CATS: Linearizability and Partition Tolerance in Scalable and Self-Organizing Key-Value Stores

Cosmin Arad, Tallat M. Shafaat, and Seif Haridi
{cosmin,tallat,seif}@sics.se
May 20, 2012

SICS Technical Report T2012:04
ISSN 1100-3154

Abstract. Distributed key-value stores provide scalable, fault-tolerant, and self-organizing storage services, but fall short of guaranteeing linearizable consistency in partially synchronous, lossy, partitionable, and dynamic networks, when data is distributed and replicated automatically by the principle of consistent hashing. This paper introduces *consistent quorums* as a solution for achieving atomic consistency. We present the design and implementation of CATS, a distributed key-value store which uses consistent quorums to guarantee linearizability and partition tolerance in such adverse and dynamic network conditions. CATS is *scalable*, *elastic*, and *self-organizing*; key properties for modern cloud storage middleware. Our system shows that consistency can be achieved with practical performance and modest throughput overhead (5%) for read-intensive workloads.

Keywords: atomic consistency, partition tolerance, scalability, self-organization, elasticity, fault tolerance, dynamic reconfiguration, distributed key-value stores.

1 Introduction

Modern web-scale applications generate and access massive amounts of semi-structured data at very high rates. To cope with such demands, the underlying storage infrastructure supporting these applications and services, must be extremely *scalable*. The need for scalability, high availability, and high performance motivated service operators to design custom storage systems [1–6] that *replicate* data and *distribute* it over a large number of machines in a data center distributed system.

Due to the semi-structured nature of the data, such systems often have a simple API for accessing data in terms of a few basic operations: *put(key, value)*, *delete(key)*, and *value = get(key)*, and hence they are referred to as *key-value stores*. The number of replicas accessed by *put* and *get* operations determines the level of data *consistency* provided by the system [7]. To achieve strong data consistency, whereby clients have the illusion of a single storage server, *put* and *get* operations need to access overlapping quorums of replicas [8]. Typically, the more servers an operation needs to wait for, the higher its latency [9]. Early designs of key-value stores [1, 2] catered for applications that did not require strong data consistency, and driven by the need for low latency and availability, they chose to provide only *eventual consistency* for *put* and *get* operations.

Eventual consistency means that for a given key, data values may diverge at different replicas, e.g., as a result of operations accessing less than a quorum of replicas or due to network partitions [10,11]. Eventually, when the application detects conflicting replicas, it needs to reconcile the conflict. This can be done automatically for data types with

monotonic update operations [12]. In general however, conflict detection and resolution increases application complexity, both syntactically, by cluttering its logic with extra code paths, and semantically, by requiring programmers to devise reconciliation logic for all potential conflicts.

There is a significant class of applications that cannot rely on an eventually consistent data store. In particular, financial and electronic health record applications, services managing critical meta-data for large cloud infrastructures [13, 14], or more generally, systems in which the results of data-access operations have external side-effects, all need a data store with strong consistency guarantees in order to operate *correctly* and *securely*. The strongest level of consistency for *put* and *get* operations, is called *atomic consistency* or *linearizability* [15] and informally, it guarantees that for every key, a *get* returns the value of the last completed *put* or the value of a concurrent *put*, and once a *get* returns a value, no subsequent *get* can return an older, stale value.

When scalable systems grow to really large number of servers their management effort increases significantly. Thus, *self-organization* and self-healing are commendable properties of modern scalable data stores [16]. Many existing key-value stores [1–4] rely on *consistent hashing* [17] for automatic data management when servers join and leave the system, or fail. Moreover, with consistent hashing all servers are symmetric. No master server means no scalability bottleneck and no single point of failure.

Scaling to a very large number of servers also increases the likelihood of *network partitions* and *inaccurate failure suspicions* [18] caused by network congestion or by the failure or misconfiguration of network equipment. For the class of critical applications mentioned above, it is imperative that consistency is maintained during these adverse network conditions, even at the expense of service availability [10, 11].

The complexities of eventual consistency and the need for atomic consistency motivated us to explore how linearizability can be achieved in scalable key-value stores based on consistent hashing [1–4]. The problem is that simply applying quorum-based *put*/*get* operations [19] within replication groups dictated by consistent hashing [20], fails to satisfy linearizability in partially synchronous, lossy, and partitionable networks with dynamic node membership. We show the pitfalls of a naïve approach and describe the challenge of achieving linearizability in Section 3. Section 4 introduces *consistent quorums* as a solution. In Section 5 we present our system's architecture and in Section 6 its performance evaluation. Section 7 surveys related work and Section 8 concludes.

Contributions. In this paper we make the following contributions:

- We introduce *consistent quorums* as an approach to guarantee linearizability in a decentralized, self-organizing, dynamic system spontaneously reconfigured by consistent hashing, and prone to inaccurate failure suspicions and network partitions.
- We showcase consistent quorums in the design and implementation of CATS, a distributed key-value store where every data item is an atomic register with linearizable *put*/*get* operations and a dynamically reconfigurable replication group.
- We evaluate the cost of consistent quorums and the cost of achieving atomic data consistency in CATS. We give evidence that consistent quorums admit (a) system designs which are scalable, elastic, self-organizing, fault-tolerant, consistent, and partition-tolerant, as well as (b) system implementations with practical performance and modest throughput overhead (5%) for read-intensive workloads.

2 Background

In this paper we leverage the research work on (a) consistent hashing, which has been used for building scalable yet weakly-consistent distributed key-value stores [1–4], and (b) quorum-based replication systems which are linearizable but not scalable [19].

Consistent Hashing. Consistent hashing [17] is a technique for balanced partitioning of data among nodes, such that adding and removing nodes requires minimum repartitioning of data. Consistent hashing employs an identifier space perceived as a ring. Both data items and nodes are mapped to identifiers in this space. Many distributed hash tables (DHTs), such as Chord [20] and Pastry [21], were built using consistent hashing.

Each node in the system maintains a *succ* pointer to its successor on the consistent hashing ring. The successor of a node n is the first node met going in the clockwise direction on the identifier ring, starting at n. Similarly, each node keeps a *pred* pointer to its predecessor. The predecessor of n is the first node met going anti-clockwise on the ring, starting at n. A node n is *responsible* for storing all key-value pairs for which the key identifier belongs to $(p.pred, p]$. For fault tolerance on the routing level, each node n maintains a *successor-list*, consisting of n's c immediate successors. For fault tolerance on the data level, all key-value pairs stored on n are replicated on the first $r-1$ nodes in n's successor-list, where r is the replication degree. A periodic stabilization algorithm was proposed in Chord [20] to maintain the ring pointers under node dynamism.

Linearizability. For a replicated storage service, linearizability provides the illusion of a single storage server: each operation applied at concurrent replicas appears to takes effect instantaneously at some point between its invocation and its response [15]; as such, linearizability is sometimes called atomic consistency. In the context of a key-value store, linearizability guarantees that for every key, a *get* always returns the value updated by the most recent *put*, never a stale value, thus giving the appearance of a global consistent memory. Linearizability is composable [15]. In a key-value store, this means that if operations on each individual key-value pair are linearizable, then all operations on the whole key-value store are linearizable.

Quorum-Based Replication Systems. For a static set of nodes replicating a data item, Attiya et al. [19] showed how a shared memory register abstraction can be implemented in a fully asynchronous message-passing system while satisfying linearizability. In their protocol, also known as ABD, each operation is applied on a majority quorum of nodes, such that the quorums for all operations always intersect in at least one node.

