Harmony: Consistency at Scale

Ivan Beschastnikh Lisa Glendenning Arvind Krishnamurthy
Thomas Anderson
{ivan,lglenden,arvind,tom}@cs.washington.edu

University of Washington

University of Washington Technical Report UW-CSE-10-09-04
September 2010

Department of Computer Science & Engineering
University of Washington
Box 352350
Seattle, Washington 98195-2350
http://www.cs.washington.edu

Table of Contents

Harmony: Consistency at Scale

Ivan Beschastnikh Lisa Glendenning Arvind Krishnamurthy
Thomas Anderson
{ivan,lglenden,arvind,tom}@cs.washington.edu *

University of Washington

University of Washington Technical Report UW-CSE-10-09-04

September 2010

Abstract

Robust and highly distributed storage systems are critical to the operation of both cloud services and P2P networks. This paper describes Harmony, a highly scalable and highly robust distributed storage system that provides clients with linearizable consistency semantics. Harmony's design incorporates techniques from both strictly consistent systems with limited scalability and highly scalable systems with weak guarantees, to provide the best of both worlds. A core contribution of our work is a novel design pattern for building consistent and scalable services through a network of self-organizing replicated state machines that cooperatively maintain shared state. By separating mechanism from policy, Harmony can adapt to a range of environments, such as P2P and enterprise cluster settings. Our prototype demonstrates low latency in single cluster environments, high throughput for WAN clusters, and high availability in the face of system instabilities in P2P settings.

1 Introduction

Nodes in a distributed system often coordinate by observing, acting on, and updating shared state. In wide-area peer-to-peer (P2P) settings, key-value stores have been used to build file systems, archival stores, web caches, application-level multicast systems, and rendezvous systems. Within datacenters, entire software stacks are built on top of key-value storage services [8] and synchronization primitives [2, 13].

To satisfy the needs of demanding P2P and cloud applications, we argue that shared storage should be designed to meet three requirements. First, the platform should be highly scalable, to handle the load from millions of clients and to dynamically accommodate fluctuations in load. Second, the storage system should be resilient to a variety of failure scenarios, as large scale systems typically experience a steady stream of server and network outages. Third, systems that guarantee stronger consistency semantics for stored data are conceptually easier for the application programmer to understand and use, significantly easing the developer's burden.

While existing systems partially satisfy some of these requirements, none provide all three of scalability, availability, and consistency. On one hand, coordinating data stores have been built to meet the availability and consistency requirements for applications in the data-center, but they often fail to provide a high degree

*This work was supported by NSF CNS-0963754.

of scalability and throughput. For instance, Google's Chubby is a centralized locking service that uses a single modestly sized group of nodes to implement locks and provide consistent storage by using distributed consensus techniques [2]. However, Chubby's scalability is severely limited as a single Chubby cell is designed to support only a few thousand operations per second. Each individual application has to therefore instantiate its own version of a Chubby cell, even though a single scalable instance would be easier to manage and more resource efficient [7]. On the other hand, DHTs successfully scale to millions of nodes, but most DHTs provide only weak, best effort guarantees regarding consistency of stored data. Consider for instance the Kademlia DHT [24] used to coordinate swarms within BitTorrent and Amazon's Dynamo [8] that is used by various e-commerce applications. Both do not support strict consistency: a read of a key's value after a write to the key is not guaranteed to return the new value. These weak consistency semantics pass a significant burden to the client, with applications having to explicitly manage data replication, ensure data availability, and reconcile conflicting replica values.

In this paper we demonstrate that these scalability, availability and consistency are not mutually exclusive. We present the design and evaluation of Harmony – a distributed system that provides a shared namespace for objects with linearalizable consistency semantics. Harmony synthesizes techniques from a spectrum of distributed storage systems. It adopts the replicated state machine (RSM) model of distributed computing to achieve high availability and consistency, as well as the self-organizing techniques of overlay systems to achieve scalability and robustness to churn.

A key contribution of this work is a design pattern for building consistent and scalable services through the use of a network of self-organizing RSMs that cooperatively manage shared state and maintain system integrity. Our design separates mechanism from policy to enable us to adapt Harmony to a range of environments. We present experimental results that evaluate the performance, availability, and consistency properties of Harmony in single cluster, multi-cluster, and P2P settings, compare it to state of the art alternatives (ZooKeeper and OpenDHT), and demonstrate that Harmony provides high availability and scalability while adjusting to system instabilities.

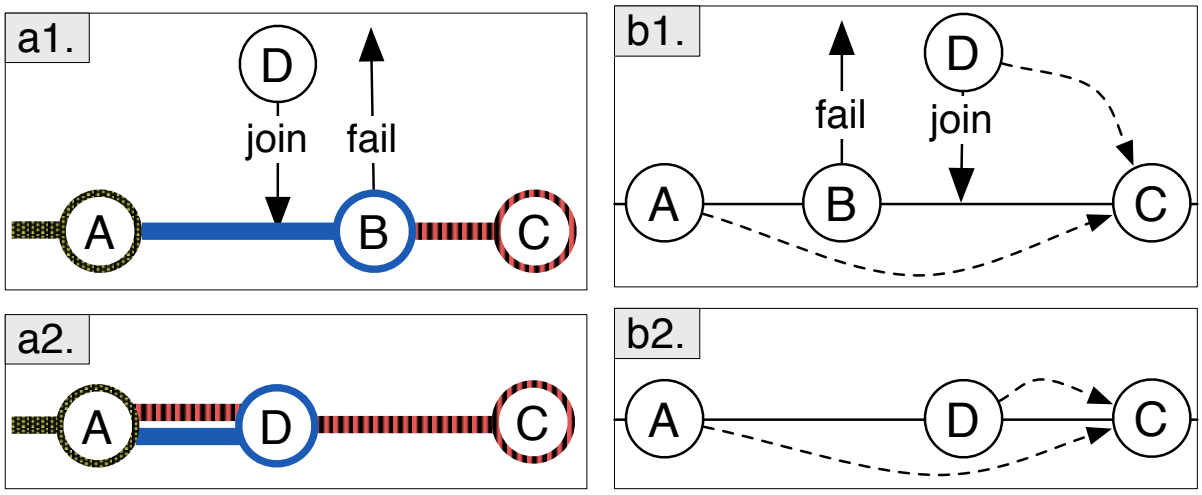
2 Harmony Overview

Harmony provides a shared namespace for objects. In this section we overview the goals and challenges that shaped Harmony's design. Harmony's design had three goals: (1) provide clients with linearalizable consistency semantics when operating on stored objects, (2) scale to large numbers of nodes, and (3) provide clients with high availability and performance.

Harmony uses a circular namespace and organizes nodes into *groups*, each of which is responsible for a contiguous segment of the namespace with each name managed by exactly one group. Groups use RSM techniques to organize themselves and provide a persistent and reliable storage abstraction that ensures linearizable consistency semantics for a set of names. Harmony groups support a client interface that allows for atomic access and update of the stored values.

Harmony groups adapt to node churn and failures by using self-reorganizing techniques borrowed from the DHT literature. For instance, a group will reconfigure itself by accepting new nodes to *join* the group in response to node failures. Groups that exceed a certain size may *split* into two adjacent smaller groups to maintain efficiency. Two adjacent groups may *merge* into one group to maintain data availability when the risk of data loss becomes high due to failed nodes. A client may send its operation to any group in Harmony; the system internally routes a client's operation to the appropriate group.

