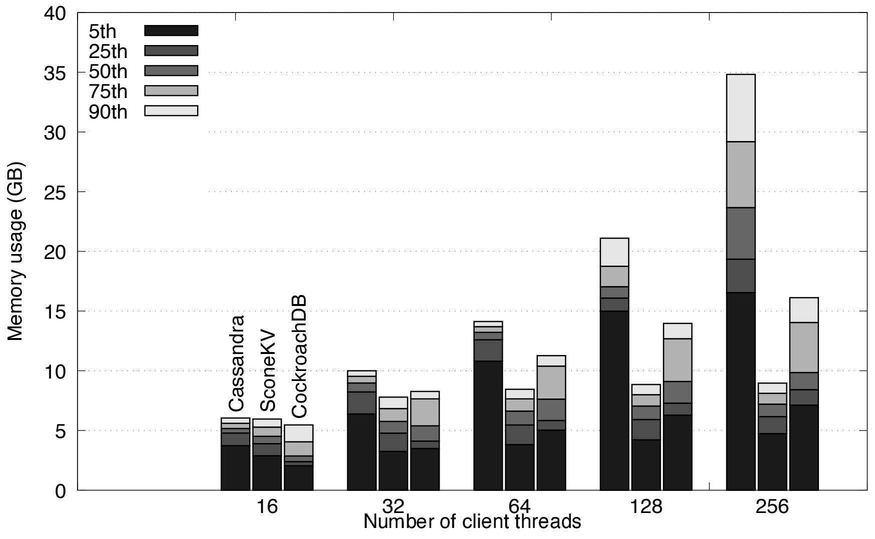

Figure 5.12: Memory usage during read-only Workload C.

Bandwidth Usage For this metric we measured the bytes sent and received over time in the physical machines, once again constructing percentile graphs. Figure 5.13 and Figure 5.14 depict the bandwidth usage in terms of upload, while Figure 5.15 and Figure 5.16 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 udpate heavy workload, which is understandable given it is responding to much less 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.

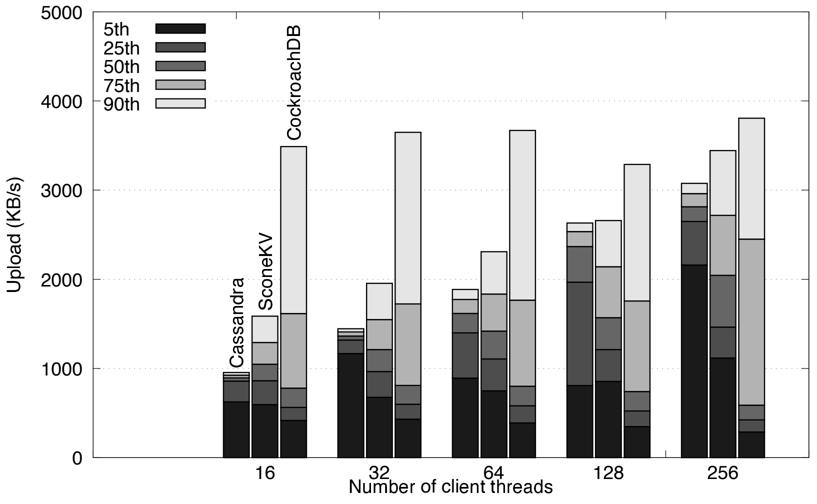

Figure 5.13: Bandwidth usage during update heavy Workload A, upload.

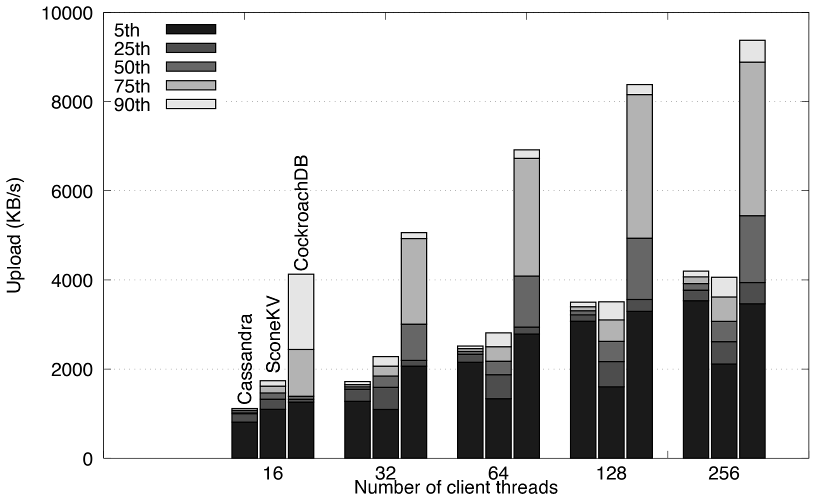

Figure 5.14: Bandwidth usage during read-only Workload C, upload.