3 Problem Statement

A general technique used for replication with consistent hashing is *successor-list* replication [20], whereby every key-value data item is replicated at a number of servers that succeed the responsible node on the consistent hashing ring. An example is shown in Figure 1 on the left. Here, the replication degree is three and a quorum is a majority, i.e., any set of two nodes from the replication group. A naïve attempt of achieving linearizable consistency is to use a shared memory register approach, e.g. ABD [19], within every replication group. This will not work as false failure suspicions, along with consistent hashing, may lead to non-intersecting quorums. The right diagram in Figure 1 depicts such a case, where node 15 falsely suspects node 10. According to node 10,

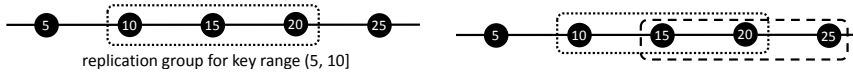

Fig. 1. Replication groups for keys in range $(5, 10]$ using consistent hashing with successor-list replication. Replication degree is three and a quorum is any set of two in the replication group.

the replication group for the keys in the range (5, 10] is {10, 15, 20}, while from node 15's perspective, the replication group for the same keys is {15, 20, 25}. Now, two different operations on key 8 may reach non-intersecting quorums leading to a violation of linearizability. For example, a *put* operation may complete after updating the value associated with key 8 at replicas 10 and 20. A subsequent *get* operation may reach replicas 15 and 25 and return a stale value despite contacting a majority of replicas.

In a key-value store with replication groups spontaneously reconfigured by consistent hashing, applying *put* and *get* operations on majority quorums is not sufficient for achieving linearizable consistency. We propose *consistent quorums* as a solution.

4 Consistent Quorums

Quorum-based protocols are typically modeled on the following pattern. An operation coordinator sends request messages to a set of participants and waits for responses or acknowledgments. Upon receiving a request message, each participant acts on the request and responds to the coordinator with an acknowledgment. The coordinator completes the operation as soon as it receives a quorum [8] of acknowledgments. Typically, essential *safety* properties of the protocol are satisfied by ensuring that the quorums for different operations intersect in at least one participant.

Quorum-intersection is easily achieved in a static system with a fixed set of nodes. In a dynamic system however, different nodes may have inconsistent views of the group membership. It is possible thus, that the number of nodes which consider themselves responsible for a key range, i.e., the number of nodes in a replication group, is larger than the replication degree. As a result, successive *put* and *get* operations may complete by contacting non-overlapping quorums, as we've shown in the previous section.

The idea is then to maintain a membership *view* of the replication group at each node which considers itself to be a replica for a particular key range according to the principle of consistent hashing. Each node in a replication group has a view $\langle v, i \rangle$, where v is the set of nodes in the replication group and i is the version number of the view.

Definition 1. *For a given replication group G, a **consistent quorum** is a regular quorum of nodes in G which are in the same view at the time when the quorum is assembled.*

When a node replies to a request it stamps its reply with its current view. The main idea is that a quorum-based operation will succeed only if it finds a quorum of nodes with the same view, i.e., a consistent quorum.

As node membership changes over time, we need a mechanism to reconfigure the membership views consistently at all replication group members. For that we devised a group reconfiguration protocol based on Paxos consensus [22], extended with an extra view installation phase and augmented with consistent quorums. We present our group reconfiguration protocol in the following section.

4.1 Paxos-Based Reconfiguration Using Consistent Quorums

Replication groups must be *dynamically reconfigured* [23] to account for new node arrivals and to restore the replication degree after group member failures. Algorithms 1-3 illustrate our Paxos-based reconfiguration protocol using consistent quorums. Earlier we defined a consistent quorum as an extension of a regular quorum. Without loss of generality, hereafter we focus on majority-based (consistent) quorums.

A reconfiguration is proposed and overseen by a coordinator node which could be a new joining node, or an existing node that suspects one of the group members to have failed. Reconfiguration $(v \Rightarrow v')$ takes the group from the current view $\langle v, i\rangle$ to the next view $\langle v', i+1\rangle$. Group size stays constant and each reconfiguration changes the membership of a replication group by a single node. One new node joins the group to replace a node which leaves the group. The reconfiguration protocol amounts to the coordinator getting the members of the current view to agree on the next view, and then *installing* the decided next view at ever node in the current and the next views, i.e., $v \cup v'$. We say that a node is in view v, once it has installed view v and before it installs the next view, v'. Nodes install views sequentially, in the order of the view versions.

The key issue catered for by our reconfiguration protocol is maintaining the quorum-intersection property for consistent quorums during reconfigurations. To make sure no two consistent quorums may exist simultaneously for a replication group, e.g., for the current and next views of a reconfiguration, the decided next view is first installed on a majority of the old members, and thereafter it is installed on the new group member.

Under high churn, different nodes may concurrently propose conflicting next views. Using consensus ensures that the next view is agreed upon by the members of the cur-

Algorithm 1 Reconfiguration coordinator

Init: p1Acks[i] $\leftarrow \emptyset$, p2Acks[i] $\leftarrow \emptyset$, p3Acks[i] $\leftarrow \emptyset$
prop[i] $\leftarrow 0$, rec[i] $\leftarrow \perp$ ▷ $\forall$ *consensus instance* i

1: **on** ⟨Propose: $(v \Rightarrow v')$⟩ **do**
2: rec[i] $\leftarrow (v \Rightarrow v')$ ▷ *proposed reconfiguration*
3: i $\leftarrow v$
4: **send** ⟨P1A: i, prop[i]⟩ **to** *all members of* v

5: **on** ⟨P1B: i, ACK, ts, rec, v⟩ **do**
6: p1Acks[i] $\leftarrow$ p1Acks[i] $\cup \{(ts, rec, v)\}$
7: **if** v = *consistentQuorum*(p1Acks[i]) **then**
8: r $\leftarrow$ *highestProposedReconfig*(p1Acks[i])
9: **if** r $\neq \perp$ **then** rec[i] $\leftarrow$ r
10: **send** ⟨P2A: i, prop[i], rec[i]⟩ **to** *all nodes in* v

11: **on** ⟨P1B: i, NACK, v⟩ $\vee$ ⟨P2B: i, NACK, v⟩ **do**
12: prop[i]++ ▷ *retry with higher proposal number*
13: **send** ⟨P1A: i, prop[i]⟩ **to** *all members of* i

14: **on** ⟨P2B: i, ACK, v⟩ **do**
15: p2Acks[i] $\leftarrow$ p2Acks[i] $\cup \{v\}$
16: **if** v = *consistentQuorum*(p2Acks[i]) **then**
17: **send** ⟨P3A: i, rec[i]⟩ **to** *all members of* v

18: **on** ⟨P3B: i, v⟩ **do**
19: p3Acks[i] $\leftarrow$ p3Acks[i] $\cup \{v\}$
20: **if** v = *consistentQuorum*(p3Acks[i]) **then**
21: **send** ⟨P4A: v', rec[i]⟩ **to** *new group member*

Algorithm 2 Current group member

1: **on** ⟨P1A: i, p⟩ **do** ▷ *acceptor*
2: **if** $p \geq$ rts[i] $\wedge$ $p \geq$ wts[i] **then**
3: rts[i] $\leftarrow p$ ▷ *promised*
4: **reply** ⟨P1B: i, ACK, wts[i], rec[i], view⟩
5: **else reply** ⟨P1B: i, NACK, view⟩

6: **on** ⟨P2A: $i, p, (v \Rightarrow v')$⟩ **do** ▷ *acceptor*
7: **if** $p >$ rts[i] $\wedge$ $p >$ wts[i] **then**
8: wts[i] $\leftarrow p$
9: rec[i] $\leftarrow (v \Rightarrow v')$ ▷ *accepted*
10: **reply** ⟨P2B: i, ACK, view⟩
11: **else reply** ⟨P2B: i, NACK, view⟩

