Scalable Consistency in Scatter

Lisa Glendenning Ivan Beschastnikh Arvind Krishnamurthy Thomas Anderson

Department of Computer Science & Engineering
University of Washington

ABSTRACT

Distributed storage systems often trade off strong semantics for improved scalability. This paper describes the design, implementation, and evaluation of Scatter, a scalable and consistent distributed key-value storage system. Scatter adopts the highly decentralized and self-organizing structure of scalable peer-to-peer systems, while preserving linearizable consistency even under adverse circumstances. Our prototype implementation demonstrates that even with very short node lifetimes, it is possible to build a scalable and consistent system with practical performance.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*

General Terms

Design, Reliability

Keywords

Distributed systems, consistency, scalability, fault tolerance, storage, distributed transactions, Paxos

1. INTRODUCTION

A long-standing and recurrent theme in distributed systems research is the design and implementation of efficient and fault tolerant storage systems with predictable and well-understood consistency properties. Recent efforts in peer-to-peer (P2P) storage services include Chord [36], CAN [26], Pastry [30], OpenDHT [29], OceanStore [16], and Kademlia [22]. Recent industrial efforts to provide a distributed storage abstraction across data centers include Amazon's Dynamo [10], Yahoo!'s PNUTS [8], and Google's Megastore [1] and Spanner [9] projects. Particularly with geographic distribution, whether due to using multiple data centers or a P2P resource model, the tradeoffs between efficiency and consistency are non-trivial, leading to systems that are complex to implement, complex to use, and sometimes both.

Our interest is in building a storage layer for a very large scale P2P system we are designing for hosting planetary scale social networking applications. Purchasing, installing, powering up, and maintaining a very large scale set of nodes across many geographically distributed data centers is an expensive proposition; it is only feasible on an ongoing basis for those applications that can generate revenue. In much the same way that Linux offers a free alternative to commercial operating systems for researchers and developers interested in tinkering, we ask: what is the Linux analogue with respect to cloud computing?

P2P systems provide an attractive alternative, but first generation storage layers were based on unrealistic assumptions about P2P client behavior in the wild. In practice, participating nodes have widely varying capacity and network bandwidth, connections are flaky and asymmetric rather than well-provisioned, workload hotspots are common, and churn rates are very high [27, 12]. This led to a choice for application developers: weakly consistent but scalable P2P systems like Kademlia and OpenDHT, or strongly consistent data center storage.

Our P2P storage layer, called Scatter, attempts to bridge this gap – to provide an open-source, free, yet robust alternative to data center computing, using only P2P resources. Scatter provides scalable and consistent distributed hash table key-value storage. Scatter is robust to P2P churn, heterogeneous node capacities, and flaky and irregular network behavior. (We have left robustness to malicious behavior, such as against DDoS attacks and Byzantine faults, to future work.) In keeping with our goal of building an open system, an essential requirement for Scatter is that there be no central point of control for commercial interests to exploit.

The base component of Scatter is a small, self-organizing group of nodes, each managing a range of keys, akin to a BigTable [6] tablet. A set of groups together partition the table space to provide the distributed hash table abstraction. Each group is responsible for providing consistent read/write access to its key range, and for reconfiguring as necessary to meet performance and availability goals. As nodes are added, as nodes fail, or as the workload changes for a region of keys, individual groups must merge with neighboring groups, split into multiple groups, or shift responsibil-

.

ity over parts of the key space to neighboring groups, all while maintaining consistency. A lookup overlay topology connects the Scatter groups in a ring, and groups execute distributed transactions in a decentralized fashion to modify the topology consistently and atomically.

A key insight in the design of Scatter is that the consistent group abstraction provides a stable base on which to layer the optimizations needed to maintain overall system performance and availability goals. While existing popular DHTs have difficulty maintaining consistent routing state and consistent name space partitioning in the presence of high churn, these properties are a direct consequence of Scatter's design. Further, Scatter can locally adjust the amount of replication, or mask a low capacity node, or merge/split groups if a particular Scatter group has an unusual number of weak/strong nodes, all without compromising the structural integrity of the distributed table.

Of course, some applications may tolerate weaker consistency models for application data storage [10], while other applications have stronger consistency requirements [1]. Scatter is designed to support a variety of consistency models for application key storage. Our current implementation provides linearizable storage within a given key; we support cross-group transactions for consistent updates to meta-data during group reconfiguration, but we do not attempt to linearize multi-key application transactions. These steps are left for future work; however, we believe that the Scatter group abstraction will make them straightforward to implement.

We evaluate our system in a variety of configurations, for both micro-benchmarks and for a Twitter-style application. Compared to OpenDHT, a publicly accessible open-source DHT providing distributed storage, Scatter provides equivalent performance with much better availability, consistency, and adaptability. We show that we can provide practical distributed storage even in very challenging environments. For example, if average node lifetimes are as short as three minutes, therefore triggering very frequent reconfigurations to maintain data durability, Scatter is able to maintain overall consistency and data availability, serving its reads in an average of 1.3 seconds in a typical wide area setting.

2. BACKGROUND

Scatter's design synthesizes techniques from both highly scalable systems with weak guarantees and strictly consistent systems with limited scalability, to provide the best of both worlds. This section overviews the two families of distributed systems whose techniques we leverage in building Scatter.

Distributed Hash Tables (DHTs): DHTs are a class of highly distributed storage systems providing scalable, key based lookup of objects in dynamic network environments. As a distributed systems building primitive, DHTs have proven remarkably versatile, with application developers having leveraged scalable lookup to support a variety of distributed applications. They are actively used in the wild as the infrastructure for peer-to-peer systems on the order of millions of users.

In a traditional DHT, both application data and node IDs are hashed to a key, and data is stored at the node whose hash value immediately precedes (or follows) the key. In many DHTs, the node storing the key's value replicates

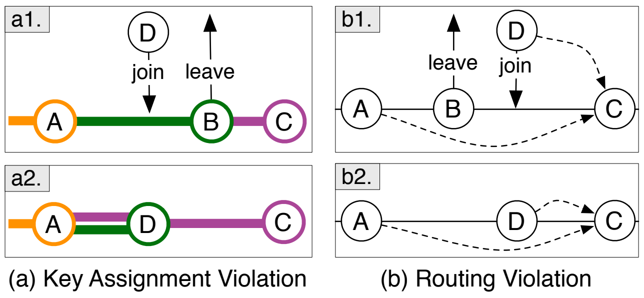


Figure 1: Two examples demonstrating how (a) key assignment consistency and (b) routing integrity may be violated in a traditional DHT. Bold lines indicate key assignment and are associated with nodes. Dotted lines indicate successor pointers. Both scenarios arise when nodes join and leave concurrently, as pictured in (a1) and (b1). The violation in (a2) may result in clients observing inconsistent key values, while (b2) jeopardizes overlay connectivity.

the data to its neighbors for better reliability and availability [30]. Even so, many DHTs suffer inconsistencies in certain failure cases, both in how keys are assigned to nodes, and in how requests are routed to keys, yielding inconsistent results or reduced levels of availability. These issues are not new [12, 4]; we recite them to provide context for our work.

Assignment Violation: A fundamental DHT correctness property is for each key to be managed by at most one node. We refer to this property as *assignment consistency*. This property is violated when multiple nodes claim ownership over the same key. In the figure, a section of a DHT ring is managed by three nodes, identified by their key values A, B, and C. A new node D joins at a key between A and B and takes over the key-range $(A, D]$. However, before B can let C know of this change in the key-range assignment, B fails. Node C detects the failure and takes over the key-range $(A, B]$ maintained by B. This key-range, however, includes keys maintained by D. As a result, clients accessing keys in $(A, D]$ may observe inconsistent key values depending on whether they are routed to node C or D.

Routing Violation: Another basic correctness property stipulates that the system maintains consistent routing entries at nodes so that the system can route lookup requests to the appropriate node. In fact, the correctness of certain links is essential for the overlay to remain connected. For example, the Chord DHT relies on the consistency of node successor pointers (routing table entries that reference the next node in the key-space) to maintain DHT connectivity [35]. Figure 1b illustrates how a routing violation may occur when node joins and leaves are not handled atomically. In the figure, node D joins at a key between B and C, and B fails immediately after. Node D has a successor pointer correctly set to C, however, A is not aware of D and incorrectly believes that C is its successor (When a successor fails, a node uses its locally available information to set its successor pointer to the failed node's successor). In this scenario, messages routed through A to keys maintained by D will skip over node D and will be incorrectly forwarded to node C. A more complex routing algorithm that allows for backtracking may avoid this scenario, but such tweaks come at the risk of routing loops [35]. More generally, such routing inconsistencies jeopardize connectivity and may lead to system partitions.

Both violations occur for keys in DHTs, e.g., one study of OpenDHT found that on average 5% of the keys are owned by multiple nodes simultaneously even in settings with low churn [31]. The two examples given above illustrate how such a scenario may occur in the context of a Chord-like system, but these issues are known to affect all types of self-organizing systems in deployment [12].