Though Harmony's design leverages prior work on self-organizing DHTs, we cannot employ these mechanisms directly as they often sacrifice structural consistency for performance *by design*. Similarly,



(a) Key Assignment Violation (b) Routing Violation

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

a straightforward realization of the RSM techniques would result in decreased availability and performance. In this section, we outline four design challenges associated with the basic approach outlined above.

Name Assignment Consistency: A basic correctness property is for each name to be managed by exactly one group. We refer to this property as *name assignment consistency*. This property is violated when multiple groups claim ownership over the same name or when a name is not assigned to any group in the system. 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 [34]. Figure 1(a) illustrates how such a scenario may occur in the context of a Chord-like system although these issues are known to effect all types of self-organizing systems in deployment [9]. 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 Consistency: Another basic correctness property requires the system to maintain consistent routing entries between groups so that the system can route client operations to the appropriate group. 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 [36]. Figure 1(b) illustrates how a routing inconsistency may occur when node events 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.[1] In this scenario, messages routed through A to

[1]When a successor fails, a node uses its locally available information to set its successor pointer to the failed node's successor.

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 [36]. More generally, such routing inconsistencies jeopardize connectivity and may lead to system partitions.

Availability: A practical system must overcome numerous issues that threaten to make the system unavailable. Harmony must cope with node failures and catastrophic group failures to provide high availability. If nodes in a single group fail rapidly, the data maintained by the group is in jeopardy of being lost and the namespace maintained by the group might become unavailable. To accommodate node churn Harmony groups *reconfigure* to exclude failing nodes, and to include new nodes. However, reconfiguration alone is insufficient – node failures could be both planned or unplanned events, and could occur when the groups are reconfiguring themselves or coordinating with other neighboring groups as part of a *split* or *merge* operation. Harmony needs to exhibit liveness in the face of pathological failures and should make progress on reconfigurations even when there are simultaneous failures.

Performance: Our RSM design is based on Paxos [15], which uses a two phase protocol for achieving quorum consensus. Paxos, however, is an expensive protocol. If every operation in Harmony were to involve a distributed consensus round over a set of nodes and if most group operations were routed through a designated leader, the resulting latency would be unsuitable for most applications. Also, high inter-node latency due to a wide geographical dispersal of nodes could further decrease performance. Finally, as a group accepts more nodes to improve availability, it becomes less efficient and its throughput diminishes. To match state of the art systems, Harmony groups must provide high throughput and low operation latency.

Harmony provides name consistency and routing consistency by ensuring that all group reconfiguration operations are atomic. When multiple groups are involved in a reconfiguration event, Harmony guarantees that the reconfigurations have transactional properties, i.e., all affected groups atomically transition to the respective new configurations (see Section 4). Harmony increases availability by allowing any node in a group to resume previously initiated group reconfigurations and by using a centralized service to detect network partition failures (see Sections 4 and 6). Finally, we extend the basic RSM design described in Section 3 to include numerous optimizations that allow for nodes in a group to operate concurrently, with many operations performed locally and the rest incurring only 1 RTT delay to a quorum, and by placing nodes with low inter-connection latency in the same group (see Sections 5 and 7).

3 Replicated State Machines

An established approach for providing a consistent and available distributed service is to use an RSM system architecture, and advances in the efficiency of consensus protocols have improved the practicality of this approach. In this section we describe our implementation of an RSM as a fundamental building block of Harmony. Section 3.1 overviews the Paxos protocol [15], and Section 3.2 outlines the RSM approach [35]. Section 3.3 introduces our novel method to *stop* a Paxos-based RSM efficiently and our implementation of RSM reconfiguration.

3.1 Paxos Overview

The consensus problem requires a set of processes to choose a single value. This paper considers the consensus problem under the asynchronous, fail-stop model of distributed computation in which processes

fail by stopping but can recover, and messages can take arbitrarily long to be delivered, can be delivered out of order, can be lost, but are not corrupted.

A consensus protocol can be characterized in terms of both *safety* and *liveness*. A solution that preserves safety must never allow two different values to be chosen despite failures. A solution exhibits liveness if it must eventually choose a value when enough processes are non- faulty and can communicate with one another.

Paxos is a consensus protocol that is always safe and is live given eventual partial synchrony and a non-pathological interleaving of concurrent proposals. Paxos can be expressed in terms of three sets of agents: *proposers* that propose values, *acceptors* that choose a single value, and *learners* that learn what value has been chosen. An instantiation of Paxos consists of one or more *ballots*, each of which proceeds in two *phases*:

Phase 1:

(a) A proposer selects a unique *ballot number* b and sends a *prepare* request with number b to some set of acceptors.

(b) If an acceptor receives a prepare request with number b greater than that of any prepare request to which it has already responded, then it responds to the request with a *promise* (i) not to accept any more proposals numbered less than b, and (ii) with the highest-numbered proposal (if any) that it has accepted.

Phase 2:

(a) If the proposer receives a response to its prepare request numbered b from a *quorum* of acceptors, then it sends an *accept* request to each of those acceptors for a proposal numbered b with a value v, where v is the value of the highest-numbered proposal that could have been chosen among the responses, or is any value if the responses reported no proposals.

(b) If an acceptor receives an accept request for a proposal numbered b, it accepts the proposal unless it has already responded to a prepare request having a number greater than b.

A value v is chosen if and only if a quorum of acceptors vote to accept v in phase 2 in some ballot. A quorum is typically defined as a majority set of acceptors.

In an asynchronous system, an instance of Paxos is not guaranteed to terminate, i.e. choose a value, when there are concurrent proposals. To expedite progress, a distinguished proposer termed a *leader* is selected, and all proposals are submitted through the leader. During a Paxos instance, a new leader may be *activated* at any time. When a new leader is activated, it chooses a ballot number b that it believes is larger than that of any previously started ballot, and initiates phase 1 of ballot b. Even with an active leader, it is still always safe for another proposer to execute a higher numbered ballot. Indeed, this is a necessary condition for liveness in the presence of leader failure.

3.2 Replicated State Machine Approach

By definition, any instantiation of Paxos decides (at most) one value. To choose many values a system executes multiple, logically distinct *instances* of Paxos, each of which follows the algorithm outlined above. Depending on the required semantics of the system, instances may be executed concurrently, or in a partially or totally ordered sequence. We use this approach in Harmony to implement the RSM model by executing a sequence of instances of a consensus protocol such that instance i chooses the i-th state machine command.

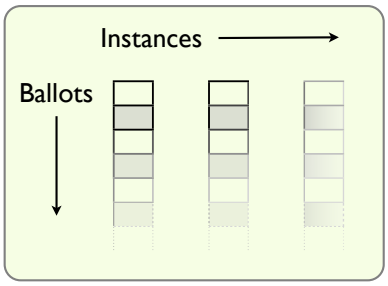


Figure 2: An RSM execution progresses both horizontally as a sequence of consensus instances and vertically with increasing ballot numbers.

Processes in the system typically perform one or more of the following roles: *clients* submit commands to be proposed and chosen; *learners* learn and execute the ordered sequence of chosen commands; and *leaders* and *acceptors* order commands using a consensus protocol. Note the distinction between *choosing* and *executing* a command. A typical RSM implementation allows multiple commands to be chosen concurrently, but does not allow command i to be executed until all commands less than i have been chosen.