12: **on** ⟨P3A: $i, (v \Rightarrow v')$⟩ **do** ▷ *learner*
13: *installView*(v')
14: **reply** ⟨P3B: i, view⟩
15: **send** ⟨Data: v', data⟩ **to** *new member* $(v' \setminus v)$

Algorithm 3 New group member

1: **on** ⟨P4A: v', rec⟩ **do**
2: *installView*(v') ▷ *makes* v' *busy*

3: **on** ⟨Data: $v', data$⟩ **do** ▷ *from old members*
4: dataSet[v'] $\leftarrow$ dataSet [v'] $\cup \{data\}$
5: **reply** ⟨DataAck: v'⟩
6: **if** G = *consistentQuorum*(dataSet) **then**
7: *storeHighestItems*(dataSet) ▷ *makes* v' *ready*

rent view, and the group reconfiguration proceeds safely. When a reconfiguration proposer p notices that the decided next view v' is different from the one it had proposed (v''), p assesses whether its proposed reconfiguration is still needed. This may be the case, for example, when v' still contains a node which p suspects to have failed. In such a scenario, p generates a new reconfiguration to reflect the new view, and then proposes it in the new protocol instance determined by v'.

In the algorithm specifications we omit the details pertaining to ignoring orphan messages or breaking ties between proposal numbers based on the proposer id. The *consistentQuorum* function tests whether a consistent quorum exists among the received view-stamped acknowledgments and if so, it returns the view of the consistent quorum.

Phases 1 and 2 of the protocol are just the two phases of Paxos augmented with consistent quorums. Phase 3 is the view installation phase. Once the next view v' is decided, the coordinator asks (P3A) the members of the current view v to install v'. Once v' is installed at a majority of nodes in v, only a minority of nodes are still in view v, and so it is safe to install v' at the new member, without allowing two simultaneous majorities (one for v and one for v'). When a member of v installs v' it also sends the data to the new member of v'. Conceptually, once the new member receives the data from a majority of nodes in the old view, it stores the data items with the highest timestamp from a majority. In practice however, we optimize the data transfer such that only keys and timestamps are pushed from all nodes in v to the new node, which then pulls the latest data items in parallel. Copying the latest items in a majority is important for satisfying linearizability for the *put*/*get* operations that occur during reconfiguration.

It is possible that multiple subsequent reconfigurations progress with a majority of nodes while the nodes in a minority do not receive any P3A messages. Later, the minority nodes may receive P3A messages in the wrong order. When a node n is instructed to apply a reconfiguration $(v \Rightarrow v')$ whereby n is a member of both v and v', but n is not yet in view v, n stores the reconfiguration in an *install queue* and applies it in the future, immediately after installing view v. View installation acknowledgment (P3B) and data transfer for v' only occur after v' is installed. It is also possible that after installing a view v', a new group member n subsequently installs newer views v'', v''', etc., before receiving all data for v'. In such cases, n stores the newer views in a *data chain* and upon receiving the data for v' it transfers it to the new members of v'', v''', etc.

We say that a reconfiguration *terminates* when a majority of nodes in v' have installed v'. To satisfy termination in the face of message loss and network partitioning (even when a majority of v fails after phase 3), the members of v will periodically keep trying to get an acknowledgment (DataAck) that the new node received all the data.

4.2 Linearizable *Put* and *Get* Operations Using Consistent Quorums

We adapted the ABD [19] algorithm, which implements an atomic register in a static asynchronous system, to work with consistent quorums in a dynamic replication group. Algorithms 4-6 illustrate our adaptation which provides linearizable *put* and *get* operations even during group reconfigurations. Any node in the system that receives a *put* or *get* operation request from a client, will act as an operation coordinator. The operation coordinator first locates the replication group for the requested key, and then engages in a two-phase quorum-based interaction with the group members.

Algorithm 4 Operation coordinator (part 1)

```
Init: rAcks[k] ← ∅, wAcks[k] ← ∅, G[k] ← ∅
      reading[k] ← false                          ▷ ∀ key k
 1: on ⟨GetRequest: k⟩ do                         ▷ from client
 2:   reading[k] ← true
 3:   send ⟨ReadA: k⟩ to all replicas of k
 4: on ⟨PutRequest: k, val⟩ do                    ▷ from client
 5:   val[k] ← val
 6:   send ⟨ReadA: k⟩ to all replicas of k
 7: on ⟨ReadB: k, ts, val, view⟩ do
 8:   rAcks[k] ← rAcks[k] ∪ {(ts, val, view)}
 9:   if G[k] ← consistentQuorum(rAcks[k]) then
10:     (t, v) ← highestTimestampValue(rAcks[k])
11:     if reading[k] then
12:       val[k] ← v
13:       send ⟨WriteA: k, t, val[k], G[k]⟩ to G[k]
14:     else
15:       send ⟨WriteA: k, t+1, val[k], G[k]⟩ to G[k]
```

Algorithm 5 Replication group member

```
1: on ⟨ReadA: k⟩ ∧ ready(k) do                    ▷ from coord.
2:   reply ⟨ReadB: k, version[k], value[k], view⟩
3: on ⟨WriteA: k, ts, val, G⟩ do                  ▷ from coord.
4:   if ts > version[k] ∧ G = view then
5:     value[k] ← val                             ▷ update local copy
6:     version[k] ← ts
7:   reply ⟨WriteB: k, view⟩
```

Algorithm 6 Operation coordinator (part 2)

```
1: on ⟨WriteB: k, view⟩ do
2:   wAcks[k] ← wAcks[k] ∪ {view}
3:   if G[k] = consistentQuorum(wAcks[k]) then
4:     if reading[k] then
5:       send ⟨GetResponse: k, val[k]⟩ to client
6:     else
7:       send ⟨PutResponse: k⟩ to client
8:     resetLocalState(k)                          ▷ rAcks, wAcks, G, reading
```

For a *get* operation, the coordinator reads the value with the latest timestamp from a consistent quorum. In the absence of concurrent *put* operations for the same key, the *get* operation completes in a single round since all value timestamps received in a consistent quorum are equal. (This optimization is omitted from Algorithm 4.) If the coordinator sees different value timestamps in a consistent quorum, a concurrent *put* is in progress, and since the coordinator cannot be sure that the latest value is already committed at a consistent quorum, it commits it himself (WriteA) before completing the *get*. This preserves linearizability by preventing a subsequent *get* from returning the old value.

For a *put* operation, the coordinator first reads the latest value timestamp from a consistent quorum, and then it commits the new value with a higher timestamp at a consistent quorum of the same view. We omit from the algorithms details pertaining to breaking ties between value timestamps based on coordinator id, or ignoring orphan messages. We also omit details about operation timeout and retrial, e.g., when a coordinator does not manage to assemble a consistent quorum within a given timeout or when the operation is retried because a view change occurred between the two phases. Coordinator retrial ensures operation termination. Clients retry upon coordinator failure.

When a server has just joined the replication group, i.e., it installed the view but is still waiting for the data, we say that view is *busy* (Algorithm 3 lines 2 and 7). Before the view becomes *ready* the server will not reply to ReadA messages (Algorithm 5 line 1). Transferring the latest values from a consistent quorum preserves linearizability.