Needless to say, inconsistent naming and routing can make it challenging for developers to understand and use a DHT. Inconsistent naming and routing also complicates system performance. For example, if a particular key becomes a hotspot, we may wish to shift the load from nearby keys to other nodes, and potentially to shift responsibility for managing the key to a well-provisioned node. In a traditional DHT, however, doing so would increase the likelihood of naming and routing inconsistencies. Similarly, if a popular key happens to land on a node that is likely to exit the system shortly (e.g., because it only recently joined), we can improve overall system availability by changing the key's assignment to a better provisioned, more stable node, but only if we can make assignment changes reliably and consistently.

One approach to addressing these anomalies is to broadcast all node join and leave events to all nodes in the system, as in Dynamo. This way, every node has an eventually consistent view of its key-range, at some scalability cost. Since key storage in Dynamo is only eventually consistent, applications must already be written to tolerate temporary inconsistency. Further, since all nodes in the DHT know the complete set of nodes participating in the DHT, routing is simplified.

Coordination Services: In enterprise settings, applications desiring strong consistency and high availability use coordination services such as Chubby [2] or ZooKeeper [14]. These services use rigorous distributed algorithms with provable properties to implement strong consistency semantics even in the face of failures. For instance, ZooKeeper relies on an atomic broadcast protocol, while Chubby uses the Paxos distributed consensus algorithm [18] for fault-tolerant replication and agreement on the order of operations.

Coordination services are, however, scale-limited as every update to a replicated data object requires communication with some *quorum* of all nodes participating in the service; therefore the performance of replication protocols rapidly degrades as the number of participating nodes increases (see Figure 9(a) and [14]). Scatter is designed with the following insight: what if we had many instances of a coordination service, cooperatively managing a large scale storage system?

3. SCATTER OVERVIEW

We now describe the design of Scatter, a scalable consistent storage layer designed to support very large scale peer-to-peer systems. We discuss our goals and assumptions, provide an overview of the structure of Scatter, and then discuss the technical challenges in building Scatter.

3.1 Goals and Assumptions

Scatter has three primary goals:

1. **Consistency:** Scatter provides linearizable consistency semantics for operations on a single key/value pair, despite (1) lossy and variable-latency network connections, (2) dynamic system membership including uncontrolled departures, and (3) transient, asymmetric communication faults.

2. **Scalability:** Scatter is designed to scale to the largest deployed DHT systems with more than a million heterogeneous nodes with diverse churn rates, computational capacities, and network bandwidths.

3. **Adaptability:** Scatter is designed to be self-optimizing to a variety of dynamic operating conditions. For example, Scatter reconfigures itself as nodes come and go to preserve the desired balance between high availability and high performance. It can also be tuned to optimize for both WAN and LAN environments.

Our design is limited in the kinds of failures it can handle. Specifically, we are not robust to malicious behavior, such as Byzantine faults and denial of service attacks, nor do we provide a mechanism for continued operation during pervasive network outages or correlated and widespread node outages. We leave adding these features to future work.

3.2 Design Overview

While existing systems partially satisfy some of our requirements outlined in the preceding paragraphs, none exhibit all three. Therefore, we set out to design a new system, Scatter, that synthesizes techniques from a spectrum of distributed storage systems.

The first technique we employ to achieve our goals is to use self-managing sets of nodes, which we term *groups*, rather than individual nodes as building blocks for the system. Groups internally use a suite of replicated state machine (RSM) mechanisms [33] based on the Paxos consensus algorithm [18] as a basis for consistency and fault-tolerance. Scatter also implements many standard extensions and optimizations [5] to the basic Paxos algorithm, including: (a) an elected leader to initiate actions on behalf of the group as a whole, and (b) reconfiguration algorithms [19] to both exclude failed members and include new members over time.

As groups maintain internal integrity using consensus protocols with provable properties, a simple and aggressive failure detector suffices. Nodes that are excluded from a group after being detected as failed can not influence any future actions of the group. On the other hand, the failure to quickly detect a failed node will not impede the liveness of the group because only a quorum of the current members are needed to make progress.

Scatter implements a simple DHT model in which a circular key-space is partitioned among groups (see Figure 2). Each group maintains up-to-date knowledge of the two neighboring groups that immediately precede and follow it in the key-space. These consistent lookup links form a global ring topology, on top of which Scatter layers a best-effort routing policy based on cached hints. If this soft routing state is stale or incomplete, then Scatter relies on the underlying consistent ring topology as ground truth.

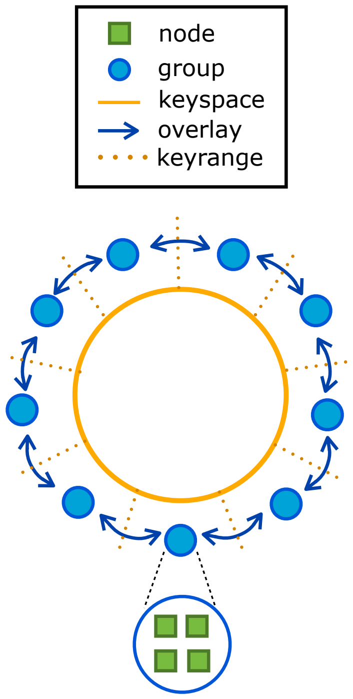


Figure 2: Overview of Scatter architecture

Carefully engineered groups go a long way to meeting our stated design goals for Scatter. However, a system composed of some static set of groups will be inherently limited in many ways. For example, if there is a burst of failures or sufficient disparity between the rate of leaves and joins for a particular group, then that group is at risk of losing a functional quorum. Not only is a static set of groups limited in robustness, but it is also restricted in both scalability and the ability to adapt gracefully to dynamic conditions. For instance, the performance of consensus algorithms degrades significantly as the number of participants increases. Therefore, a static set of groups will not be able to incrementally scale with the online addition of resources. As another example, if one group is responsible for a hotspot in the key-space, it needs some way of coordinating with other groups, which may be underutilized, to alleviate the hotspot.

Therefore, we provide mechanisms to support the following *multi-group operations*:

- *split*: partition the state of an existing group into two groups.
- *merge*: create a new group from the union of the state of two neighboring groups.
- *migrate*: move members from one group to a different group.
- *repartition*: change the key-space partitioning between two adjacent groups.

Although our approach is straightforward and combines well-known techniques from the literature, we encountered a number of technical challenges that may not be apparent from a cursory inspection of the high-level design.

Atomicity: Multi-group operations modify the routing state across multiple groups, but as we discussed in Section 2, strong consistency is difficult or impossible to guarantee when modifications to the routing topology are not atomic. Therefore, we chose to structure each multi-group operation in Scatter as a distributed transaction. We illustrate this design pattern, which we call nested consensus, in Figure 3. We believe that this general idea of structuring protocols as communication between replicated participants, rather than between individual nodes, can be applied more generally to the construction of scalable, consistent distributed systems.

Nested consensus uses a two-tiered approach. At the top tier, groups execute a two-phase commit protocol (2PC), while within each group the actions that the group takes are agreed on using consensus protocols. Multi-group operations are coordinated by whichever group decides to initiate the transaction as a result of some local policy. As Scatter is decentralized, multiple groups can concurrently initiate conflicting transactions. Section 4 details the mechanisms used to coordinate distributed transactions across groups.

Performance: Strong consistency in distributed systems is commonly thought to come with an unacceptably high performance or availability costs. The challenge of maximizing system performance influenced every level of Scatter's design and implementation — whether defined in terms of latency, throughput, or availability — without compromising core integrity. Although many before us have shown that strongly consistent replication techniques can be implemented efficiently at small scale, the bigger challenge for us was the additional layer of "heavy-weight" mechanisms — distributed transactions — on top of multiple instantiations of independent replicated state machines.

Self Organization: Our choice of complete decentralization makes the design of policies non-trivial. In contrast to designs in which a system deployment is tuned through human intervention or an omnipotent component, Scatter is tuned by the actions of individual groups using local information for optimization. Section 6 outlines various techniques for optimizing the resilience, performance, and load-balance of Scatter groups using local or partially sampled non-local information.

4. GROUP COORDINATION

In this section, we describe how we use nested consensus to implement multi-group operations. Section 4.1 characterizes our requirements for a consistent and available overlay topology. Section 4.2 details the nested consensus technique, and Section 4.3 walks through a concrete example of the group split operation.

4.1 Overlay Consistency Requirements

Scatter's overlay was designed to solve the consistency and availability problems discussed in Section 2. As Scatter is defined in terms of groups rather than nodes, we will slightly rephrase the assignment consistency correctness condition as the following system invariant: groups that are adjacent in the overlay agree on a partitioning of the key-space between them. For individual links in the overlay to remain highly available, Scatter maintains an additional invariant: a group can always reach its adjacent groups. Although these invariants are locally defined they are sufficient to provide global consistency and availability properties for Scatter's overlay.