Key to the efficiency of a Paxos-based RSM is the observation that ballot phases can be performed simultaneously for all instances. When a new leader is activated for the RSM, it executes phase 1 of a higher numbered ballot for all instances whose outcome it does not already know. A common pattern of messages for executing an RSM command is: (1) a client submits a command c to an active leader; (2) the leader selects the next instance i and sends ballot b, phase 2a messages with the value c to some quorum of acceptors; (3) the acceptors send ballot b, phase 2b messages with the value c to some set of learners.

3.3 Dynamic Configuration

In the classic consensus problem, there is a fixed set of n processes, at most f of which can fail. In practice, processes not only leave the system, but also join the system to proactively maintain system availability and to scale out in response to load. We call the set of processes executing an RSM the *configuration* of the RSM, and the process of changing the configuration of a system *reconfiguration*.

There are several existing methods for reconfiguring an RSM [17, 20]. We chose an approach that combines the executions of a sequence of separate, non-reconfigurable state machine instances. Reconfiguration then involves three largely orthogonal concerns:

(i) *Stopping* the current RSM r_j. This requires reaching consensus on the chosen commands and ensuring that no other commands can be chosen.
(ii) *Choosing* the configuration of the next RSM.
(iii) *Starting* the next RSM r_{j+1} such that its initial state is the final state of RSM r_j.

3.3.1 Stopping an RSM

We designed and implemented a novel technique for stopping Paxos-based RSMs, which we term *convergence*.

Condition 3.1. An RSM instance is composed of an unbounded set of consensus instances I executed by some configuration P. An RSM instance has *converged* if and only if some value v_i has been chosen for every instance i in I.

The key technique we use for convergence is to execute a phase of the Paxos protocol simultaneously across some subset of I. This is a generalization of the common method of leader activation defined in the previous subsection. When a process $p^* \in P$ is *activated* for some subset I^* of I, it chooses a ballot number b^* that it believes is larger than that of any previously started ballot in I^*, and executes phase 1 of ballot b^* for all instances in I^*. Note that in an implementation of this technique, an unbounded number of Paxos messages can usually be represented efficiently in a single physical message for each sender-receiver pair.

To overview our method in the simplest case, we assume that there are no faulty processes in P and that p^* receives responses from all P. After receiving all phase 1b messages, p^* knows, for each $p \in P$ and each $i \in I^*$, the highest ballot that p has promised or accepted in i. From this information, p^* can deduce a number of facts about this snapshot of the global state of the RSM, for example, for each $i \in I^*$: (a) whether some value v_i has been chosen, and (b) if any ballots greater than b^* have been executed.

Due to the non-blocking nature of Paxos, an individual process is never guaranteed complete or coherent knowledge of the system; however, Paxos establishes some conditions for which, if the condition holds in some snapshot of the instance, the condition will hold for all following snapshots of the instance. For example, the safety property of Paxos guarantees that if a process learns at some time t that a value v_i has been chosen for an instance i, then it knows that, for all time greater than t, v_i will be chosen for i.

This property of Paxos motivates our use of the term convergence. If a Paxos instance continues to progress by executing ballots during which there exists a non-faulty quorum of processes, then the instance will eventually reach the state where (i) a value is chosen, and (ii) all non-faulty processes in the instance have learned this value – i.e., the local knowledge of all processes converges to the same state.

The insight behind our mechanism for RSM convergence is that phase 2, as with phase 1, can be executed simultaneously across some subset of I. To implement convergence, we define a $noop$ RSM command, the execution of which has no effect on the state of the RSM.

Let's assume that following the execution of phase 1 for ballot b^*, p^* doesn't learn of any higher-numbered ballots. p^* can now execute phase 2 of ballot b^* for all instances $i \in I^*$ by sending an accept messages to all P as follows:

(i) If some v_i may have been chosen for i, the message proposes v_i.
(ii) If no values have been accepted for i, the message proposes $noop$.
(iii) If some set of values have been accepted but not chosen for i, the message can propose either one of these values or $noop$.

We again consider the behavior of this algorithm in the simplest case of no failures and a single active proposer. When p^* receives all phase 2 responses for b^*, a value has been chosen for all I^*. If $I^* = I$, then, by Condition 3.1, the RSM instance has converged. Thus in the simple case convergence completes in one message round.

Since our method of convergence is executed within the RSM abstraction using the Paxos protocol, it shares the safety and liveness properties of Paxos, and many known techniques for efficiency apply.

Safety Say, for example, that there exist some number $\leq f$ of faulty processes in P. The RSM can still reach convergence with some quorum. If a formerly faulty processor recovers after convergence, it will not be able to introduce new RSM commands as all instances have chosen a value, but it will be able to learn the final state of the RSM to update its local state. As another example, say we have two concurrent proposers, one that is proposing normal client commands and one that is executing convergence. Our convergence mechanism is agnostic both to which values are chosen, and to whether convergence happens in one ballot or over some number of ballots interleaved with competing ballots. The only condition necessary for correctness is that some value is chosen for each instance in I. If there is a data race between

a $noop$ command and some other command in some instance, at most one will be chosen. An application implemented as a convergable RSM can choose a policy for such cases – e.g., a process won't propose any client commands once it has seen a $noop$ proposal.

Liveness As with other RSM commands, the progress of convergence depends on (a) the cooperation of proposers to defer when collision is detected, and (b) a failure detecter to activate a distinguished proposer if the current proposer of convergence fails. Thus, standard RSM techniques apply to convergence, with perhaps one extra condition. If an active leader fails while proposing a client command, the application semantics likely allow the RSM to "drop" this command until the client resubmits it; however, if some process partially executes convergence before failing, it is probably most efficient for the next active leader to continue convergence.

Efficiency In practice, the infinite set I can be encoded as a finite but growing set I', and the process of convergence establishes some upper bound on I'. Yet a straightforward implementation of the convergence mechanism described in this section in the simple case would require two rounds of $O(|P|)$ messages, each of which contains information linear in $|I'|$. The complexity of this mechanism can be reduced in the common case without compromising safety by assuming that some quorum of P is synchronized with respect to their view of $|I'|$.

Related Work Our method builds on ideas from previous proposals for reconfiguration of RSMs [17]. An obvious approach is to implement reconfiguration as an RSM command such that an RSM execution is a continuous timeline of different configurations. A significant drawback of this approach is that it prohibits proceses from executing instance i unless they know the values that have been chosen for all instances less than i, since some previous instance could have been a reconfiguration.

This performance penalty can be ameliorated such that a reconfiguration command chosen for i takes effect some α commands later [20]; however, this optimization does not eliminate the root cause of the blocking behavior of this approach – imposing unnecessary and restrictive serialization on the actions of an RSM.

Approaches which implement reconfiguration by stopping non-reconfigurable RSMs expose more opportunities for inherent concurrency. One approach for stopping an RSM implements *stop* as an RSM command which must be chosen for some i and the effect of which is that any following commands have no effect (are chosen but not executed) – this approach has similar properties to the reconfigurable RSM approach as well as the undesirable necessity of modifying the RSM abstraction with unintuitive behavior. A related technique is to choose some i and propose $noop$s for all instances $\geq i$. Our mechanism is a generalization of this technique and does not require i to be explicitly chosen by a process – the upper bound i after which all instances have chosen *noops* is implicitly determined as a result of executing convergence.