4.3 Network Partitions and Inaccurate Failure Suspicions

A *network partition* is a situation where the nodes of a distributed system are split into disconnected components which cannot communicate with each other. This is a special case of message loss whereby a set of links systematically drop messages for a while, until the network partition heals. A closely related situation is that of *inaccurate failure suspicions* where due to similar network failures or congestion, some nodes may suspect other nodes to have crashed after not receiving responses from them for long

enough. We say that a distributed protocol is *partition tolerant* if it continues to satisfy its correctness properties despite these adverse network conditions.

Paxos [22] and ABD [19] are intrinsically partition tolerant; since they depend on majority quorums, operations in any partition that contains a majority quorum will succeed. To maintain their partition tolerance when applying Paxos and ABD within a consistent hashing ring, we use consistent quorums to preserve their *safety* properties. To preserve their *liveness* properties, we employ a ring unification algorithm [24] that repairs the consistent hashing ring topology after a transient network partition, hence reconciling node responsibilities dictated by consistent hashing with existing replication group views, and thus reducing the number of group reconfigurations. This makes our overall solution partition tolerant, satisfying both safety and liveness properties.

4.4 Safety

Lemma 1. *After a successful (terminated) reconfiguration* $(v \Rightarrow v')$*, at most a minority of nodes in* v *may still have* v *installed.*
Proof. If reconfiguration $(v \Rightarrow v')$ terminated, it must have completed phase 3, thus a majority of nodes in v must have installed v' (or yet a newer view). Therefore, at most a minority of nodes in v may still have v installed. □

Lemma 2. *For a particular key replication group, there cannot exist two disjoint majorities (w.r.t. view size* V*) with consistent views, at any given time.*
Proof. Case 1 (same view): no view is ever installed on more than V nodes. Therefore, there can never exist two or more disjoint majorities with the same consistent view. Case 2 (consecutive views $v \Rightarrow v'$): by the algorithm (phase 3), a majority for v' cannot exist before v' is installed at a majority of v. Once a majority of nodes in v have installed v', they now constitute a majority in v' and by Lemma 1 at most a minority of v still has v installed, thus two disjoint majorities for views v and v' cannot exist simultaneously. Case 3 (non-consecutive views $v \rightsquigarrow v''$): views are always installed in sequence. For the group to reach view v'' from v, a majority of v must have first applied a reconfiguration $(v \Rightarrow v')$. At that time, by Case 2, a consistent majority for v ceased to exist. □

Lemma 3. *For a particular key replication group, no sequence of network partitions and mergers may lead to disjoint consistent quorums.*
Proof. By the algorithm, a majority of nodes in view v must be available and connected for a reconfiguration $(v \Rightarrow v')$ to succeed. Thus, reconfigurations can only occur in partitions containing a majority of nodes, while nodes in any minority partitions are stuck in view v. Case 1: network partition splits group in multiple minority partitions so no reconfiguration can occur; when the partitions merge, by Case 1 of Lemma 2 we cannot have disjoint consistent quorums. Case 2: a sequence of network partitions and reconfigurations (in a majority partition M) results in multiple minority partitions that later merge (independently from M). Because every reconfiguration generates a new view, the views available in different minority partitions are all distinct and thus, they cannot form a consistent quorum (disjoint from a consistent quorum in M). □

From Lemmas 2 and 3, we have Theorem 1 which is sufficient for linearizability.

Theorem 1. *For any key replication group, no two disjoint consistent quorums may exist simultaneously. Therefore, any two consistent quorums always intersect.*

4.5 Liveness

Lemma 4. *Provided a consistent quorum of the current view v is accessible, a group reconfiguration $(v \Rightarrow v')$ will eventually terminate.*

Argument. Given that the reconfiguration coordinator does not crash, and a majority of v is accessible, with periodic retrials to counter for message loss, the coordinator will eventually succeed in installing v' on a majority of v'. If the coordinator crashes, another node will become coordinator, and guided by consistent hashing, it will propose new reconfigurations to reconcile the group membership with the ring membership. □

Corollary 1. *Provided that all network partitions cease, every ongoing group reconfiguration will eventually terminate.*

Argument. After all network partitions merge, even groups that had been split into multiple minority partitions are now merged, thus satisfying the premise of Lemma 4. □

Lemma 5. *Provided a consistent quorum is accessible,* put *and* get *operations will eventually terminate.*

Argument. Given that an operation's coordinator does not crash before the operation completes, it will periodically retry to assemble a consistent quorum for the operation's key, until one becomes available and connected. When a client detects that the coordinator crashed, the client retries its operation with a different coordinator. □

From Lemmas 4 and 5, and Corollary 1, we have Theorem 2 regarding termination.

Theorem 2. *For any key replication group, provided a consistent quorum is available and connected, any* put *and* get *operations issued in the same partition, and any reconfigurations will eventually terminate. Moreover, if the network is fully connected, all operations and reconfigurations will eventually terminate.*

4.6 Simulation-Based Correctness Tests

We implemented these algorithms in the CATS system using the Kompics message-passing component framework [25]. Kompics allowed us to execute the system in deterministic simulation mode. We devised a wide range of experiment scenarios comprising concurrent reconfigurations and failures, and used an exponential message latency distribution with a mean of 89ms. We verified stochastically that our algorithms satisfied their safety invariants and liveness properties, in all scenarios for 1 million RNG seeds.

5 System Architecture

To validate and evaluate the idea of consistent quorums, we have designed and built the CATS system, a scalable and self-organizing key-value store which leverages consistent quorums to provide linearizable consistency and partition tolerance. CATS was implemented (in Java) using the Kompics message-passing component framework [25], which allows the system to readily leverage multi-core hardware by executing concurrent components in parallel on different cores. We now describe the high-level system architecture of CATS. Figure 2 illustrates the main protocol components of a single

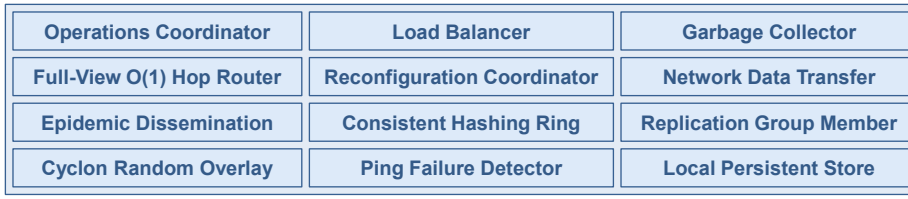

Fig. 2. Overview of the protocol components architecture at a single CATS server.

CATS server. In addition to this, a client library, linked with application clients, is in charge of locating servers and relaying *put* and *get* operations from the application.

One fundamental building block for CATS is the **Consistent Hashing Ring** module. It subsumes a *periodic stabilization* (PS) protocol [20] for maintaining the ring pointers under node dynamism, as dictated by consistent hashing. Since PS does not cater for network partitions and mergers, it is possible that during a transient network partition, PS reorganizes the ring into two disjoint rings. We use a *ring unification* (RU) protocol [24] to repair pointers and converge to a single ring after a network partition. Thus, CATS's Consistent Hashing Ring overlay is partition tolerant. Both PS and RU are *best-effort* protocols: they do not guarantee lookup consistency [26] and may lead to non-overlapping quorums (Section 3). The Consistent Hashing Ring module relies on a **Ping Failure Detector** component to monitor its ring neighbors. The failure detector is *unreliable* [18] and it can inaccurately suspect monitored nodes to have crashed.