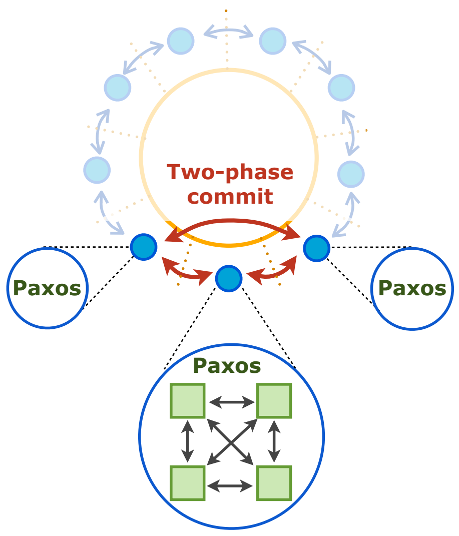


Figure 3: Overview of nested consensus. Groups coordinate distributed transactions using a two-phase commit protocol. Within each group, nodes coordinate using the Paxos distributed consensus algorithm.

We can derive further requirements from these conditions for operations that modify either the set of groups, the membership of groups, or the partitioning of the key-space among groups. For instance, in order for a group G_a to be able to communicate directly with an adjacent group G_b, G_a must have knowledge of some subset of G_b's members. The following property is sufficient, but perhaps stronger than necessary, to maintain this connectivity: every adjacent group of G_b has up-to-date knowledge of the membership of G_b. This requirement motivated our implementation of operations that modify the membership of a group G_b to be *eagerly replicated* across all groups adjacent to G_b in the overlay.

In keeping with our goal to build on classic fault-tolerant distributed algorithms rather than inventing ad-hoc protocols, we chose to structure group membership updates as distributed transactions across groups. This approach not only satisfied our requirement of eager replication but provided a powerful framework for implementing the more challenging multi-group operations such as group splits and merges. Consider, for example, the scenario in Figure 4 where two adjacent groups, G_1 and G_2, propose a merge operation simultaneously. To maintain Scatter's two overlay consistency invariants, the adjacent groups G_0 and G_4 must be involved as well. Note that the changes required by G_1's proposal and G_2's proposal conflict — i.e., if both operations were executed concurrently they would violate the structural integrity of the overlay. These anomalies are prevented by the atomicity and concurrency control provided by our transactional framework.

4.2 Nested Consensus

Scatter implements distributed transactions across groups using a technique we call *nested consensus* (Figure 3). At a high level, groups execute a two-phase commit protocol

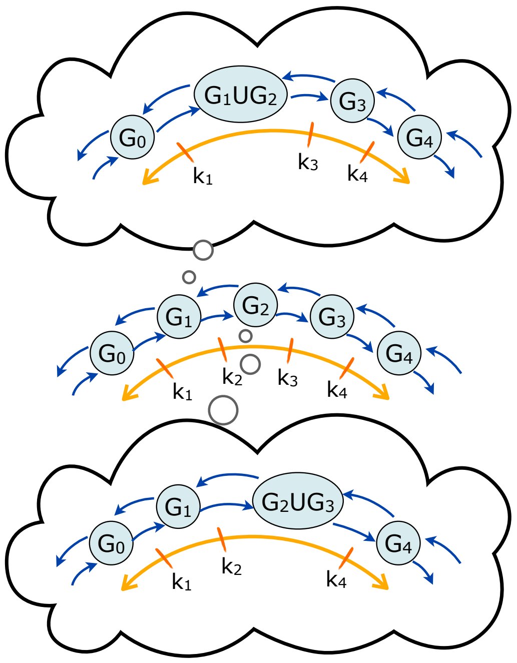


Figure 4: Scenario where two adjacent groups, G_1 and G_2, propose a merge operation simultaneously. G_1 proposes a merge of G_1 and G_2, while G_2 proposes a merge of G_2 and G_3. These two proposals conflict.

(2PC); before a group executes a step in the 2PC protocol it uses the Paxos distributed consensus algorithm to internally replicate the decision to execute the step. Thus distributed replication plays the role of write-ahead logging to stable storage in the classic 2PC protocol.

We will refer to the group initiating a transaction as the *coordinator* group and to the other groups involved as the *participant* groups. The following sequence of steps loosely captures the overall structure of nested consensus:

1. The coordinator group replicates the decision to initiate the transaction.
2. The coordinator group broadcasts a transaction *prepare* message to the nodes of the participant groups.
3. Upon receiving the *prepare* message, a participant group decides whether or not to commit the proposed transaction and replicates its vote.
4. A participant group broadcasts a *commit* or *abort* message to the nodes of the coordinator group.
5. When the votes of all participant groups is known, the coordinator group replicates whether or not the transaction was committed.
6. The coordinator group broadcasts the outcome of the transaction to all participant groups.
7. Participant groups replicate the transaction outcome.
8. When a group learns that a transaction has been committed then it executes the steps of the proposed transaction, the particulars of which depend on the multi-group operation.

Note that nested consensus is a non-blocking protocol. Provided a majority of nodes in each group remain alive and connected, the two phase commit protocol will terminate. Even if the previous leader of the coordinating group

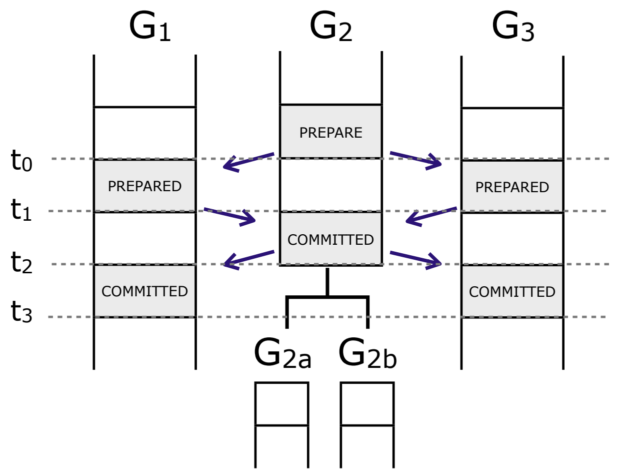


Figure 5: Group G_2 splits into two groups, G_{2a} and G_{2b}. Groups G_1, G_2, and G_3 participate in the distributed transaction. Causal time advances vertically, and messages between groups are represented by arrows. The cells beneath each group name represent the totally-ordered replicated log of transaction steps for that group.

fails, another node can take its place and resume the transaction. This is not the case for applying two phase commit to managing routing state in a traditional DHT.

In our implementation the leader of a group initiates every action of the group, but we note that a judicious use of broadcasts and message batching lowers the apparently high number of message rounds implied by the above steps. We also think that the large body of work on optimizing distributed transactions could be applied to further optimize performance of nested consensus, but our experimental evaluations in Section 7 show that performance is reasonable even with a relatively conservative implementation.

Our implementation encourages concurrency while respecting safety. For example, the storage service (Section 5) continues to process client requests during the execution of group transactions except for a brief period of unavailability during any reconfiguration required by a committed transaction. Also, groups continue to serve lookup requests during transactions that modify the partitioning of the key-space provided that the lookups are serialized with respect to the transaction commit.

To illustrate the mechanics of nested consensus, the remainder of the section walks through an example group split operation and then considers the behavior of this mechanism in the presence of faults and concurrent transactions.

4.3 Example: Group Split

Figure 5 illustrates three groups executing a split transaction. For clarity, this example demonstrates the necessary steps in nested consensus in the simplest case — a non-faulty leader and no concurrent transactions. At t_0, G_2 has replicated its intent to split into the two groups G_{2a} and G_{2b} and then sends a 2PC PREPARE message to G_1 and G_3. In parallel, G_1 and G_3 internally replicate their vote to commit the proposed split before replying to G_0. After each group has learned and replicated the outcome (COMMITTED) of the split operation at time t_3, then the following updates are executed by the respective group: (1) G_1 updates its successor pointer to G_{2a}, (2) G_3 updates its predecessor pointer to G_{2b}, and (3) G_2 executes a *replicated state machine reconfiguration* to instantiate the two new groups which partition between them G_2's original key-range and set of member nodes.

To introduce some of the engineering considerations needed for nested consensus, we consider the behavior of this example in more challenging conditions. First, suppose that the leader of G_1 fails after replicating intent to begin the transaction but before sending the PREPARE messages to the participant groups. The other nodes of G_1 will eventually detect the leader failure and elect a new leader. When the new leader is elected, it behaves just like a restarted classical transaction manager: it queries the replicated write-ahead log and continues executing the transaction. We also implemented standard mechanisms for message timeouts and re-deliveries, with the caveat that individual steps should be implemented so that they are idempotent or have no effect when re-executed.