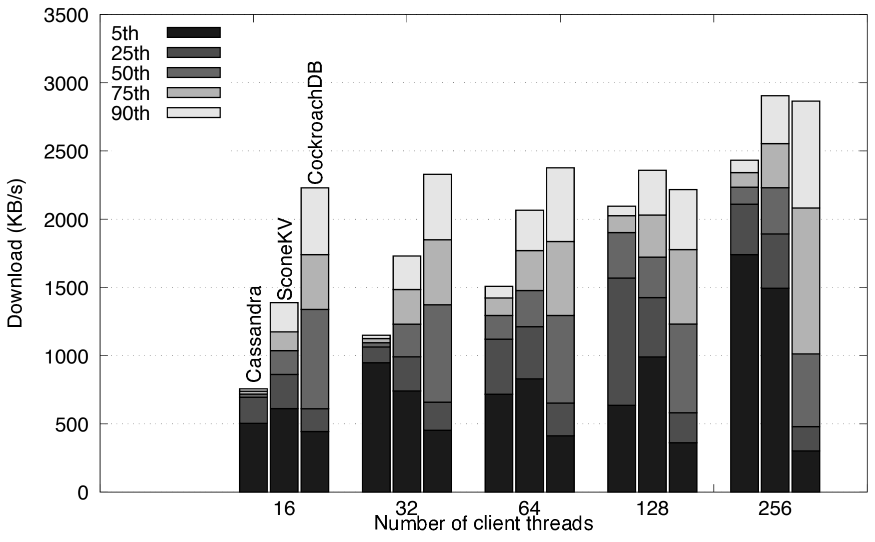

Figure 5.15: Bandwidth usage during update heavy Workload A, download.

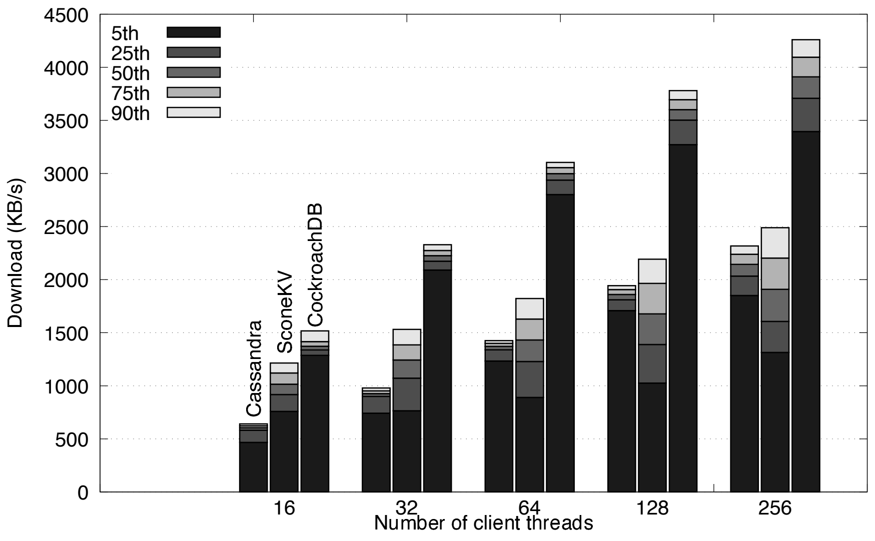

Figure 5.16: Bandwidth usage during read-only Workload C, download.

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 5.17 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 for 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 this experiments), and thus the impact is negligible. For read heavy workloads (Figure 5.18) Cockroach still writes much more to disk than the alternatives, but an order of magnitude less than in the update heavy workload and providing 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 having more data to persist to disk.