Another crucial component of CATS is the **Cyclon Random Overlay**. This module encapsulates the Cyclon gossip-based membership protocol [27]. Cyclon implements a *peer sampling service* which provides every server with a continuous stream of random nodes in the system. We use this stream to build a full membership view of the system in the **Full-View O(1) Hop Router** component. This enables an **Operation Coordinator** to very efficiently look up (in $O(1)$ hops [28]) the responsible replicas for a given key-value pair. The view at each node is not required to immediately reflect changes in node membership and so, the view can be stale for short periods, for large system sizes. A membership change detected in a local neighborhood is propagated to the rest of the system by the **Epidemic Dissemination** module. This module also relies on the random peer sampling service to quickly and robustly broadcast churn events to all servers [29].

The **Replication Group Member** module handles consistent group membership views and view reconfigurations (acting as an *acceptor* and *learner* in Algorithms 2 and 3). View reconfigurations are *proposed* (Algorithm 1) by the **Reconfiguration Coordinator** component, which monitors the state of the Consistent Hashing Ring and tries to reconcile the replication group membership with the ring membership. To avoid multiple nodes proposing the same reconfiguration operation, we employ a *selfish* mechanism, whereby only the node responsible for a key-range replication group (according to consistent hashing) proposes a reconfiguration in this group. Apart from averting multiple nodes from proposing the same reconfiguration, this mechanism has an added benefit. In consistent hashing, there is always at least one node responsible for each key range, and this node will keep attempting to repair the replication group for that key range. Periodic retrials will make sure that replication group reconfigurations will eventually terminate for all key ranges, despite message loss and transient network partitions.

The **Network Data Transfer** component implements optimizations for fetching data to the new node joining a replication group after a view reconfiguration. The new node transfers data in parallel from existing replicas, by splitting the requested data among all replicas. This results in better bandwidth utilization, fast data transfer due to parallel downloads, and it avoids loading a single replica. If a replica fails during data transfer, the requesting node reassigns the requests sent to the failed node, to the remaining alive replicas. Before transferring values, each replica first transfers keys and their timestamps to the new node. Based on the timestamps, the new node retrieves the latest value from the node with the highest time stamp for each key. This avoids unnecessary transfers of values from existing replicas to the new replica, thus lowering bandwidth usage.

The Replication Group Member module also handles operation requests coming from an Operation Coordinator (Algorithms 4 and 6), hence acting as a replica storage server in Algorithm 5. In serving operation requests, it relies on a local key-value storage interface provided by the **Local Persistent Store** module. Currently, we have three different implementations for the Local Persistent Store module. The first is based on the Java Edition of BerkeleyDB (SleepyCat) [30], the second leverages LevelDB [31], and the third uses an in-memory sorted map. In this paper we evaluated the in-memory implementation. The persistent stores are used to implement individual-node and system-wide recovery protocols. Crash-recovery, while using persistent storage, is very similar to the node being partitioned away for a while. In both cases, when a node recovers or the partition heals, the node has the same configuration and data as it had before the failure/partition. Hence, our algorithms already support crash-recovery since they are partition tolerant. System-wide coordinated shutdown and recovery protocols are very important in cloud environments. They constitute the subject of work in progress and we omit their details here as they are outside the scope of this paper.

Systems built on consistent hashing can achieve load balancing by employing the concept of *virtual nodes* as in Dynamo [1], Riak [3], or Voldemort [4]. Each physical node joins the system as multiple virtual nodes using different identifiers. The number of virtual nodes hosted by a physical node, and the placement of each virtual node largely addresses any load-imbalance. In CATS, the **Load Balancer** module handles load balancing. It relies on the Epidemic Dissemination protocol to aggregate statistics about the load at different nodes in the system. These statistics are then used to make load balancing decisions, such as moving virtual nodes on the identifier space, and creating or removing virtual nodes. Load balancing enables us to support *range queries* in the Operation Coordinator, by allowing the keys to be stored in the system without hashing, and removing the load imbalances arising from this. Load balancing and range queries in CATS are the subject of work in progress.

The **Garbage Collector** module implements a periodic mechanism of garbage collection, to avoid unnecessary copies of data lingering around in the system as a result of transient network partitions. For example, if a replica n of group G gets partitioned away, G may still have a consistent quorum in a majority partition. Hence, G can be reconfigured, and thus evolve into subsequent new groups. After the partition ceases, the data values stored at node n are stale, and so, they can be considered garbage. Garbage collection runs periodically and it makes sure to remove only data for views that were already reconfigured and for which node n is no longer a member of.

6 Experimental Evaluation

In this section, we evaluate the performance, in terms of throughput and operation latencies, as well as the scalability and elasticity of our implementation of the CATS system. Furthermore, we evaluate the performance overhead of achieving consistency, and we perform a comparison with Cassandra 1.1.0, a system with an architecture similar to CATS. We ran our experiments on the Rackspace cloud infrastructure, using 16 GB RAM server instances. We used the YCSB [32] benchmark as a load generator for our experiments. We evaluated two workloads with uniform distribution of keys; a read-intensive workload comprising of 95% reads and 5% updates, and an update-intensive workload comprising of 50% reads and 50% updates. We chose to perform updates instead of inserts in the workload to keep the data set constant throughout the experiment. Such a choice does not have side-effects since the protocol for an update operation is the same as the one for an insert operation. Unless otherwise specified, we used data values of size 1 KB. We placed the servers at equal distance on the consistent hashing ring to avoid being side-tracked by load-balancing issues.

6.1 Performance

In the first set of experiments, we measured the performance of CATS in terms of the average latency per operation and the throughput of the system. We increased the load, i.e. the dataset size and the operations request rate, proportionally to the number of servers, by increasing the number of keys initially inserted into CATS, and the number of YCSB clients, respectively. For instance, we load 300 thousand keys and use 1 client for generating requests for 3 servers, 600 thousand keys and 2 clients for 6 servers, and so on. For each system size, we varied the request load by varying the number of threads in the YCSB clients. For low number of client threads, the request rate is low and thus the servers may be under-utilized, while a high number of client threads can overload the servers. We started with 4 threads, and doubled the thread count each time for the next experiment until 128 threads.

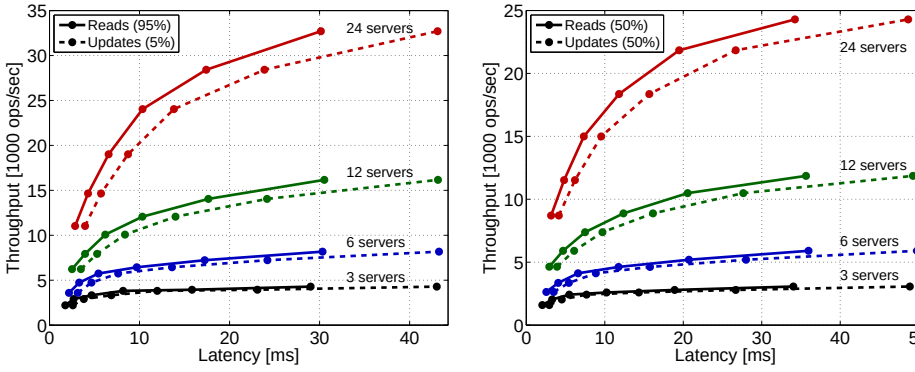

Fig. 3. Performance for a read-intensive (left) and an update-intensive (right) workload.

Figure 3 shows the results, averaged over three runs, for various number of servers. For each server count, as the request load increases, the throughput also increases till a certain value after which the only latency increases without an increase in throughput. Such a state depicts that the system is saturated and cannot offer more throughput. In other words, when the system is underloaded (few client threads), the latencies are low yet server resources are not fully utilized. As the request rate is increased by increasing the number of client threads, the latency and throughput also increase until a certain throughput is offered. For instance, for 3 servers and a read-intensive workload, the system saturates at approximately 4000 operations$/$sec with an average latency of 8 milliseconds (and 32 YCSB client threads). Further increasing the request rate does not increase the throughput, while the latency keeps increasing. This depicts an overloaded system, where the current number of servers cannot serve all incoming requests, leading to queueing effects. This behavior is same for both workloads.