We return to the question of concurrency control. Say that G_1 proposed a merge operation with G_2 simultaneously with G_2's split proposal. The simplest response is to enforce mutual exclusion between transactions by participant groups voting to abort liberally. We implemented a slightly less restrictive definition of conflicting multi-group operations by defining a lock for each link in the overlay. Finer-grained locks reduce the incidence of deadlock; for example, two groups, G_1 and G_3, that are separated by two hops in the overlay would be able to update their membership concurrently; whereas with complete mutual exclusion these two operations would conflict at the group in the middle (G_2).

5. STORAGE SERVICE

A consistent and scalable lookup service provides a useful abstraction onto which richer functionality can be layered. This section describes the storage service that each Scatter group provides for its range of the global key-space. To evaluate Scatter, we implemented a peer-to-peer Twitter-like application layered on a standard DHT-interface. This allowed us to do a relatively direct comparison with OpenDHT in Section 7.

As explained in Section 4, each group uses Paxos to replicate the intermediate state needed for multi-group operations. Since multi-group operations are triggered by environmental changes such as churn or shifts in load, our design assumes these occur with low frequency in comparison to normal client operations. Therefore Scatter optimizes each group to provide low latency and high throughput client storage.

To improve throughput, we partition each group's storage state among its member nodes (see Figure 6). Storage operations in Scatter take the form of a simple read or write on an individual key. Each operation is forwarded to the node of the group assigned to a particular key – referred to as the *primary* for that key.

The group leader replicates information regarding the assignment of keys to primaries using Paxos, as it does with the state for multi-group operations. The key assignment is cached as soft state by the routing service in the other Scat-

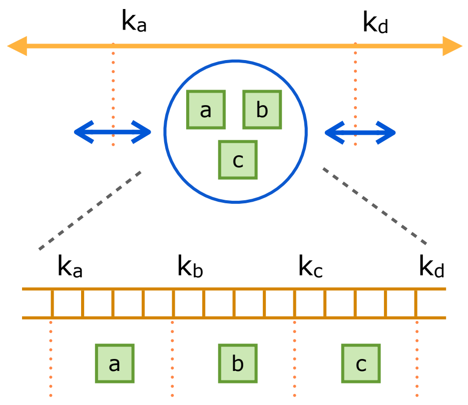


Figure 6: Example Scatter group composed of three nodes (a, b, c) and assigned to the key-range $[k_a, k_d)$. The group's key-range is partitioned such that each node of the group is the *primary* for some subset of the group's key-space. The primary of a key-range owns those keys and both orders and replicates all operations on the keys to the other nodes in the group; e.g., a is assigned $[k_a, k_b]$ and replicates all updates to these keys to b and c using Paxos.

ter groups. All messages are implemented on top of UDP, and Scatter makes no guarantees about reliable delivery or ordering of client messages. Once an operation is routed to the correct group for a given key, then any node in the group will forward the operation to the appropriate primary. Each primary uses Paxos to replicate operations on its key-range to all the other nodes in the group – this provides linearizability. Our use of the Paxos algorithm in this case behaves very much like other primary-backup replication protocols – a single message round usually suffices for replication, and operations on different keys and different primaries are not synchronized with respect to each other.

In parliamentary terms [18], the structure within a group can be explained as follows. The group nodes form the group *parliament* which elects a parliamentary leader and then divides the law into disjoint areas, forming a separate *committee* to manage each resulting area of the law independently. All members of parliament are also a member of every committee, but each committee appoints a different committee chair (i.e., the primary) such that no individual member of parliament is unfairly burdened in comparison to his peers. Because the chair is a distinguished proposer in his area, in the common case only a single round of messages is required to pass a committee decree. Further, since committees are assigned to disjoint areas of the law, decrees in different committees can be processed concurrently without requiring a total ordering of decrees among committees.

In addition to the basic mechanics described in this section and the previous section, Scatter implements additional optimizations including:

- **Leases**: Our mechanisms for delegating keys to primaries does not require time-based leases; however, they can be turned on for a given deployment. Leases allow primaries to satisfy reads without communicating to the rest of the group; however, the use of leases can also delay the execution of certain group operations when a primary fails.
- **Diskless Paxos**: Our implementation of Paxos does not require writing to disk. Nodes that restart just rejoin the system as new nodes.
- **Relaxed Reads**: All replicas for a given key can answer read requests from local – possibly stale – state. Relaxed reads violate linearizability, but are provided as an option for clients.

	Load Balance	Latency	Resilience
Low churn Uniform latency	✓		
Low churn Non-uniform latency	✓	✓	
High churn Non-uniform latency	✓	✓	✓

Table 1: Deployment settings and system properties that a Scatter policy may target. A ✓ indicates that we have developed a policy for the combination of setting and property.

6. SAMPLE POLICIES

An important property of Scatter's design is the separation of policy from mechanism. For example, the mechanism by which a node joins a group does not prescribe how the target group is selected. Policies enable Scatter to adapt to a wide range of operating conditions and are a powerful means of altering system behavior with no change to any of the underlying mechanisms.

In this section we describe the policies that we have found to be effective in the three experimental settings where we have deployed and evaluated Scatter (see Section 7). These are: (1) low churn and uniform network latency, (2) low churn and non-uniform network latency, and (3) high churn and non-uniform network latency. Table 1 lists each of these settings, and three system properties that a potential policy might optimize. A ✓ in the table indicates that we have developed a policy for the corresponding combination of deployment setting and system property. We now describe the policies for each of the three system properties.

6.1 Resilience

Scatter must be resilient to node churn as nodes join and depart the system unexpectedly. A Scatter group with $2k+1$ nodes guarantees data availability with up to k node failures. With more than k failures, a group cannot process client operations safely. To improve resilience, Scatter employs a policy that prompts a group to merge with an adjacent group if its node count is below a predefined threshold. This maintains high data availability and helps prevent data loss. This policy trades-off availability for performance since smaller groups are more efficient.

To determine the appropriate group size threshold we carried out a simulation, parameterized with the base reconfiguration latency plotted in Figure 12(b). Figure 7 plots the probability of group failure for different group sizes for two node churn rates with node lifetimes drawn from heavy-tailed Pareto distributions observed in typical peer-to-peer systems [3, 32]. The plot indicates that a modest group size of 8-12 prevents group failure with high probability.

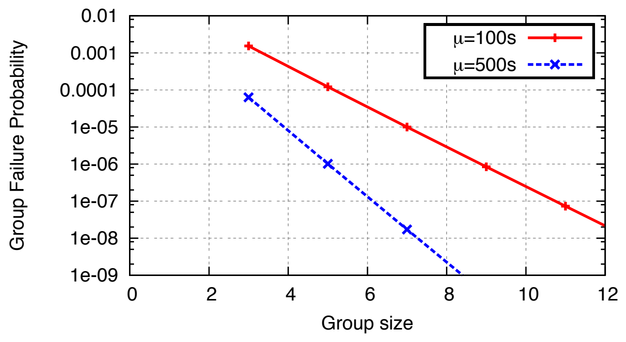


Figure 7: Impact of group size on group failure probability for two Pareto distributed node churn rates, with average lifetimes $\mu = 100s$ and $\mu = 500s$.

The resilience policy also directs how nodes join the system. A new node samples k random groups and joins the group that is most likely to fail. The group failure probability is computed using node lifetime distribution information, if available. In the absence of this data, the policy defaults to having a new node join the sampled group with the fewest nodes. The default policy also takes into account the physical diversity of nodes in a group, e.g., the number of distinct BGP prefixes spanned by the group. It then assigns a joining node to a group that has the smallest number of nodes and spans a limited number of BGP prefixes in order to optimize for both uncorrelated and correlated failures. We performed a large-scale simulation to determine the impact of the number of groups sampled and found that checking four groups is sufficient to significantly reduce the number of reconfiguration operations performed later. If multiple groups have the expected failure probability below the desired threshold, then the new node picks the target group based on the policy for optimizing latency as described below.

6.2 Latency

Client latency depends on its time to reach the primary, and the time for the primary to reach consensus with the other replicas. A join policy can optimize client latency by placing new nodes into groups where their latencies to the other nodes in the group will be low. The *latency-optimized join* policy accomplishes this by having the joining node randomly select k groups and pass a `no-op` operation in each of them as a pseudo primary. This allows the node to estimate the performance of operating within each group. While performing these operations, nodes do not explicitly join and leave each group. The node then joins the group with the smallest command execution latency. Note that latency-optimized join is used only as a secondary metric when there are multiple candidate groups with the desired resiliency properties. As a consequence, these performance optimizations are not at the cost of reduced levels of physical diversity or group robustness. Experiments in Section 7.1.1 compare the latency-optimized join policy with $k = 3$ against the random join policy.

The *latency-optimized leader selection* policy optimizes the RSM command latency in a different way – the group elects the node that has the lowest Paxos agreement latency as the leader. We evaluate the impact of this policy on reconfiguration, merge, and split costs in Section 7.1.3.