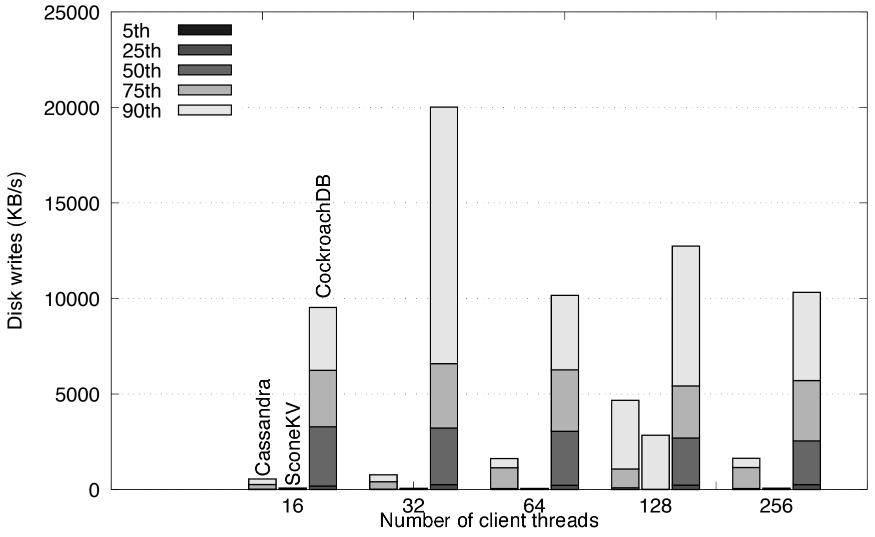

Figure 5.17: Disk write usage during update heavy Workload A.

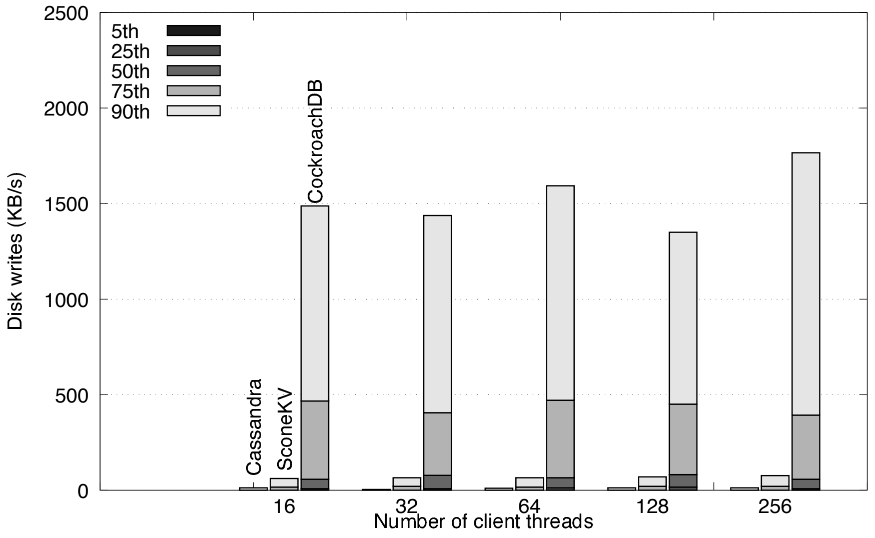

Figure 5.18: Disk write usage during read-only Workload C.

5.4 Discussion

This chapter presented and analyzed the experimental evaluation of SconeKV, comparing it with two state-of-the-art systems, one at each end of the consistency spectrum. The experimental results show that, while not displaying the same raw performance as some competitors in certain scenarios, SconeKV can scale as desired, with representative workloads.

SconeKV is able to perform as required while providing serializable isolation on distributed transactions. It should be noted that Cassandra only provides eventual consistency, while Cockroach does not support concurrent updates at this level of scale, as workloads A and F especially demonstrated in Figure 5.1 and Figure 5.7, respectively. It should be taken into account that this is a prototype and unoptimized, Section 6.1 details some of the possible improvements that could make it even closer to Cassandra regarding performance, without compromising the consistency guarantees provided.

SconeKV requires fewer resources than its competitors in most of the metrics and scenarios tested. We could also conclude that Cockroach's disk utilization is the most probable justification for the poor performance it exhibits in write heavy workloads. This problem is not observed in either Cassandra or SconeKV.

It is also important to note that Cassandra's membership problems affect not only the consistency guarantees it is able to provide during view changes, but they also impact the bootstrapping of the system. Cassandra does not support multiple nodes joining the membership at the same time, requiring nodes to be deployed one by one as soon as the previous has joined. This is a severe drawback when compared with both SconeKV and Cockroach, both supporting all the entire cluster at once and achieving a complete view in less than a minute.

Due to resource constraints, we were not able to evaluate the scalability of SconeKV in terms of the number of nodes, the same applying to geo-replicated deployments. Ideally, we would have deployed the three systems with 40, 80, 120 nodes and displayed the results, but the hardware at our disposal was not enough for such experiments. In any case, the design of SconeKV gives us confidence that it would scale horizontally with a proportional number of buckets. The membership layer is known to scale to hundreds of nodes [San18]. If the transactions have the same size as those used in the previous experiments, the number of buckets involved in each transaction would remain similar and thus maintain the complexity of the decision protocol.