To address limitations of the above approaches, Lamport proposed Stoppable Paxos [16], which implements a *stop* RSM command but guarantees that no value can be chosen for following instances. This guarantee is provided by restricting the behavior of proposers to not propose any command for instance i if stop *may* have been chosen for some instance less than i. Convergence is conceptually simpler, and requires no modifications to standard consensus mechanisms.

3.3.2 Starting an RSM

In Harmony, choosing the configuration of each RSM instance is managed by a higher level consensus abstraction, so we defer the details of this mechanism to Section 4.

The remaining concern to address in this section is initializing the state of state machine r_{j+1} from the final state of state machine r_j. First, we recall that the abstraction provided by an RSM is that given some initial state s_j it executes a sequence of k commands to produce state s_{j+k}. Given that state machine r_j has terminated, it follows that the final state of r_j either (i) has been determined by the processes of r_j, or (ii) can be determined by any process in the system by learning the initial state of r_j and the complete sequence of chosen commands – assuming that replaying the execution of these commands is deterministic.

Once the processes of state machine r_{j+1} have learned their initial state, they can begin executing commands. Since we have separated *choosing* and *execution* of commands, state machine r_{j+1} can concurrently learn its initial state while choosing new commands to minimize the observable unavailability of the system.

4 Harmony: A Naming Service

In this section, we discuss how we use RSMs as building blocks for scalable and decentralized distributed systems. Inspired by the lean interface of highly distributed lookup systems such as Chord [36], we agree with the underlying premise that a shared namespace is an essential component of many highly scalable distributed systems. This section details how we designed and implemented a consistent and scalable *naming service* that addresses anomalies caused by weak atomicity in existing systems as discussed in Section 2.

4.1 The Naming Problem

- i.e., We define the naming problem as a mapping between *namespaces* (sets of names) and *groups* (sets of processes) for a system. We implemented a naming service that maintains the invariant that every namespace and every group is in exactly one pair in a matching, which we can define more formally as:

Condition 4.1. For some *logical time* τ, given a partition $N(\tau)$ of the complete namespace N such that members of $N(\tau)$ are namespaces, and a partition $P(\tau)$ of P such that members of $P(\tau)$ are groups, there exists a *perfect matching* $M(\tau)$ between the members of $N(\tau)$ and the members of $P(\tau)$.

We implement namespaces as contiguous intervals (using modular arithmetic) of the fixed size integral range $[0, 2^k - 1]$. To implement the naming service, for each pair $(n,\ g)$ in a given matching $M(\tau)$, we create an RSM that holds the state n and is executed by the processes in g.

We store group membership information in a distributed data structure to minimize coupling between groups – specifically, we chose to use a *ring*, which is a circular doubly-linked list that is aligned with N. We implement this ring by expanding the state of the RSM for group g to include references to g's *predecessor* and *successor*. Our system now has the additional property that *any group in the system can locate any other group in the system by traversing the ring*. This property can be expressed more formally as:

Condition 4.2. A distributed location topology is *connected* at time τ if and only if for all groups u, v in $P(\tau)$, there exists a path from u to v.

A reference to group u stored in group v must provide the following information: (a) the relative orientation of u to v with respect to N (predecessor or successor), and (b) how to communicate with u. Deriving a representation of (a) is straightforward. A sufficient, but perhaps conservative, encoding of (b) is the set of

identifiers for all the processes in u. This encoding introduces an additional *local* invariant that is sufficient to satisfy the global Condition 4.2:

Condition 4.3. Every group has a consistent view of the membership of its preceding and succeeding groups.

What we have described in this section so far is a snapshot of our system at some point in logical time τ. For the remainder of this section, we present how we augmented our design using an innovative approach to accomodate a highly dynamic environment in which (i) processes leave and join groups, and (ii) entire groups are both created and disbanded. Further, we show how our approach maintains our invariants across such transient instabilities.

4.2 Coordination of Groups

If the membership of a group u changes without synchronously updating the view of u's predecessor and successor, then Condition 4.3 is violated. Therefore, we require a mechanism that updates the membership of u and any distributed references to u *atomically* and *consistently*. This is a problem that has been identified (e.g. [19],[21]), but to the best of our knowledge has not previously been solved in practice for a highly distributed system.

Before describing our solution, we note that since real-time synchrony of distributed actions can't be guaranteed, that the global state of a distributed system is only consistent in the context of logical time [14]; so we introduce a logical clock to the state of a group. As described in Section 3, we implement a reconfigurable RSM by a sequence of state machines with bounded executions; so we advance the logical time of a group when the group RSM reconfigures. We also revise our representation of a reference to a group to include the logical time of the group in addition to membership information, and we can now refine our invariants on observing global state at some point τ in global logical time to be the result of a consistent distributed snapshot on the system [23].

In Harmony, all operations that atomically mutate state that is *shared across groups* are structured as *transactions*. We illustrate this transactional approach by detailing our technique for atomically updating the shared view of a group.

4.2.1 Coordinating Ring Updates

For group u, let the predecessor of u be w and let the successor of u be v. To modify the membership of u, we must also modify (a) the successor reference in w, and (b) the predecessor reference in v.

Up to now, we have described the state that is held by a group RSM, but we have not discussed the set of *commands* that this RSM exports to manipulate its state. To define the group RSM commands, we first reformulate the state of a group in pseudocode:

```
type Group:
    Set<Process> members
    Group* predecessor
    Group* successor
    Namespace namespace
    Time time
```

We informally denote a command to get the field of an instance as `object.field`, and to set the value of a field by `object.field = value`. Let t_u be the value of `u.time` when it initiates a view update operation,

and let $t'_u > t_u$ be the value of `u.time` when the update completes. Similarly, let p_u and p'_u be the value of `u.members` before and after the operation. Then the operation encompasses the following commands:

```
u.members = p'_u
u.time = t'_u
w.successor.members = p'_u
w.successor.time = t'_u
v.predecessor.members = p'_u
v.predecessor.time = t'_u
```

We now describe how to execute these commands using a *two phase commit* (2PC) protocol [18].

To implement the 2PC protocol, we add a write-ahead log field, `log`, to each group RSM. We overview how we implemented this log mechanism to highlight some of the considerations in extending techniques used by processes to execute distributed transactions to RSMs.

Executing a transaction appends two entries to the log of each participating group: *transaction begin* and *transaction end*, corresponding to phases 1 and 2 of the protocol. Each entry contains sufficient information such that any process in the system can take over the coordinator role of the transaction, and messages re-executing some part of the protocol are idempotent.

This transaction state includes (a) a unique transaction identifier, (b) the set of commands forming the transaction, and (c) whether the transaction was committed or aborted. When a group records a transaction entry, it includes its participant vote of commit or abort in the log. How this vote is determined is discussed later.

For a group view update, the coordinating group of the transaction is the group u whose membership is being updated. A process in the system – normally the active leader of the coordinating group – coordinates all phases of the protocol using the following sequence:

1. Append the `begin` entry to `u.log`.
2. Concurrently submit the `begin` entry to v and w.
3. If u, v and w all vote to commit, u reconfigures.
4. Concurrently submit a `end` entry to u, v and w that includes whether the transaction was committed and any modifications to the set of transaction commands.

In the following paragraphs, we consider important properties of this technique along with related clarifications.