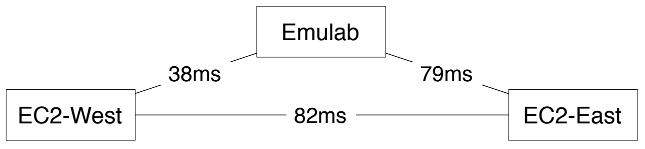


Figure 8: Inter-site latencies in the multi-cluster setting used in experiments.

6.3 Load Balance

Scatter also balances load across groups in order to achieve scalable and predictable performance. A simple and direct method for balancing load is to direct a new node to join the group that is heavily loaded. The *load-balanced join* policy does exactly this – a joining node samples k groups, selects groups that have low failure probability, and then joins the group that has processed the most client operations in the recent past. The load-balance policy also repartitions the keyspace among adjacent groups when the request load to their respective keyspaces is skewed. In our implementation, groups repartition their keyspaces proportionally to their respective loads whenever a group's load is a factor of 1.6 or above that of its neighboring group. As this check is performed locally between adjacent groups, it does not require global load monitoring, but it might require multiple iterations of the load-balancing operation to disperse hotspots. We note that the overall, cumulative effect of many concurrent locally optimal modifications is non-trivial to understand. A thorough analysis of the effect of local decisions on global state is an intriguing direction for future work.

7. EVALUATION

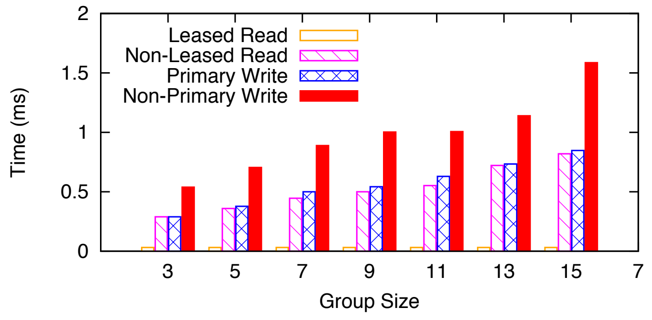
We evaluated Scatter across three deployment environments, corresponding to the churn/latency settings listed in Table 1: (1) **single cluster**: a homogeneous and dedicated Emulab cluster to evaluate the low churn/uniform latency setting; (2) **multi-cluster**: multiple dedicated clusters (Emulab and Amazon's EC2) at LAN sites connected over the wide-area to evaluate the low churn/non-uniform latency setting; (3) **peer-to-peer**: machines from PlanetLab in the wide-area to evaluate the high churn/non-uniform latency setting.

In all experiments Scatter ran on a single core on a given node. On Emulab we used 150 nodes with 2.4GHz 64-bit Xeon processor cores. On PlanetLab we used 840 nodes, essentially all nodes on which we could install both Scatter and OpenDHT.

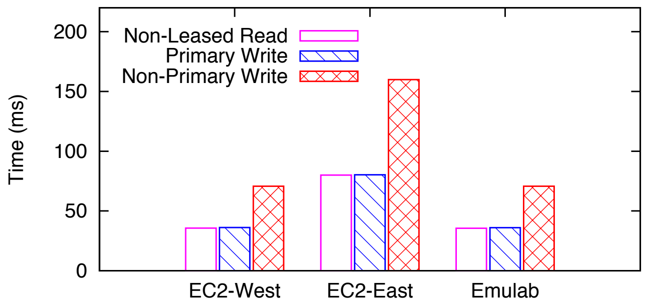
For multi-cluster experiments we used 50 nodes each from Emulab (Utah), EC2-West (California) and EC2-East (Virginia). The processors on the EC2 nodes were also 64-bit processor cores clocked at 2.4GHz. Figure 8 details the inter-site latencies for the multi-cluster experiments. We performed our multi-cluster experiments using nodes at physically distinct locations in order to study the performance of our system under realistic wide-area network conditions.

We used Berkeley-DB for persistent disk-based storage, and a memory cache to pipeline operations to BDB in the background.

Section 7.1 quantifies specific Scatter overheads with deployments on dedicated testbeds (single-cluster, multi-cluster). We then evaluate Scatter at large scales on PlanetLab with



(a) Single Cluster



(b) Multi-Cluster

Figure 9: Latency of different client operations in (a) a single-cluster deployment for groups of different sizes, and (b) a multi-cluster deployment in which no site had a majority of nodes in the group.

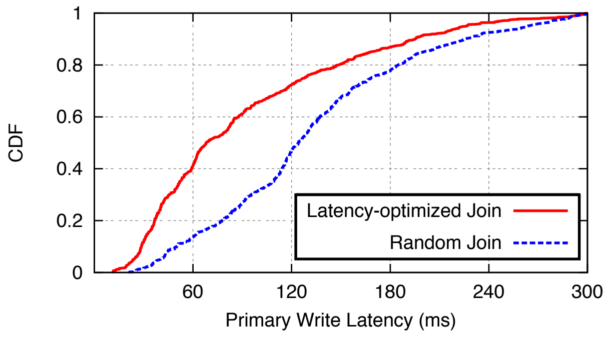


Figure 10: The impact of join policy on write latency in two PlanetLab deployments. The latency-optimized join policy is described in Section 6.2. The random join policy directs nodes to join a group at random.

varying churn rates in the context of a Twitter-like publish-subscribe application called Chirp in Section 7.2, and also compare it to a Chirp implementation on top of OpenDHT.

7.1 Microbenchmarks

In this section we show that a Scatter group imposes a minor latency overhead and that primaries dramatically increase group operation processing throughput. Then, we evaluate the latency of group reconfiguration, split and merge. The results indicate that group operations are more expensive than client operations, but the overheads are tolerable since these operations are infrequent.

7.1.1 Latency

Figure 9 plots a group's client operation processing latency for single cluster and multi-cluster settings. The plotted latencies do not include the network delay between the client and the group. The client perceived latency will have an additional delay component that is simply the latency from the client to the target group. We present the end-to-end application-level latencies in Section 7.2.

Figure 9(a) plots client operation latency for different operations in groups of different sizes. The latency of leased reads did not vary with group size – it is processed locally by the primary. Non-leased reads were slightly faster than primary writes as they differ only in the storage layer overhead. Non-primary writes were significantly slower than primary-based operations because the primary uses the faster leader-Paxos for consensus.

In the multi-cluster setting no site had a node majority. Figure 9(b) plots the latency for operations that require a primary to coordinate with nodes from at least one other site. As a result, inter-cluster WAN latency dominates client operation latency. As expected, operations initiated by primaries at EC2-East had significantly higher latency, while operations by primaries at EC2-West and Emulab had comparable latency.

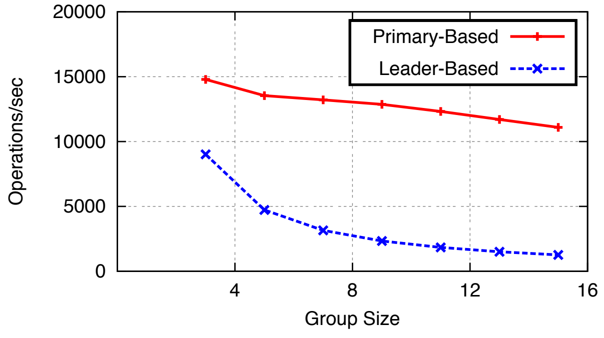
To illustrate how policy may impact client operation latency, Figure 10 compares the impact of latency-optimized join policy with $k = 3$ (described in Section 6.2) to the random join policy on the primary's write latency in a PlanetLab setting. In both PlanetLab deployments, nodes joined Scatter using the respective policy, and after all nodes joined, millions of writes were performed to random locations. The effect of the latency-optimized policy is a clustering of nodes that are close in latency into the same group. Figure 10 shows that this policy greatly improves write performance over the random join policy – median latency decreased by 45%, from 124ms to 68ms.

Latencies in the PlanetLab deployment also demonstrate the benefit of majority consensus in mitigating the impact of slow-performing outlier nodes on group operation latency. Though PlanetLab nodes are globally distributed, the 124ms median latency of a primary write (with random join policy) is not much higher than that of the multi-cluster setting. Slow nodes impose a latency cost but they also benefit the system overall as they improve fault tolerance by consistently replicating state, albeit slowly.