6.2 Scalability

In our next experiments, we evaluated the scalability of CATS. We increased the dataset size and requests rate proportionally to the number of servers, as we did in the performance experiments. Figure 4 shows the throughput of the system as we vary the number of servers for both workloads. The figure shows that CATS scales linearly with a slope of one. With small number of servers, it is more likely that requests already arrive at one of the replicas for the requested key. Thus, the number of messages sent over the network is smaller. This explains the slightly higher throughput for 3 and 6 servers. The reason for linear scaling is that CATS is completely decentralized and all nodes are symmetric. Owing to linear scalability, the number of servers needed to achieve a certain throughput or to handle a certain rate of requests, can be calculated easily when deploying CATS in a cloud environment, provided the load is balanced across the servers. Such a decision can be made actively by either an administrator, or a feedback control loop that monitors the rate of client requests.

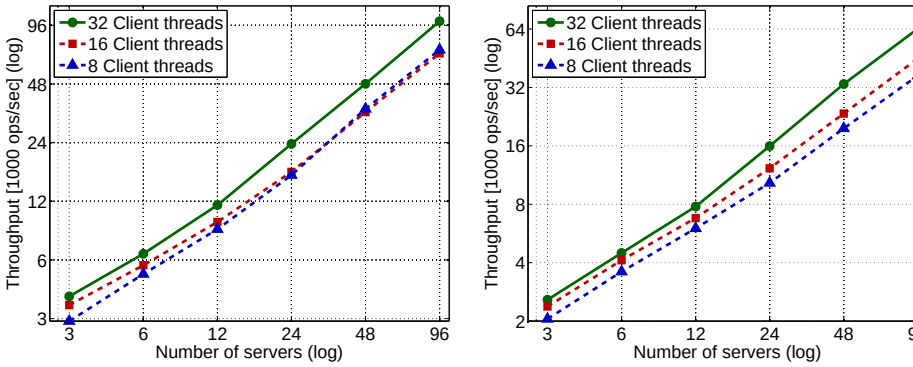

Fig. 4. Scalability for a read-intensive (95%, left) and an update-intensive (50%, right) workload.

6.3 Elasticity

A highly desirable property for systems running in cloud environments is *elasticity*, the ability to add or remove servers while the system is running. When a system is overloaded, i.e. latency per operation is so high that it violates an service-level agreement (SLA), the performance can be improved by adding new servers. Similarly, when the load is very low, one can reduce running costs by decreasing the number of servers without violating any SLA. A system with good elasticity should perform better as servers are added, with a short disruption while the system reconfigures to include the new servers. The length of the disruption depends on the amount of data that needs to be transfered for the reconfiguration. A well-behaved system should have low latencies during this disruption window so that the end clients are not affected. In this experiment, we evaluated the elasticity of CATS. We started the system with 3 servers, loaded 2.4 million 1 KB values, and injected a high operation request rate via the YCSB client. While the workload was running and keeping the request rate constant, we added a new server every 10 minutes until the server count doubled to 6 servers. Afterwards, we started to remove servers every 10 minutes until we were back to 3 servers. We measured the average of operation latencies in 1 minute intervals throughout the experiment. The results of our experiment are presented in Figure 5. These results show that CATS incorporates changes in the number of servers with short windows (1–2 minutes) of disruption when the reconfiguration occurs, while the average latencies remain bounded by $2 \times x$ where x is the latency before the reconfiguration was triggered. Furthermore, since CATS is scalable, the latency approximately halves when the number of servers doubles to 6 between 30–50 minutes compared to 3 servers between 0–10 minutes. As nodes are removed after 50 minutes, the latency starts increasing as expected.

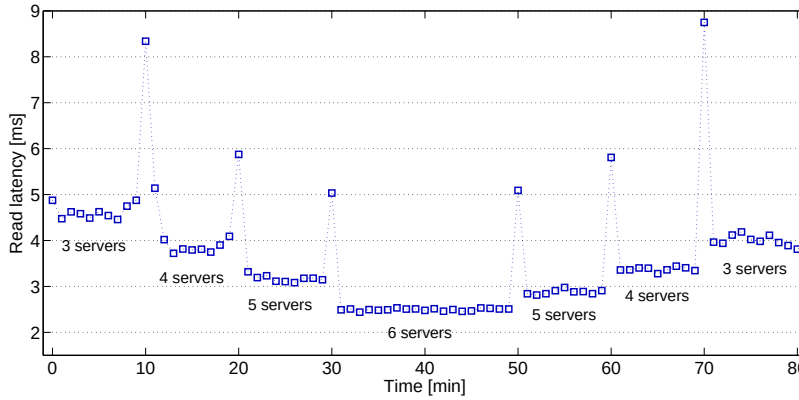

Fig. 5. Elasticity for a read-only workload.

6.4 Overhead of Atomic Consistency and Consistent Quorums

Next, we evaluated the overhead of atomic consistency compared to eventual consistency. For a fair comparison, we implemented eventual consistency in CATS, enabled through a configuration parameter. Here, read and write operations are always performed in one phase, and read-impose (read-repair in Cassandra terms) is never used.

When a node n performs a read operation, it sends read requests to all replicas. Each replica replies with a timestamp and value. After n receives replies from a majority of replicas, it returns the value with the highest timestamp as a result of the read operation. Similarly, when a node m performs a write operation, it sends write requests to all replicas, using the current wall clock time as a timestamp. Upon receiving such a request, a replica stores the value and timestamp only if the received timestamp is higher than the replica's local timestamp. The replica then sends an acknowledgment to the writer m. Node m considers the write operation complete upon receiving acknowledgments from a majority of the replicas.

We also measured the overhead of consistent quorums. For these measurements, we modified CATS such that nodes did not send the replication group view in the read and write messages. Removing the replication group from messages reduces the size of messages, and hence requires less bandwidth.

For these experiments, we varied the size of the stored values, and we measured the throughput of a system with 3 servers. Our results, averaged over five runs, are shown in Figure 6. The results show that as the value size increases, the throughput drops. This implies that the network becomes a bottleneck for larger value sizes. The same trend is observable in both workloads. Furthermore, as the value size increases, the cost of using consistent quorums becomes negligible. For instance, the loss in throughput for both workloads when using consistent quorums is less than 5% for 256 bytes values, 4% for 1KB values, and 1% for 4KB values.