Chapter 6

Conclusion

Classic relational databases provided a strong foundation for application development with their transactional support and consistency guarantees. Unfortunately, their design based on classical consensus does not reach the scale required today, compromising availability and performance. Today's large scale, worldwide applications have turned to scalable storage solutions that provide the availability that users demand. These systems achieve this by relaxing their consistency guarantees and using optimistic replication, in conjunction, allowing for concurrent updates that result in inconsistent observations by clients. Some systems dealt with this problem, delegating the complexity of conflict resolution to the application developers. To provide strong consistency at a large scale, storage systems require a membership solution that scales whilst providing strongly consistent views across the cluster.

In this thesis, we have designed and implemented SconeKV: a strongly consistent key-value storage with support for distributed transactions. It 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. SconeKV leverages a scalable membership layer with strong probabilistic consistency guarantees - PRIME. This allows it to consistently determine bucket masters without the need for extra agreement (leader election) and enables the use of an agreement protocol with a weak liveness property (2PC), reducing the impact on system availability. The experimental evaluation compared SconeKV with two state-of-the-art systems, one at each end of the consistency spectrum. Cassandra on one end, highly scalable, eventually consistent. CockroachDB on the other end, guaranteeing strong consistency and providing ACID properties. 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 while providing serializable distributed transactions.

6.1 Future Work

Meaningful Transaction Identifiers The current approach for generating *txIDs* is simply concerned with them being unique. To avoid distributed deadlocks, we employ a pessimistic ordering mechanism for transactions that is not fair and can even lead to unnecessary reversion of local decisions. Hybrid logical clocks [KDM+14] could be a useful addition, substituting the current client transaction counter. This would enable fairer ordering of transactions (basing them on commit time), possibly even allowing for consistent snapshots of the system.

Distributed Snapshot Isolation Snapshot isolation is still strongly consistent while allowing for greater concurrency given its optimistic behaviour. A distributed implementation would require additional synchronization regarding the attribution of global timestamps, and thus a study should be performed in order to determine its viability.

Log Compaction SconeKV's prototype is not optimized, especially in terms of its memory utilization. The current optimization does not flush the state machine log to disk, which would result in memory exhaustion in long running applications. Another possible optimization is that updates are eventually overridden. Log construction as it is hinders the ability to compact the log, removing old, irrelevant updates. Altering the log construction from the committed transaction simply to database writes would enable such log compaction. This improvement would also decrease the time required to add a new machine to a bucket, reducing the number of operations required for it to achieve the same consistent state.

Evaluate SconeKV Further As discussed in Section 5.4, due to resource constraints we were not able to evaluate SconeKV's horizontal scalability nor deploy it in geo-replicated scenarios. An important direction for future work comprises a deeper study of SconeKV in those scenarios, comparing its performance and scalability with the baselines.

Bibliography

[ALO00] Atul Adya, Barbara Liskov, and Patrick O'Neil. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073)*, pages 67–78. IEEE, 2000.

[ALR13] Sérgio Almeida, João Leitão, and Luís Rodrigues. Chainreaction: a causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 85–98, 2013.

[Apa] Apache. [cassandra-9667] strongly consistent membership and ownership.

[BBG$^+$95] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. *ACM SIGMOD Record*, 24(2):1–10, 1995.

[BHO$^+$99] Kenneth P Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal multicast. *ACM Transactions on Computer Systems (TOCS)*, 17(2):41–88, 1999.

[Bre00] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, pages 343477–343502. Portland, OR, 2000.

[CDE$^+$13] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[CDG$^+$08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[Con] Chris Conrad. Norbert.

[CS93] D Cheriton and Dale Skeen. Understanding the limitations of totally order communications. In *Fourteenth ACM Symposium on Operating System Principles, Operating System Review, Asheville, NC*, volume 27, 1993.

[CST+10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[DGM02] Abhinandan Das, Indranil Gupta, and Ashish Motivala. Swim: Scalable weakly-consistent infection-style process group membership protocol. In *Proceedings International Conference on Dependable Systems and Networks*, pages 303–312. IEEE, 2002.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[DPC18] Armon Dadgar, James Phillips, and Jon 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)*, pages 22–25. IEEE, 2018.

[EGH+03] P Th Eugster, Rachid Guerraoui, Sidath B Handurukande, Petr Kouznetsov, and A-M Kermarrec. Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems (TOCS)*, 21(4):341–374, 2003.