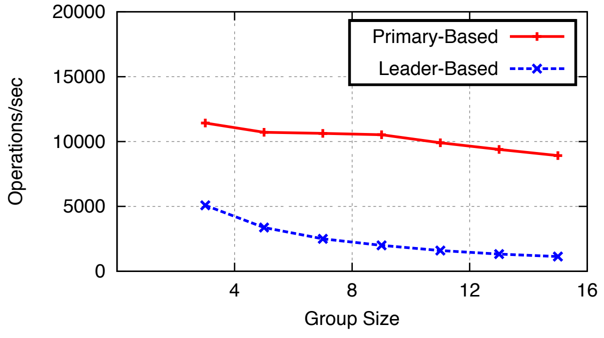
7.1.2 Throughput

Figure 11 plots write throughput of a single group in single cluster and multi-cluster settings. Writes were performed on randomly selected segments. Throughput was determined by varying both the number of clients (up to 20) and the number of outstanding operations per client (up to 100).

The figure demonstrates the performance benefit of using primaries. In both settings, a single leader becomes a scalability bottleneck and throughput quickly degrades for groups with more nodes. This happens because the message overhead associated with executing a group command is linear in group size. Each additional primary, however, adds

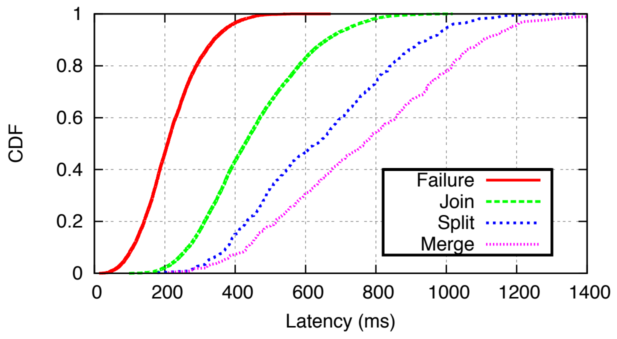


(a) Single Cluster

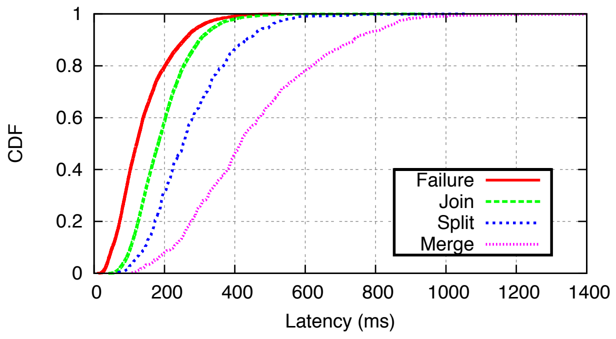


(b) Multi-cluster

Figure 11: Scatter group throughput in single cluster and multi-cluster settings.



(a) Unoptimized



(b) Join and Leader Optimized for Latency

Figure 12: CDFs of group reconfiguration latencies for a P2P setting with two sets of policies: (a) random join and random leader, and (b) latency-optimized join and latency-optimized leader.

extra capacity to the group since primaries process client operations in parallel and also pipeline client operations. The result is that in return for higher reliability (afforded by having more nodes) a group's throughput decreases only slightly when using primaries.

Though the latency of a typical operation in the multi-cluster deployment is significantly higher than the corresponding operation in the single cluster setting, group throughput in the multi-cluster setting (Figure 11(a)) is within 30% of the group throughput in a single cluster setting (Figure 11(b)). And for large groups this disparity is marginal. The reason for this is the pipelining of client requests by group primaries, which allows the system to mask the cost of wide-area network communication.

7.1.3 Reconfiguration, Split, and Merge

We evaluated the latency cost of group reconfiguration, split, and merge operations. In the case of failure, this latency is the duration between a failure detector sensing a failure and the completion of the resulting reconfiguration. Table 2 lists the average latencies and standard deviations for single and multi- cluster settings across thousands of runs and across group sizes 2-13. These measurements do not account for data transfer latency.

Basic single cluster latency. In the single cluster setting all operations take less than 10ms. Splitting and merging are the most expensive operations as they require co-

	Single cluster	Multi-cluster (Unopt.)	Multi-cluster (Opt. leader)
Failure	2.04 ± 0.44	90.9 ± 31.8	55.6 ± 7.6
Join	3.32 ± 0.54	208.8 ± 48.8	135.8 ± 15.2
Split	4.01 ± 0.73	246.5 ± 45.4	178.5 ± 15.1
Merge	4.79 ± 1.01	307.6 ± 69.8	200.7 ± 24.4

Table 2: Group reconfiguration, split, and merge latencies in milliseconds and standard deviations for different deployment settings.

ordination between groups, and merging is more expensive because it involves more groups than splitting.

Impact of policy on multi-cluster latency. The single-cluster setting provides little opportunity for optimization due to latency homogeneity. However, in the multi-cluster settings, we can decrease the latency cost with a leader election policy. Table 2 lists latencies for two multi-cluster deployments, one with a random leader election policy, and one that used a latency-optimized leader policy described in Section 6.2. From the table, the latency optimizing policy significantly reduced the latency cost of all operations.

Impact of policy on PlanetLab latency. Figure 12 plots CDFs of latencies for the PlanetLab deployment. It compares the random join with random leader policies (Figure 12(a)) against latency-optimized join and latency-optimized leader policies described in Section 6.2 (Figure 12(b)). In combination, the two latency optimizing policies shift the CDF curves to the left, decreasing the latency of all operations – reconfiguration, split and merge.

7.2 Application-level Benchmarks

To study the macro-level behavior of Scatter, we built and deployed Chirp, a Twitter-like application. In this section we compare PlanetLab deployments of Chirp on top of Scatter and OpenDHT. We compare our implementation with OpenDHT, which is an open-source DHT implementation that is currently deployed on PlanetLab. OpenDHT uses lightweight techniques for DHT maintenance, and its access latencies are comparable to that of other DHTs [28]. It therefore allows us to evaluate the impact of the more heavy-weight techniques used in Scatter.

For a fair comparison, both Scatter and OpenDHT send node heartbeat messages every 0.5s. After four consecutive heartbeat failures OpenDHT re-replicates failed node's keys, and Scatter reconfigures to exclude the failed node and re-partitions the group's keyspace among primaries. Additionally, Scatter used the same base-16 recursive routing algorithm as is used by OpenDHT. Only forward and reverse group pointers were maintained consistently in Scatter, but it relied on these only when the soft routing state turned out to be inconsistent. In both systems the replication factor was set to provide at least seven 9s of reliability, i.e., with an average lifetime of 100 seconds, we use 9 replicas (see Figure 7).

To induce churn we use two different lifetime distributions, Poisson and Pareto. Pareto is a typical way of modeling churn in P2P systems [3, 32], and Poisson is a common reference distribution. For both churn distributions a node's join policy joined the group with the lowest expected residual lifetime — for Poisson this is equivalent to joining the group with the fewest nodes.

7.2.1 *Chirp overview*

Chirp works much like Twitter; to participate in the system a user u creates a user-name, which is associated with two user-specific keys, $K^u_{updates}$ and $K^u_{follows}$, that are computed by hashing u's user-name. A user may write and post an *update*, which is at most 140 characters in length; *follow* another user; or fetch updates posted by the users being followed. An update by a user u is appended to the value of $K^u_{updates}$. When u follows u', the key $K^{u'}_{updates}$ is appended to $K^u_{follows}$, which maintains the list of all users u is following.

Appending to a key value is implemented as a non-atomic read-modify-write, requiring two storage operations. This was done to more fairly compare Scatter and OpenDHT. A key's maximum value was 8K in both systems. When a key's value capacity is exceeded (e.g., a user posts over 57 maximum-sized updates), a new key is written and the new key is appended to the end of the value of the old key, as a pointer to the continuation of the list. The Chirp client application caches previously known tails of each list accessed by the user in order to avoid repeatedly scanning through the list to fetch the most recent updates. In addition, the pointer to the tail of the list is stored at its header so that a user's followers can efficiently access the most recent updates of the user.

We evaluated the performance of Chirp on Scatter and OpenDHT by varying churn, the distribution of node lifetimes, and the popularity distribution of keys. For the experiments below, we used workloads obtained from Twitter measurement studies [17, 15]. The measurements include both the updates posted by the users and the structure of the social network over which the updates are propagated.

7.2.2 *Impact of Churn*

We first evaluate the performance by using node lifetime distributions that are Poisson distributed and by varying the mean lifetime value from 100 seconds to 1000 seconds. We based our lifetime distributions on measurements of real-world P2P systems such as Gnutella, Kazaa, FastTrack, and Overnet [32, 13, 7, 34]. For this experiment, the update/fetch Chirp workload was synthesized as follows: we played a trace of status updates derived from the Twitter measurement studies, and for each user u posting an update, we randomly selected one of u's followers and issued a request from this user to the newly posted update. Figure 13 plots performance, availability, and consistency of the fetches in this workload as we vary churn. Each data point represents the mean value for a million fetch operations.