Atomicity The commands associated with a transaction are only executed by a group after the end entry has been executed with a commit outcome. All of the commands are then batched into the execution of a single RSM command, updating the local state of each group atomically. Note that there may be an observable real-time lag between the participating groups updating their local state, but an up-to-date value can always be observed by a client by querying all three groups and comparing logical timestamps.

Availability This mechanism is copy-on-write, so it is safe for clients to continue to observe the view of group u at t_u during the transaction.

Isolation For brevity, we simplify the topic of concurrent transactions by requiring complete mutual exclusion between committing transactions. We enforce this restriction by having a group vote to abort a transaction if there exist any transaction begin entries in its log for which there is not a corresponding transaction end entry. This avoids deadlock, but requires groups to retry transactions. We note however that many

of the techniques that have already been developed to improve transaction throughput and latency should be applicable.

Liveness When a new leader is activated for a group, it queries the log to determine if there are any outstanding transactions. If the leader's group is coordinating an outstanding transaction, then the lader executes the next step in the sequence. Otherwise, the leader queries the other groups in the transaction for the corresponding end entry.

4.3 Remapping the Namespace

A straightforward extension of the view update operation described above would be to batch updates to two neighboring groups u, v in the same transaction, for example, to migrate processes between groups for overall load balancing.

Implementing this operation requires just a few modifications to the algorithm described for a single view update. First, four groups must participate in the transaction: u, v, and the neighbors of u and v. Second, both u and v must learn the outcome of phase 1 to reconfigure concurrently in step 3 of the sequence. Third, phase 2 is not executed until both u and v have completed reconfiguration.

Now we discuss another powerful application of our approach – creating and disbanding entire groups to modify the global mapping of namespaces to groups. One could also devise a complementary operation for neighboring groups to renegotiate the namespace boundary between them, but we'll restrict the scope of this subsection to two operations: *split*, which partitions the state of an existing group into two groups, and *merge*, which creates a new group from the union of the state of two neighboring groups.

Atomicity is required of modifications to namespace partitioning and mapping in order to maintain Condition 4.1, so we structure namespace remappings as transactions. The main difference between the split operation and the view update operation is that in step 3, first two new groups u_1 and u_2 are created. There is no need for u to reconfigure – it just stops its RSM, and the RSMs of u_1 and u_2 initialize their state from the partitioned final state of u. As with the batched view update operation described earlier in this subsection, it requires the participation of u_1 and u_2 to execute the second phase.

The converse logic implements merge. During step 3, one new group is created from the union of the final states of the two merged groups.

5 Harmony: A Storage System

The previous section described how to use RSMs to design a decentralized and consistent naming service. We now reuse these techniques to add a storage service to Harmony that associates data with this namespace. Figure 3 visualizes this two-tier system. The key technique described in this section is how to apply the techniques from Section 4 to the management of multiple RSMs in a group. The system presented in this section is the system we implemented and evaluated in Section 8.

5.1 Storage Interface

An important step in designing a storage system is defining an interface, which includes both the *data model* and *data consistency semantics*. Linearizability [12] is a correctness property that is both easy to reason about and straightforward to implement with an RSM. Linearizability can be loosely paraphrased as behaving as though processes are interleaved at the granularity of a complete operation on a single object.

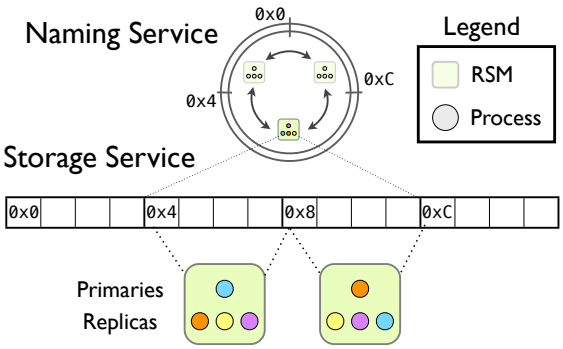


Figure 3: Example of an instantiation of the Harmony storage design for an address space with $k = 4$ and $m = 2$. There are three groups with four processes each. One group manages two segments, and the other groups each manage one. The focus is on the composition of services within one of the groups.

Our goal was to define a storage interface that was low-level and balanced expressive power and application performance. We chose a segmented memory model similar to that of Sinfonia [1], motivated by the data locality it provides.

A k bit address in our model uses an m bit prefix to identify a *segment*, and the remaining $k - m$ bits to identify an *offset* in the segment. An address names a byte, so a segment is a fixed-size, contiguous region of memory.

Informally, operations on our data model take the form of unary operators on segments, meaning that an individual operation can access any subset of exactly one segment. An operation can combine conditionals, byte literals, reads and writes to the segment, and simple arithmetic operators. A sequence of primitive operations can also be batched into a composite operation. Harmony provides linearizability consistency semantics at the granularity of an operation on a segment.

A segmented memory data model has a number of advantages over a flat-addressing memory scheme. It allows a higher-level programming model as compared to single-byte operations, and the abstraction of a fixed-size memory chunk is a familiar one. This memory model also leverages the properties of RSMs to make strong guarantees about the atomicity of batched operations on multiple objects located within the same segment. Segments provide shared-nothing partitioning, enhancing the parallelism of the system.

5.2 Storage Implementation

We implement this data model by adding an additional RSM to each group for every segment associated with the group's namespace. These RSMs share the configuration (i.e. group membership) of the namespace RSM described in Section 4, but may have different leaders to increase the overall throughput of the group. We term the active leader for the RSM associated with a particular segment the segment *primary*, and the other processes executing the RSM *replicas*.

Linearizable semantics are implemented by executing data operations as RSM commands. Lower latency per operation is achieved by submitting operations to the primary for a segment, but it is always safe to submit operations through replicas, enabling continuous availability of the storage system even as individual processes fail.

Note that our storage design introduces no additional coupling between groups, but does introduce an additional constraint that must be respected during namespace remapping (Section 4):

Condition 5.1. A namespace partition must align with segment boundaries.

The execution of namespace remapping operations on the storage system is essentially the same as for the simpler naming service, but when groups are created or destroyed they must start and stop multiple RSMs rather than just one.

5.3 Flexible Operation Semantics

A design based on the RSM approach can take advantage of the underlying properties of the consensus protocol to allow clients a high degree of control over tradeoffs between the availability, consistency, and performance of individual operations. For example, clients have the option of continuing to make progress while the system stabilizes in response to faults by temporarily using weaker operation consistency guarantees.

Relaxed Reads To illustrate some possibilities, we first differentiate between *mutating* and *non-mutating* classes of operations. Non-mutating operations can access the locally cached, but potentially inconsistent, state of any process in the RSM; however, mutating operations must always be executed as an RSM command.

Pipelining A client of the system can pipeline operations to a single segment; however, such operations are considered concurrent, and the order in which they are executed may be different from the order in which the client issued them.

Leases Leases [10] can be used to prevent processes that are not the leader from proposing any consensus values. This allows the leader to satisfy a non-mutating operation from its local cache without weakening consistency guarantees. Leases trade lower latency of reads in the common case for a longer period of unavailability in the uncommon case of lease-holder failure.

6 Availability in Harmony

In the previous section, we described how to design a storage system with strong guarantees under certain failure conditions. We now discuss mechanisms for maintaining reasonable behavior during extreme failures.