[EGKM04] PT Eugsterand, R Guerraoui, AM Kermarrec, and L Massoulie. From epidemics to distributed computing. *IEEE computer*, 37(5):60–67, 2004.

[FP02] Pascal Felber and Fernando Pedone. Probabilistic atomic broadcast. In *21st IEEE Symposium on Reliable Distributed Systems, 2002. Proceedings.*, pages 170–179. IEEE, 2002.

[GHSV19] Rachid Guerraoui, Jad Hamza, Dragos-Adrian Seredinschi, and Marko Vukolic. Can 100 machines agree? *arXiv preprint arXiv:1911.07966*, 2019.

[GKM01] Ayalvadi J Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. Scamp: Peer-to-peer lightweight membership service for large-scale group communication. In *International Workshop on Networked Group Communication*, pages 44–55. Springer, 2001.

[Has] HashiCorp. Serf documentation.

[HDYK04] Naohiro Hayashibara, Xavier Defago, Rami Yared, and Takuya Katayama. The/spl phi/accrual failure detector. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.*, pages 66–78. IEEE, 2004.

[HKJR10] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8. Boston, MA, USA, 2010.

[JVG$^+$07] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten Van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS)*, 25(3):8, 2007.

[KBC$^+$00] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 190–201. ACM, 2000.

[KDM$^+$14] Sandeep S Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. Logical physical clocks. In *International Conference on Principles of Distributed Systems*, pages 17–32. Springer, 2014.

[KJ10] Manos Kapritsos and Flavio Paiva Junqueira. Scalable agreement: Toward ordering as a service. In *HotDep*, 2010.

[KLL$^+$97] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC*, volume 97, pages 654–663, 1997.

[Kol04] Boris Koldehofe. Simple gossiping with balls and bins. *Stud. Inform. Univ.*, 3(1):43–60, 2004.

[L$^+$01] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[Lab] Cockroach Labs. Cockroachdb.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[Lam06] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[LC12] Barbara Liskov and James Cowling. Viewstamped replication revisited. 2012.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[LPR07] João Leitão, José Pereira, and Luis Rodrigues. Hyparview: A membership protocol for reliable gossip-based broadcast. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 419–429. IEEE, 2007.

[MMF$^+$15] Miguel Matos, Hugues Mercier, Pascal Felber, Rui Oliveira, and José Pereira. Epto: An epidemic total order algorithm for large-scale distributed systems. In *Proceedings of the 16th Annual Middleware Conference*, pages 100–111. ACM, 2015.

[OL88] Brian M Oki and Barbara 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*, pages 8–17, 1988.

[OO14] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.

[RD01] Antony Rowstron and Peter 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*, pages 329–350. Springer, 2001.

[RFH$^+$01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. *A scalable content-addressable network*, volume 31. ACM, 2001.

[RGR$^+$04] Sean Rhea, Dennis Geels, Timothy Roscoe, John Kubiatowicz, et al. Handling churn in a dht. In *Proceedings of the USENIX Annual Technical Conference*, volume 6, pages 127–140. Boston, MA, USA, 2004.

[RJ08] Benjamin Reed and Flavio P Junqueira. A simple totally ordered broadcast protocol. In *proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, page 2. ACM New York, NY, USA, 2008.

[RV11] Etienne Riviere and Spyros Voulgaris. Gossip-based networking for internet-scale distributed systems. In *International Conference on E-Technologies*, pages 253–284. Springer, 2011.

[San18] Francisco Santos. Prime : Probabilistic membership – large scale membership and consistency. 2018.

[Sch90] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[SMG$^+$18] Lalith Suresh, Dahlia Malkhi, Parikshit Gopalan, Ivan Porto Carreiro, and Zeeshan Lokhandwala. Stable and consistent membership at scale with rapid. In *2018 USENIX Annual Technical Conference USENIX ATC 18*, pages 387–400. ACM, 2018.

[SMK$^+$01] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[VGVS05] Spyros Voulgaris, Daniela Gavidia, and Maarten Van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and systems Management*, 13(2):197–217, 2005.

[VRDGT08] Robbert Van Renesse, Dan Dumitriu, Valient Gough, and Chris Thomas. Efficient reconciliation and flow control for anti-entropy protocols. In *proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, page 6. Citeseer, 2008.

[VRS04] Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, 2004.

[ZHS$^+$04] Ben Y Zhao, Ling Huang, Jeremy Stribling, Sean C Rhea, Anthony D Joseph, and John D Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on selected areas in communications*, 22(1):41–53, 2004.