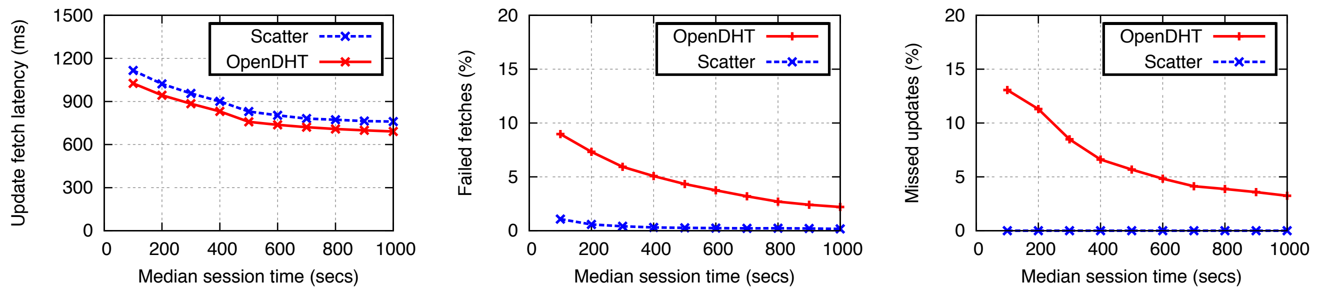
Figure 13(a) indicates that the performance of both systems degrades with increasing churn as routing state becomes increasingly stale, and the probability of the value residing on a failed node increases. OpenDHT slightly outperforms Scatter in fetch latency because a fetch in Scatter incurs a round of group communication.

Figure 13(b) shows that Scatter has better availability than OpenDHT. The availability loss in OpenDHT was often due to the lack of structural integrity, with inconsistent successor pointer information or because a key being fetched has not been assigned to any of the nodes (see Figure 1). To compute the fetch failure for Scatter in Figure 13(b) an operation was considered to have failed if a response has not been received within three seconds. The loss of availability for Scatter was because an operation may be delayed for over three seconds when the destination key belonged to a group undergoing reconfiguration in response to churn.

Figure 13(c) compares the consistency of the values stored in the two systems. OpenDHT's inconsistency results confirmed prior studies, e.g., [31] — even at low churn rates over 5% of the fetches were inconsistent. These inconsistencies stem from a failure to keep replicas consistent, either because an update to a replica failed or because different nodes have different views regarding how the keyspace is partitioned. In contrast, Scatter had no inconsistencies across all experiments.

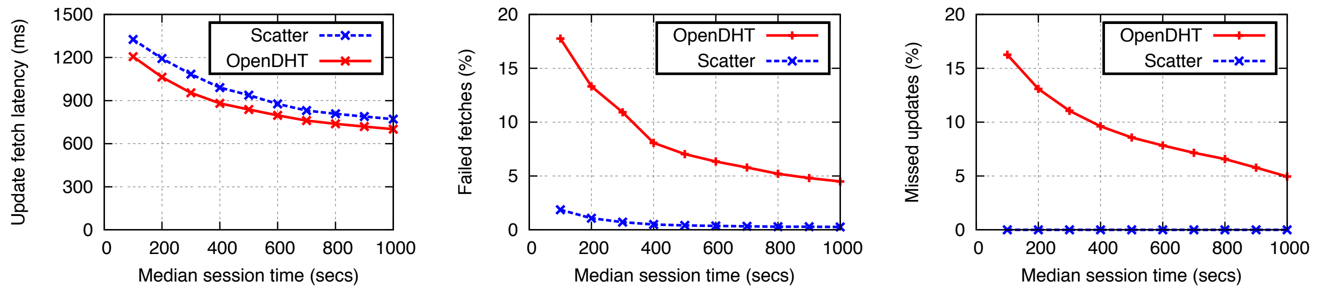
7.2.3 *Heavy tailed node lifetimes*

Next, we considered a node lifetime distribution in which nodes are drawn from a heavy-tailed Pareto distribution that is typical of many P2P workloads. Heavy-tailed distributions exhibit "memory", i.e., nodes who have been part of the system for some period of time are more likely to persist than newly arriving nodes. Scatter provides for a greater ability to optimize for skewed node lifetime distribution due to its group abstraction. Note that all of the keys associated with a group are replicated on the same set of nodes, whereas in OpenDHT each node participates in multiple different replica sets. In this setting, Scatter takes into account the measured residual lifetime distribution in the various reconfiguration operations, e.g., which group an arriving node



(a) Performance (b) Availability (c) Consistency

Figure 13: Impact of varying churn rates for Poisson distributed lifetimes. The graphs plot measurements for P2P deployments of Chirp for both Scatter (dashed line), and OpenDHT (solid line).



(a) Performance (b) Availability (c) Consistency

Figure 14: Impact of varying churn rates for Pareto distributed lifetimes ($\alpha = 1.1$).

should join, when should groups merge or split, and in determining the optimal size of the group to meet the desired (seven 9s) availability guarantee. For these experiments the workload was generated in the same way as the workload used in Section 7.2.2.

OpenDHT slightly outperformed Scatter with respect to access latency (see Figure 14(a)). However, Scatter's availability fared better under the heavy-tailed churn rate than that of OpenDHT (Figure 14(b)). As before, Scatter had no inconsistencies, while OpenDHT was more inconsistent with the heavy tailed churn rate (Figure 14(c)).

7.2.4 *Non-uniform load*

In the next experiment, we studied the impact of high load on Scatter. For this experiment, we batched and issued one million updates from the Twitter trace, and after all of the updates have been posted, the followers of the selected users fetched the updates. The fetches were issued in a random order and throttled to a rate of 10,000 fetches per second for the entire system. Note that in this experiment the keys corresponding to popular users received more requests, as the load is based on social network properties. The load is further skewed by the fact that users with a large number of followers are more likely to post updates [17].

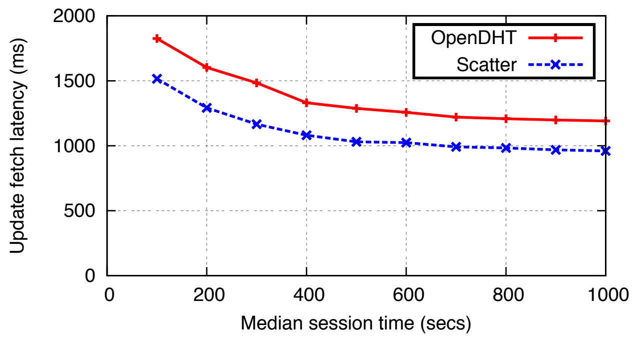
Figure 15(a) shows that Scatter had a slightly better fetch latency than OpenDHT due to its better load balance properties. However, latency in Scatter tracked OpenDHT's latency as in prior experiments (Figures 13(a) and 14(a)).

Figure 15(b) plots the normalized node load for Scatter and OpenDHT. This was computed in both systems by tracking the total number of fetch requests processed by a node, and then dividing this number by the mean. The figure shows that Scatter's load-balance policy (Section 6.3) is effective at distributing load across nodes in the system. OpenDHT's load distribution was more skewed.

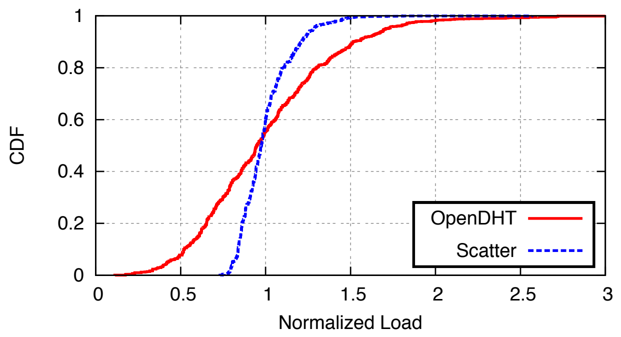
7.2.5 *Scalability*

For our final set of experiments, we evaluated the scalability of Scatter and its ability to adapt to variations in system load. We also compared Scatter with ZooKeeper, a system that provides strongly consistent storage. As ZooKeeper is a centralized and scale-limited system, we built a decentralized system comprising of multiple ZooKeeper instances, where the global keyspace is statically partitioned across the different instances. We also optimized the performance of this ZooKeeper-based alternative by basing the keyspace partitioning on the historical load estimates of the various key values; we split our workload into two halves, used the first half to derive the keyspace partitioning, and then performed the evaluations using the second half of the trace. Each ZooKeeper instance comprised of five nodes. We performed these experiments without node churn, as the system based on ZooKeeper did not have a management layer for dealing with churn.

Figure 16 plots the average throughput results with standard deviations as we vary the number of nodes in the system. The throughput of Scatter is comparable to that of the ZooKeeper-based system for small number of nodes, indicating that Scatter stacks up well against a highly optimized implementation of distributed consensus. As we increase the number of nodes, the performance of ZooKeeper-based alternative scales sub-linearly. This indicates that, even though



(a) Latency



(b) Node Load

Figure 15: High load results for Chirp with node churn distributed as Pareto($\alpha = 1.1$). (a) Scatter has better latency than OpenDHT at high loads; (b) Scatter maintains a more balanced distribution of load among its nodes than OpenDHT.