Harmony is designed to maintain high availability and to avoid data loss by merging groups that have few nodes with adjacent groups. The resilience policy used by Harmony (Section 7) uses a threshold on the number of nodes below which a group should merge. This threshold trades-off availability for performance since smaller groups are more efficient. A Harmony 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. Relaxing this constraint to provide weak consistency semantics is future work.

Harmony also deals with network partitions at a slight cost to availability. Harmony operates a *meta-group*, which is a group that maintains state on *group leases* for the entire system. The meta-group acts as a distributed failure detector, which declares groups in the partition that cannot reach the meta-group as failed. Each group in Harmony must periodically re-acquire a lease on its portion of the namespace from the meta-group. If the meta-group does not receive a lease request in time for a section of the namespace, it assigns the unleased section to the group managing the nearest section of the namespace. Due to lack of space, we only briefly overview the relevant mechanisms.

	Load Balance	Latency	Resilience
Single Cluster	✓		
Multi-Cluster	✓	✓	
P2P		✓	✓

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

The meta-group is comprised of some number of leaders from the groups making up Harmony. When a group's lease period nears expiration, the group leader contacts the meta-group to extend the lease. The leader does so by submitting a command to the meta-group. Once this command is processed, the group leader commits the new lease period information to its group. For this to work, groups mandate that the leader of the group can reach the meta-group. Since nodes making up the meta-group can fail, the meta-group may signal to some number of group leaders to join the meta-group. This occurs when the leaders contact the meta-group to extend the group lease. To make sure that group leaders can locate the meta-group, the meta-group additionally sends each group leader the current meta-group membership as part of the lease extension protocol. [2]

7 Separating Policy from Mechanism

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

In this section we first describe the policies that we have found to be effective in the three settings where we have deployed and evaluated Harmony (see Section 8). These are: (1) Single cluster: dedicated and geographically co-located cluster of machines often on the same LAN, (2) Wide-area multi-cluster: dedicated clusters at multiple locations connected by a wide-area network, and (3) P2P: non-dedicated machines that participate in peer-to-peer systems. 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.

7.1 Load Balance

In single cluster and multi-cluster settings, system load balance is an important criteria for 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 and joins the group that has processed the most client operations in the recent past.

[2] As a back of the envelope analysis consider a meta-group with 9 nodes in a single cluster setting. It has a throughput of 13,000 ops/s (Section 8) and can therefore support 13,000 groups if the group lease period is 1 second and leases expire simultaneously. A lease period of 30 seconds raises the number of groups to 390,000. With 10 nodes per group, this meta-group can support a Harmony deployment of 3.9 million nodes. This calculation indicates that the meta-group can scale to arbitrate network partition failures in large deployments.

7.2 Latency

The latency to execute RSM commands depends not only on the mode under which it is invoked (as discussed in Section 5.3) but also on the inter-node latencies that determine the cost of achieving a quorum. For multi-cluster and P2P deployments, the network latency dominates the latency of executing RSM commands. A join policy can optimize by placing nodes with low inter-connection latency into the same group. The *latency-optimized join* policy accomplishes this by having the joining node randomly select k groups and pass a `no-op` operation in all of them. The node then joins the group with the smallest command execution latency.

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 on reconfiguration, merge, and split costs in Section 8.2.

7.3 Resilience

In a P2P setting, Harmony must be resilient to node churn as nodes join and depart the system unexpectedly. To improve system resilience, we employ 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. The policy also directs new nodes to sample a random set of k groups and join the group that is most likely to fail. The 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. If multiple groups have the same expected failure probability, then the new node picks the target group with the lowest RSM command execution cost as described above, thus using latency as the secondary metric.

We now describe alternative policies for which we have no evaluation. These policies are more involved, and demonstrate the flexibility of expressing policy in Harmony to accommodate a variety of user-defined criteria.

7.4 P2P: Group Splitting for Performance

In the above policies, we did not take into account an important P2P node lifetime heuristic – an observed node lifetime correlates with remaining node lifetime [11]. In P2P environment the total node lifetime of a group and a group's inter-node latency are important properties that can guide splitting. We've designed a P2P splitting policy, *P2P-split*, that optimizes for both the performance of the groups, as well as the total lifetime of each group.

When a group splits this policy guides how nodes are assigned to the two resulting groups. The P2P-split policy improves group performance by minimizing inter-node latencies in the two resulting groups. Second, it maximizes the total lifetime of each group, which we define as the sum of all members' lifetimes. A large total lifetime suggests that the group will have fewer members leaving or failing in the near future. The first goal implies that nodes that are far apart should probably be assigned to different groups, while the second goal implies that the total lifetime of the two resulting group should be approximately equal.

The P2P-split policy is not optimal since optimizing even one of the two goals is NP-hard, let alone both of them. To begin with, consider all the nodes as unassigned and the two split groups as empty. Sort all the edges connecting two nodes in the original group by their latencies. Then consider the edges in decreasing order of their latencies. For each edge (u, v) that is considered: If u and v are both unassigned, assign the

longer-lived of u and v to the group with smaller total lifetime and assign the second node to the other group. Else if one of u and v is assigned, assign the second node to the other group. Skip the node pair if both nodes are already assigned. Finally, at any time, if any of the two split groups contains half of all the nodes, put all remaining unassigned nodes into the other group and stop. This is used to maintain equal split group sizes.

7.5 Maintaining Multi-Cluster Robustness

Harmony deployments in managed setting such as multi-cluster require particular combination of policies to maintain robustness. In this setting, we define a policy to ensure that Harmony can survive entire cluster failures, while maintaining reasonable performance.

The key to surviving cluster failures is to have high cluster diversity. We define *group cluster diversity* as the number of unique clusters that have nodes participating in the group. Our resulting group splitting policy strictly maintains this diversity for Harmony groups above a threshold and also takes into account node lifetime and inter-node latency by reusing the *P2P-split* algorithm above. Finally, note that our group split policy still operates under the constraint that the two resulting split groups have equal size.

Our resulting *multi-cluster split* policy maintains group cluster diversity while still optimizing for performance. This policy is designed as follows. We view nodes of the original group as belonging to distinct *clusters* [3]. Our goal is to maximize the number of clusters spanned by *both* groups resulting from the split. That is, producing one group with high diversity and another with small diversity worse than producing two groups with high diversity. Suppose that the original group spans cluster $C_1, .., C_m, .., C_{m+p}$ with $a_i > 1$ nodes in cluster C_i where $i \leq m$ and with only 1 node in each of the next p clusters. For each of the first m clusters C_i, we run the *P2P-split* algorithm to split the nodes in this cluster into groups of sizes b_i and c_i (i.e. $b_i + c_i = a_i$). Then we run this algorithm again to split the nodes in the remaining p clusters into groups of sizes b_0 and c_0 (i.e. $b_0 + c_0 = p$).

Finally we assign each of these b_i- and c_i-groups to one of the two resulting groups so that the two groups have the same total number of nodes (each of them takes exactly one of b_i- or c_i-group for every $0 \leq i \leq m$). This can be done because the above group splitting algorithm guarantees that for every i, $|b_i - c_i| \leq 1$.

In practice we can require that every group has a minimum cluster diversity of K. This means that at any time in the system, every group must span at least K clusters. However, the *multi-cluster split* algorithm, although optimal, is not sufficient to maintain this requirement [4].