Figure 6 also shows the cost of achieving atomic consistency by comparing the throughput of regular CATS with the throughput of our eventual consistency implementation. The results show that the overhead of atomic consistency is negligible for a read-intensive workload but as high as 25% for an update-intensive workload. The reason for this difference in behavior between the two workloads is that for a read-intensive workload, the second phase for reads (read-impose/read-repair) is rarely needed, since the number of concurrent writes to the same key are very low due to the large number of keys in the workload. For an update-intensive workload, due to many concurrent

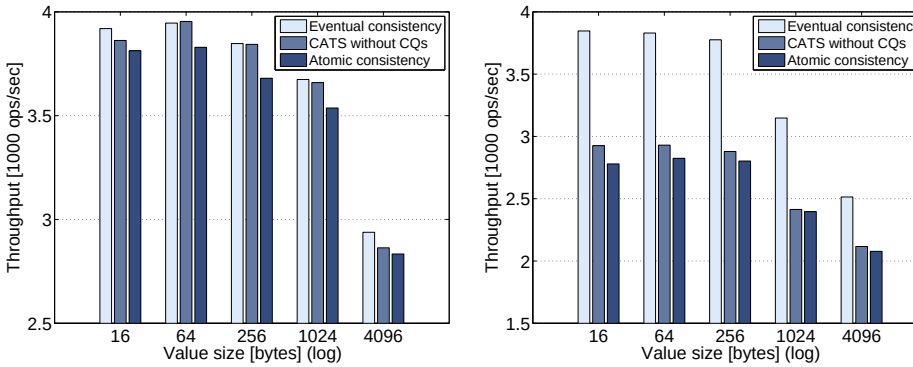

Fig. 6. Overhead of Atomic consistency and Consistent quorums for a read-intensive (95%, left) and an update-intensive (50%, right) workload.

writes, the read operations often require to impose the read value. Hence, in comparison to an update-intensive workload, the overhead of achieving linearizability is very low (less than 5% loss in throughput for all value sizes) for a read-intensive workload. We believe that this is an important result. Applications that are read-intensive can opt for atomic consistency without a significant loss in performance, while avoiding the complexities of using eventual consistency.

6.5 Comparison with Cassandra

The architecture of Cassandra [2] and Dynamo [1] are closest to CATS as both use consistent hashing with successor-list replication. Since Cassandra is available openly, here we compare the performance of CATS with that of Cassandra. We are comparing our research system with a system that leverages half a decade of implementation optimizations and fine tuning. Our aim is to give readers an idea about the relative performance difference, which, taken together with our evaluation of the cost of consistency, may give an insight into the cost of atomic consistency if implemented in Cassandra. We leave the actual implementation of consistent quorums in Cassandra to future work.

We used Cassandra 1.1.0 for our experiments, and used the `QUORUM` consistency level for a fair comparison with CATS. We chose the initial data size such that the working set would fit in main memory. Furthermore, since CATS only stores data in main memory while Cassandra uses disk, we set `commitlog_sync: periodic` in Cassandra for a fair comparison to minimize the effects to disk activity on operation latencies. Figure 7 shows a comparison of average latencies, averaged over five runs, for the same workloads for Cassandra and Eventual consistency implemented in CATS. The trend of higher latencies for large value sizes remains the same for both systems and workloads as the network starts to become a bottleneck. For CATS, read and write latencies are the same since both involve the same message complexity (one phase) and the same message sizes. On the other hand, Cassandra writes are faster than reads, which is a known fact since writes require no reads or seeks, while reads may

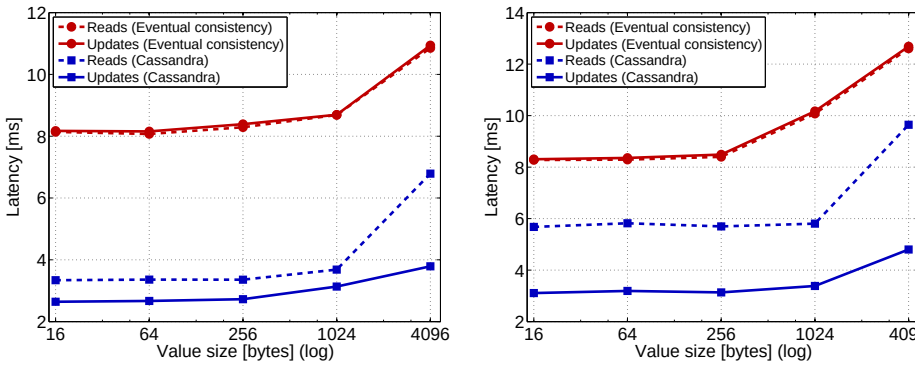

Fig. 7. Comparison of latencies for Cassandra and CATS with Eventual Consistency for a read-intensive (95%, left) and an update-intensive (50%, right) workload.

need to read multiple SSTables[1]. The results show that the operation latencies in CATS are approximately three times higher than in Cassandra (except reads in an update-intensive workload, where the effects of commit log disk accesses affect Cassandra's performance).

Given our comparison between Cassandra and Eventual consistency in CATS, and the low loss in throughput for achieving atomic consistency compared to eventual consistency (Section 6.4), we believe that an implementation of consistent quorums in Cassandra can provide linearizable consistency without considerable loss in performance (e.g. less than 5% loss for a read-intensive workload).

7 Related Work

Eventually-Consistent Key-Value Stores. Distributed key-value stores, such as Cassandra [2] and Dynamo [1], employ principles from DHTs to build scalable and self-managing data stores. In contrast to CATS, these systems chose availability over atomic consistency, hence only providing eventual consistency. While eventual consistency is sufficient for some applications, the complexities of merging divergent replicas can be non-trivial. Furthermore, we show through evaluation that the overhead of atomic consistency is very low for read-intensive workloads.

Reconfigurable Replication Systems. To handle dynamic environments, atomic registers were extended by protocols such as RDS [33] and DynaStore [34] to be reconfigurable. Similarly, SMART [35] enabled reconfiguration in replicated state machines. While these systems can handle dynamism and provide atomic consistency, they are not scalable as they cannot partition the data across a large number of machines.

Consistent Meta-Data Stores. Data-center systems providing distributed coordination and consistent meta-data storage services, such as Chubby [13] and ZooKeeper [14, 36], provide linearizability and support crash-recovery, but are neither scalable, nor reconfigurable. We believe that the idea of consistent quorums applied to consistent hashing rings can be used to scale such meta-data stores to larger capacities.

Master-based key-value stores, such as Bigtable [5], HBase [6], and MongoDB [37], rely on a central server for coordination and data partitioning. Similarly, Spinnaker [38] uses Zookeeper [14] for the coordation and data partition. Since these systems are centralized, their scalability is limited. In contrast, CATS is decentralized and all nodes are symmetric, allowing for unlimited scalability.

Scalable and Consistent Key-Value Stores. Similar to CATS, Scatter [39] is a scalable and consistent distributed key-value store. Scatter employs an extra subsystem and policies to decide when to reconfigure (split and merge) replication groups. While this makes Scatter flexible, it also requires a distributed transaction across three adjacent replication groups for the split and merge reconfiguration operations to succeed. In contrast, CATS has a simpler and more efficient, both in number of messages and message delays, reconfiguration protocol that does not need a distributed transaction. In CATS, each reconfiguration operation only operates on the replication group that is being reconfigured. Therefore, the period of unavailability to serve operations is much

[1] http://wiki.apache.org/cassandra/ArchitectureOverview

shorter (almost non-existent) in CATS, compared to Scatter. Furthermore, we focus on consistent-hashing at the node level, which makes our approach directly implementable in existing key-value stores like Cassandra [2]. Lastly, the unavailability of Scatter's implementation precludes a detailed comparison, e.g. in terms of data unavailability during reconfiguration, and elasticity.

Related Work on Consistency. An orthogonal approach to atomic consistency is to explore the tradeoffs between consistency and performance. For instance, PNUTS [40] introduces time-line consistency, whereas COPS [41] provides causal consistency at scale. These systems provide consistency guarantees weaker than linearizability, yet stronger guarantees than eventual consistency. While such systems perform well, the semantics of the consistency models they offer restricts the class of applications that can use these systems.

Fault-tolerant Replicated Data Management. Abbadi et al. [42] proposed a fault-tolerant protocol for replicated data management. Their solution is similar to CATS with respect to quorum-based operations and consensus-based replication group reconfigurations. In contrast to their solution, CATS uses the consistent hashing ring, which enables CATS to be self-managing and self-organizing under churn. Consistent hashing partitions the keys in a balanced manner, and the notion of responsibility in terms of which nodes are responsible for storing which key ranges is well-defined. Thus, the reconfigurations required when nodes join and fail is dictated by consistent hashing. Furthermore, owing to the routing mechanisms employed by CATS, any node can find any data item in a few hops even for very large network sizes.