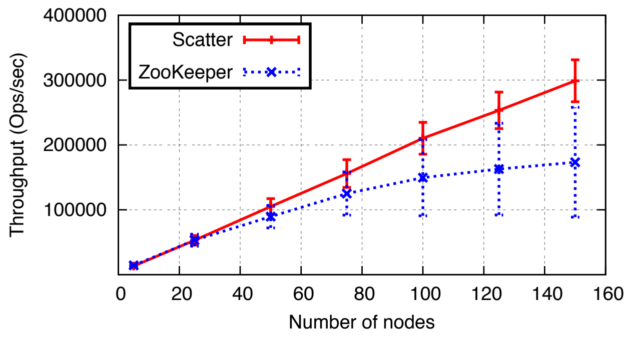


Figure 16: Comparison of Scatter with a system that composes multiple ZooKeeper instances. The figure provides the throughput of the two systems as we vary the number of nodes.

the keyspace partitioning was derived based on historical workload characteristics, the inability to adapt to dynamic hotspots in the access pattern limits the scalability of the ZooKeeper-based system. Further, the variability in the throughput also increases with the number of ZooKeeper instances used in the experiment. In contrast, Scatter's throughput scales linearly with the number of nodes, with only a small amount of variability due to uneven group sizes and temporary load skews.

8. RELATED WORK

Our work is made possible by foundational techniques for fault tolerant distributed computing such as Paxos [18], replicated state machines [33], and transactions [20]. In particular, our design draws inspiration from the implementation of distributed transactions across multiple replication groups in Viewstamped Replication [25].

A number of recent distributed systems in industry also rely on distributed consensus algorithms to provide strongly consistent semantics — such systems provide a low-level control service for an ecosystem of higher-level infrastructure applications. Well-known examples of such systems include Google's Chubby lock service [2] and Yahoo's ZooKeeper coordination service [14]. Scatter extends the techniques in such systems to a larger scale.

At another extreme, peer-to-peer systems such as distributed hash tables (DHTs) [26, 30, 22, 29] provide only best-effort probabilistic guarantees, and although targeted at planetary scale have been found to be brittle and slow in the wild [27, 28]. Still, the large body of work on peer-to-peer system has numerous valuable contributions. Scatter benefits from many decentralized self-organizing techniques such as sophisticated overlay routing, and the extensive measurements on workload and other environmental characteristics in this body of work (e.g. [11]) are invaluable to the design and evaluation of effective policies [23].

One recent system showing that DHTs are a valuable abstraction even in an industrial setting is Amazon's Dynamo [10], a highly available distributed key-value store supporting one of the largest e-commerce operations in the world. Unlike Scatter, Dynamo chooses availability over consistency, and this trade-off motivates a different set of design choices.

Lynch et al. [21] propose the idea of using state machine replication for atomic data access in DHTs. An important insight of this theoretical work is that a node in a DHT can be made more robust if it is implemented as a group of nodes that execute operations atomically using a consensus protocol. An unsolved question in the paper is how to atomically modify the ring topology under churn, a question which we answer in Scatter with our principled design of multi-group operations.

Motivated by the same problems with large scale DHTs (as discussed in Section 2), Castro et al. developed MSPastry [4]. MSPastry makes the Pastry [30] design more dependable, without sacrificing performance. It does this with robust routing, active probes, and per-hop acknowledgments. A fundamental difference between MSPastry and Scatter is that Scatter provides provable consistency guarantees. Moreover, Scatter's group abstraction can be reused to support more advanced features in the future, such as consistency of multi-key operations.

Although we approached the problem of scalable consistency by starting with a clean slate, other approaches in the literature propose mechanisms for consistent operations layered on top of a weakly-consistent DHT. Etna [24] is a representative system of this approach. Unfortunately such systems inherit the structural problems of the underlying data system, resulting in lower object availability and system efficiency. For example, inconsistencies in the underlying routing protocol will manifest as unavailability at the higher layers (see Figures 13(b) and 14(b)).

9. CONCLUSION

This paper presented the design, implementation and evaluation of Scatter — a scalable distributed key-value storage system that provides clients with linearalizable semantics. Scatter organizes computing resources into fault-tolerant groups, each of which independently serve client requests to segments of the keyspace. Groups employ self-organizing techniques to manage membership and to coordinate with other groups for improved performance and reliability. Principled and robust group coordination is the primary contribution of our work.

We presented detailed evaluation results for various deployments. Our results demonstrate that Scatter is efficient in practice, scales linearly with the number of nodes, and provides high availability even at significant node churn rates. Additionally, we illustrate how Scatter provides tunable knobs to effectively adapt to the different deployment settings for significant improvements in load balance, latency, and resilience.

Acknowledgments.

This work was supported in part by grant CNS-0963754 from the National Science Foundation. We would like to thank Vjekoslav Brajkovic and Justin Cappos for their contributions to earlier versions of Scatter. We would also like to thank our shepherd Ant Rowstron and the anonymous reviewers for their feedback.

10. REFERENCES

[1] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proc. of CIDR*, 2011.

[2] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proc. of OSDI*, 2006.

[3] F. Bustamante and Y. Qiao. Friendships that last: Peer lifespan and its role in P2P protocols. In *Proc. of IEEE WCW*, 2003.

[4] M. Castro, M. Costa, and A. Rowstron. Performance and dependability of structured peer-to-peer overlays. In *Proc. of DSN*, 2004.

[5] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos Made Live: An Engineering Perspective. In *Proc. of PODC*, 2007.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, 26(2), 2008.

[7] J. Chu, K. Labonte, and B. N. Levine. Availability and Locality Measurements of Peer-To-Peer File Systems. In *Proc. of ITCom*, 2002.

[8] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.*, 1:1277–1288, August 2008.

[9] J. Dean. Large-Scale Distributed Systems at Google: Current Systems and Future Directions, 2009.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proc. of SOSP*, 2007.

[11] J. Falkner, M. Piatek, J. P. John, A. Krishnamurthy, and T. Anderson. Profiling a Million User DHT. In *Proc. of IMC*, 2007.

[12] M. J. Freedman, K. Lakshminarayanan, S. Rhea, and I. Stoica. Non-transitive connectivity and DHTs. In *Proc. of WORLDS*, 2005.

[13] P. K. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, Modeling, and Analysis of a Peer-to-Peer File-Sharing Workload. In *Proc. of SOSP*, 2003.

[14] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-Free Coordination for Internet-scale systems. In *Proc. of USENIX ATC*, 2010.

[15] J. Yang and J. Leskovec. Temporal Variation in Online Media. In *Proc. of WSDM*, 2011.

[16] J. Kubiatowicz, D. Bindel, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proc. of ASPLOS*, 2000.

[17] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *Proc. of WWW*, 2010.

[18] L. Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2), 1998.

[19] L. Lamport, D. Malkhi, and L. Zhou. Reconfiguring a State Machine. *ACM SIGACT News*, 41(1), 2010.

[20] B. W. Lampson and H. E. Sturgis. Crash recovery in a distributed data storage system. Technical report, Xerox Parc, 1976.

[21] N. A. Lynch, D. Malkhi, and D. Ratajczak. Atomic Data Access in Distributed Hash Tables. In *Proc. of IPTPS*, 2002.

[22] P. Maymounkov and D. Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Proc. of IPTPS*, 2002.

[23] M. Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10), 2001.

[24] A. Muthitacharoen, S. Gilbert, and R. Morris. Etna: A fault-tolerant algorithm for atomic mutable DHT data. Technical Report MIT-LCS-TR-993, MIT, June 2005.

[25] B. M. Oki and B. H. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proc. of PODC*, 1988.

[26] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A Scalable Content-Addressable Network. In *Proc. of SIGCOMM*, 2001.

[27] S. Rhea, B. Chun, J. Kubiatowicz, and S. Shenker. Fixing the Embarrassing Slowness of OpenDHT on PlanetLab. In *Proc. of WORLDS*, 2005.

[28] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. In *Proc. of USENIX ATC*, 2004.

[29] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A Public DHT Service and Its Uses. In *Proc. of SIGCOMM*, 2005.

[30] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of Middleware*, 2001.

[31] S. Sankararaman, B.-G. Chun, C. Yatin, and S. Shenker. Key Consistency in DHTs. Technical Report UCB/EECS-2005-21, UC Berkeley, 2005.

[32] S. Saroiu, P. Gummadi, and S. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proc. of MMCN*, 2002.

[33] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4), 1990.

[34] S. Sen and J. Wang. Analyzing Peer-to-Peer Traffic Across Large Networks. *IEEE/ACM Transactions on Networking*, 2004.

[35] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. Technical Report MIT-LCS-TR-819, MIT, Mar 2001.

[36] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions on Networking*, 11(1), 2003.