A key to preventing such unfavorable group compositions is by defining a complementary Harmony join policy. When a group leader receives a request from a new node to join the group, it approves this request only if this join would not cause any future group split to violate the K diversity constraint. If the Harmony split threshold is $splitthresh$, and the group spans $m + p$ clusters as above, then the leader should accept a new node only if $m + (p + splitthresh - n)/2 \geq K$ and the node is diversity-optimal. For space reasons we omit the details of the proof.

Our implementations of the policies above indicate that by separating policies from mechanisms it is easy to re-purpose Harmony's algorithms towards new environments without any modification to the mechanisms. According to `sloccount`, the policy to select a random node as the leader is 8 LOC, the policy

[3] We assume that in the network topology nodes in the same cluster have small latency to each other, while nodes in different clusters have high inter-node latency.

[4] If $K = 3$ and we want to split a group of size 10 that spans 4 clusters, with the first cluster containing 7 nodes from the group and the remaining three clusters with only one node each. Then it is impossible to split this group into two groups, both of which span at least K clusters.

that selects the oldest node as the leader is 15 LOC, and the implementation of the leader selection policy above is just 46 LOC. For split, the algorithm to split a group randomly is 5 LOC, while the much more complex *P2P-split* and *multi-cluster split* algorithms are implemented in 64 LOC and 89 LOC respectively. These counts are for complete and functional implementations.

7.6 Partitioning Group Addresses

We mentioned that the group leader partitions the group's addresses among the primaries in Section 5. We did not, however, specify whether the group's addresses is partitioned evenly or how the leader chooses to assign primaries to partitions. The leader employs two pieces of information for this assignment: client workload statistics, and expected node lifetimes.

Harmony uses the policy that the constructed partitions (1) balance aggregate client request rates across the primaries (if possible) and (2) maximize data availability based on expected node lifetimes according to the popularity of the addresses.

For example, nodes that are more likely to fail (those that have been around a short time) should not be assigned popular addresses because when these nodes fail, this assignment will create unavailability for clients. Similarly, they should be given smaller addresses to reduce overhead of repartitioning. Therefore, a leader may assign an empty partition to a node if it just joined the system. Such a node will be a replica for the group's addresses, but it will not be a primary for any of the group's addresses. As the node persists in the group, it is given additional responsibilities.

P2P systems' client workloads are typically heavy-tailed. To deal with heavy-tailed workloads, Harmony load balances client requests within a single group to deal with hotspots at the group level. To balance client requests, Harmony assigns different primaries different sized segments of the group's addresses and attempts to satisfy the invariant that primaries receive the same *expected* number of client requests. The leader periodically recomputes this. Nodes report client request statistics to the leader each time the leader re-leases the primary partitions. The leader determines the appropriate partitioning of its addresses to the primaries based on reported client requests in its addresses. To assign nodes to partitions, the algorithm weighs each partition by the performance profile of each node.

8 Evaluation

We evaluated Harmony across the three diverse deployment scenarios discussed in Section 7. We used Emulab for single cluster deployments, Emulab and Amazon's EC2 for multi-cluster deployments, and PlanetLab for P2P deployments. For each environment, we present results from large scale deployments to illustrate Harmony's scalability and resilience to node churn. We also present benchmarks for a single RSM and a single group, such as the latency of executing a RSM command or the cost of performing a group merge, which illustrate performance as well as quantify Harmony's availability.

In all experiments Harmony 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 Harmony 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. We present results for client read and write operations on an address as representative of the more general non-mutating and mutating classes of operations on segments. We used Berkeley-DB for persistent disk-based storage, and a memory cache to pipeline operations to BDB in the background. In the graphs, each measurement result was repeated between a hundred thousand and

a few million times. In the group split and merge experiments, the merge and split thresholds were set at 8 and 12 nodes respectively.

8.1 Latency and Throughput

Latency. Figure 4 plots a group's client operation processing latency for single cluster and multi-cluster settings. RTT network latencies between sites in the multi-cluster experiments were as follows: (EC2-East, EC2-West: 82ms), (EC2-East, Emulab: 79ms), (EC2-West, Emulab: 38ms). The plotted latencies do not include the network delay between the client and the group.

Figure 4(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 4(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 5 compares the impact of latency-optimized policy with $k = 3$ (described in Section 7) to the random join policy on the primary's write latency in a P2P setting. In both P2P deployments, nodes joined Harmony 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 5 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 P2P setting also demonstrate the benefit of majority consensus in mitigating the impact of slow-performing outlier nodes on group operation latency. Though P2P 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.

Throughput. Figure 6 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 an RSM command is linear in group size. Each additional primary, however, adds 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.

The single cluster and multi-cluster settings have a significant disparity in operation latency. However, group throughput in the multi-cluster setting (Figure 6(a)) is within 30% of the group throughput in a single cluster setting (Figure 6(b)). And for large groups this disparity is marginal. The reason for this is pipelining of client requests by group primaries.

8.2 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.

	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 and standard deviations for different deployment settings.

Basic single cluster latency. In the single cluster setting all operations take less than 10ms. The ability to stop an RSM efficiently (described in Section 3.3) helps to keep this latency low across all setings. Splitting and merging are the most expensive operations as they require coordination 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 7. From the table, the latency optimizing policy significantly reduced the latency cost of all operations.

Impact of policy on P2P latency. Figure 7 plots CDFs of latencies for the P2P setting. It compares the random join with random leader policies (Figure 7(a)) against latency-optimized join and latency-optimized leader policies described in Section 7 (Figure 7(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.

8.3 Scalability

To evaluate Harmony's scalability we compared Harmony's throughput to ZooKeeper's in a single cluster setting while varying the number of participating nodes. Because ZooKeeper's performance degrades with more than 5 nodes, we compare ZooKeeper's throughput with 5 nodes against Harmony's throughput with a variable number of nodes (from 5 to over 100). Additionally, Harmony experienced node churn at a rate that is exponentially distributed with a median session time of 1 hour. This rate is aggressive for a single cluster setting, and illustrates Harmony's worst-cast throughput scalability in this setting. Figure 8.3 plots the average throughput results with standard deviations. Harmony's throughput scales linearly with the number of nodes, with some variability due to churn and uneven group sizes. Zookeeper does not provide any mechanisms for scaling beyond a single group deployment.

Load-balanced join policy. In a single cluster setting, the latency-optimized join policy would have little effect. Instead, a load-balanced join policy is more appropriate.

We also evaluate the impact of load-balanced join policy (described in Section 7) on normalized system load. Figure 9 compares this policy (with $k = 3$) to the random join policy. Clients selected random keys to write, and a group's load was the total writes it processed so far. A group's normalized load is its load divided by the average load across all groups. As Figure 9 shows, the effect of the load-balanced policy was to considerably reduce the load skew in the system.

8.4 Consistency and Availability

Consistency overhead. A common deficiency of systems providing strict consistency is their performance penalty. To understand the performance impact of strict consistency in Harmony, we compared the performance of write operations in Harmony and in OpenDHT [30] in a P2P setting. These measurements were performed under no churn. Figure 8.4 plots the CDF of the latencies collected for both systems. OpenDHT's median latency is 301ms and Harmony's median latency is 371ms. Both systems use the *same* routing protocol, therefore the overhead in Harmony is due primarily to the latency of executing an RSM operation. The extra cost of consistency in Harmony is equivalent to having an extra hop in a multi-hop DHT lookup.