Discussion: Alternatives to Operations on Majority Quorums. For some applications majority quorums may be too strict. To accommodate specific read-intensive or update-intensive workloads, they might want flexible quorum sizes for *put* and *get* operations, like *read-any-update-all* or *read-all-update-any*, despite the fault-tolerance caveats entailed. Interestingly, our ABD-based two-phase algorithm, depends on majority quorums for linearizability, however, given more flexible yet overlapping quorums, the algorithm still satisfies *sequential consistency* [43], which is still a very useful level of consistency. On a related node, the idea of primary-backup replication could be applied onto the consistent replication groups of CATS, to enable efficient primary reads.

8 Conclusions

In this paper we have shown that it is non-trivial to achieve linearizable consistency in dynamic, scalable, and self-organizing key-value stores which distribute and replicate data according to the principle of consistent hashing. We introduced consistent quorums as a solution to this problem for partially synchronous network environments prone to message loss, network partitioning, and inaccurate failure suspicions. As examples, we presented adaptations of Paxos and ABD augmented with consistent quorums.

We described the design, implementation, and evaluation of CATS, a distributed key-value store that leverages consistent quorums to provide linearizable consistency and partition tolerance. CATS is self-managing, elastic, and it exhibits unlimited linear scalability, all of which are key properties for modern cloud computing storage middleware. Our evaluation shows that it is feasible to provide linearizable consistency for

those applications that do indeed need it, e.g., with less than 5% throughput overhead for read-intensive workloads. Our research system implementation can deliver practical levels of performance, comparable with that of similar but heavily-optimized industrial systems like Cassandra. This suggests that if implemented in Cassandra, consistent quorums can deliver atomic consistency with acceptable performance overhead.

Future Work. Consistent quorums provide a consistent view of replication groups. Such consistent views can be leveraged to implement distributed multi-item transactions. CATS can be extended to support column-oriented APIs, indexing, and search.

Interactive Demonstration. The CATS key-value store is open source. An interactive demonstration of CATS is available at http://cats.sics.se/demo/.

Acknowledgments. This work was supported in part by grant 2009-4299 from the Swedish Research Council, by Ericsson AB, and by the SICS CNS Center. We thank Muhammad Ehsan ul Haque and Hamidreza Afzali for their work on persistency and recovery, range queries, and load balancing. We would also like to thank Amr El Abaddi, Jim Dowling, and Niklas Ekström for interesting discussions and feedback.

References

1. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: SOSP '07
2. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. **44** (April 2010) 35–40
3. Basho Riak: http://wiki.basho.com/Riak.html/ (2012)
4. Feinberg, A.: Project Voldemort: Reliable distributed storage. In: ICDE '11
5. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A distributed storage system for structured data. ACM Trans. Comput. Syst. **26**(2) (June 2008) 4:1–4:26
6. Apache HBase: http://hbase.apache.org/ (2012)
7. Bailis, P., Venkataraman, S., Franklin, M.J., Hellerstein, J.M., Stoica, I.: Probabilistically bounded staleness for practical partial quorums. PVLDB **5**(8) (2012) 776–787
8. Gifford, D.K.: Weighted voting for replicated data. In: SOSP '79
9. Abadi, D.J.: Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. Computer **45** (2012) 37–42
10. Brewer, E.A.: Towards robust distributed systems (abstract). In: PODC '00
11. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News **33**(2) (June 2002) 51–59
12. Alvaro, P., Conway, N., Hellerstein, J., Marczak, W.R.: Consistency Analysis in Bloom: a CALM and Collected Approach. In: CIDR'11
13. Burrows, M.: The chubby lock service for loosely-coupled distributed systems. In: OSDI '06, USENIX Association
14. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: wait-free coordination for internet-scale systems. In: USENIX ATC'10, USENIX Association
15. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3) (July 1990) 463–492
16. Stonebraker, M., Madden, S., Abadi, D.J., Harizopoulos, S., Hachem, N., Helland, P.: The end of an architectural era: (it's time for a complete rewrite). In: VLDB '07, VLDB Endowment (2007) 1150–1160

17. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D.: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In: STOC '97, New York, NY, USA, ACM (1997) 654–663
18. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM **43**(2) (March 1996) 225–267
19. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message-passing systems. J. ACM **42**(1) (January 1995) 124–142
20. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: SIGCOMM '01
21. Rowstron, A., Druschel, P.: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In: Middleware '01
22. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2) (May 1998)
23. Lamport, L., Malkhi, D., Zhou, L.: Reconfiguring a state machine. SIGACT News **41**(1) (March 2010) 63–73
24. Shafaat, T.M., Ghodsi, A., Haridi, S.: Dealing with network partitions in structured overlay networks. Peer-to-Peer Networking and Applications **2** (2009) 334–347
25. Arad, C., Dowling, J., Haridi, S.: Developing, simulating, and deploying peer-to-peer systems using the Kompics component model. In: COMSWARE '09
26. Shafaat, T.M., Moser, M., Schütt, T., Reinefeld, A., Ghodsi, A., Haridi, S.: Key-based consistency and availability in structured overlay networks. In: InfoScale '08
27. Voulgaris, S., Gavidia, D., van Steen, M.: Cyclon: Inexpensive membership management for unstructured p2p overlays. J. Network Syst. Manage. **13**(2) (2005) 197–217
28. Gupta, A., Liskov, B., Rodrigues, R.: Efficient routing for peer-to-peer overlays. In: NSDI'04
29. Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., Terry, D.: Epidemic algorithms for replicated database maintenance. In: PODC '87
30. BerkeleyDB: www.oracle.com/technology/products/berkeley-db/ (2012)
31. Ghemawat, S., Dean, J.: LevelDB. http://code.google.com/p/leveldb/ (2012)
32. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: SoCC '10
33. Chockler, G., Gilbert, S., Gramoli, V., Musial, P.M., Shvartsman, A.A.: Reconfigurable distributed storage for dynamic networks. J. Parallel Distrib. Comput. **69** (January 2009)
34. Aguilera, M.K., Keidar, I., Malkhi, D., Shraer, A.: Dynamic atomic storage without consensus. J. ACM **58** (April 2011) 7:1–7:32
35. Lorch, J.R., Adya, A., Bolosky, W.J., Chaiken, R., Douceur, J.R., Howell, J.: The smart way to migrate replicated stateful services. In: EuroSys '06
36. Junqueira, F.P., Reed, B.C., Serafini, M.: Zab: High-performance broadcast for primary-backup systems. In: DSN '11, IEEE Computer Society
37. MongoDB: http://www.mongodb.org/ (2012)
38. Rao, J., Shekita, E.J., Tata, S.: Using Paxos to build a scalable, consistent, and highly available datastore. Proc. VLDB Endow. **4** (January 2011) 243–254
39. Glendenning, L., Beschastnikh, I., Krishnamurthy, A., Anderson, T.: Scalable consistency in Scatter. In: SOSP '11
40. Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.A., Puz, N., Weaver, D., Yerneni, R.: PNUTS: Yahoo!'s hosted data serving platform. Proc. VLDB Endow. **1**(2) (August 2008) 1277–1288
41. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In: SOSP '11
42. El Abbadi, A., Skeen, D., Cristian, F.: An efficient, fault-tolerant protocol for replicated data management. In: PODS '85
43. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. **28**(9) (September 1979) 690–691