Availability To measure the impact of reconfiguration, split, and merge operations on the availability in Harmony we modified the routing algorithm to fail when the destination group for a client operation is in the middle of one of these operations (typically, in this case the client operation blocks). We measured the resulting availability loss in a P2P setting for different churn rates – node lifetimes were exponentially distributed with varying median values. Additionally, we collected availability loss results for OpenDHT, which was deployed in the same setting and used the same churn rate. As discussed in Section 2, unavailability in OpenDHT occurs due to inconsistent successor pointer maintenance. Figure 8.4 plots the availability loss in Harmony and OpenDHT as median node lifetimes increase. Harmony maintain higher availability in all cases. Moreover, unlike in OpenDHT, unavailability due to node churn in Harmony is temporary, as the system always recovers to a consistent state. One of the reasons why Harmony has high availability is because it stops RSMs efficiently (described in Section 3.3).

Inconsistency. To test whether the Harmony prototype maintains consistency at all times in practice, we deployed Harmony in a P2P setting and used millions of pairs of client operations that wrote and later read a random location. Each pair of operations were performed by two different PlanetLab hosts. If the value read was not what was written, it was flagged as an inconsistency. Inconsistency results for Harmony and OpenDHT at different churn rates are plotted in Figure 8.4. Harmony had zero inconsistencies across all churn rates, and in all of our experiments to date. OpenDHT had some inconsistency at all churn rates. This is consistent with prior work, which found that on average 5% of keys in OpenDHT are owned by multiple nodes simultaneously even in settings with low churn [34]

9 Related Work

One set of related work that motivated our design are systems that provide strong consistency and high performance. These include the Chubby lock server [2], and Yahoo's ZooKeeper [13]. Chubby is a centralized service that uses a single group of nodes to provide consistent storage with distributed consensus over RSMs [2]. ZooKeeper has a similar design but presents a different API. Both systems have limited scalability. Improving their scalability while simultaneously preserving consistency remains an open research problem. Harmony provides the same guarantees as Chubby and ZooKeeper, but has superior scalability.

Harmony employs numerous self-organizing techniques that previously appeared in systems that distribute application state and achieve high scalability, but trade-off consistency. DHTs store application state in a scalable manner across a set of machines and use multi-hop lookup tables to locate the desired data. The most widely available DHTs [27, 37, 32, 24] and storage systems that use them [6, 33, 3, 8] provide only weak, best effort guarantees regarding consistency of stored data. Moreover, these systems exhibit consistent routing only under certain assumptions of rates of node failure and churn. When these assumptions are violated, these systems can be brittle – Chord, for example, is well known to fail at sufficient churn rates. Weak consistency undermines the utility of the storage substrate and raise the complexity burden for developers who develop applications on top of the substrate.

The Harmony policies we describe are well known and have been previously shown to be robust in self-organizing settings [25]. We show that these algorithms are equally applicable to strongly consistent systems like Harmony. For example, Kademlia [24], the DHT used to coordinate swarm membership within BitTorrent, is deployed across millions of clients but provides weak consistency; a read of a key's value after a write to the key is not guaranteed to return the new value. Similarly, Amazon's Dynamo [8] – a distributed store used for e-commerce applications within the data center – sacrifices consistency under certain failure scenarios. In such cases, it is the responsibility of the application to manage its state replication. There have also been extensive research efforts to improve DHT performance, and to design DHTs that withstand churn [28, 29]. This work informs several of our design decisions with Harmony. For example, many of the join and leader election policies we describe are motivated by this line of work.

A line of work that shares our goals but has a different approach attempts to achieve strong consistency on top of an inconsistent data store. Etna [26] is a representative system of this approach. Unfortunately such systems inherit consistency problems from the underlying data store, resulting in lower object availability and system efficiency. For example, inconsistencies in the underlying routing protocol will manifest as unavailability at the higher layers.

The location service is a key component of Harmony. Prior work by Rodriguez on membership services [31] builds a service which survives byzantine failures. However, this work is predicated on a centralized (but replicated) membership service that provides only modest scalability. Harmony operates at higher levels of scalability and is self-managing – the tasks of monitoring nodes, detecting group membership changes, and reconfiguring groups is all highly distributed. A more recent system, Census [4], is a scalable membership service that provides functionality that is similar to our location service. However, having groups maintain their own membership is a more efficient and robust solution in our setting. As a result Harmony's design is highly scalable, and has minimal inter-group dependency.

In a position paper, Lynch et al. [21] propose the use of state machine replication for atomic data access in DHTs. The important insight of this work is to implement a node in the traditional DHT with a group of nodes, whose read and write operations are atomic and use group-wide consensus. Harmony adopts this approach in its design of self-organizing mechanisms. We extend the basic idea with new mechanisms and abstractions that are necessary to have a practical system. The group abstraction has also been used similarly in PRing [5], a structure that extends Chord with range querying capabilities. PRing performs key range merges and splits in response to unbalanced workload, while Harmony performs the more difficult task of merging and slitting replica groups.

Prior work on migration of replicated stateful services, such as the recent SMART Replication [20] also employs Paxos for agreement between replicas on client request. Unlike SMART, Harmony does not support migration between arbitrary replica sets. The key innovation in Harmony over SMART is to coordinate independent RSMS and to leverage self-organization techniques to improve system scalability. This paper's focus is on articulating a correct and in many ways a conservative system design. Mencius [22]

demonstrates RSM optimizations that are relevant to both multi-cluster and P2P Harmony deployments

10 Conclusion

This paper presented the design, implementation and evaluation of Harmony – a distributed object storage system that provides clients with linearalizable semantics, high performance, and scalability. Harmony leverages the RSM abstraction to organize nodes into highly reliable groups, each of which independently serve client requests to segments of the namespace. 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 P2P, multi-cluster, and single cluster deployments. Our results demonstrate that Harmony 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 separation of mechanism from policy enables Harmony to effectively adapt to the different deployment settings for significant gain in load balance, latency, and resilience.

References

[1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Transactions on Computer Systems*, 27(3), 2009.

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

[3] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, S. Shenker, and J. Hellerstein. A case study in building layered DHT applications. In *Proc. of SIGCOMM*, 2005.

[4] J. Cowling, D. R. K. Ports, B. Liskov, R. A. Popa, and A. Gaikwad. Census: Location-Aware membership management for Large-Scale distributed systems. In *Proc. of USENIX ATC*, San Diego, CA, USA, 2009.

[5] A. Crainiceanu. P-Ring: An Efficient and Robust P2P Range Index Structure. In *SIGMOD'07*, New York, NY, USA, 2007. ACM.

[6] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of SOSP*, Banff, Alberta, Canada, 2001.

[7] J. Dean. Large-Scale distributed systems at google: Current systems and future directions, 2009.

[8] 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*, Stevenson, Washington, USA, 2007.

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

[10] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. of SOSP*, 1989.

[11] 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 *SOSP*, pages 314–329, 2003.

[12] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM TOPLAS*, 12(3), 1990.

[13] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proc. of USENIX ATC*, volume 8, Boston, MA, USA, 2010.

[14] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7), 1978.

[15] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2), 1998.

[16] L. Lamport, D. Malkhi, and L. Zhou. Stoppable Paxos. TechReport, Microsoft Research, 2008.

[17] L. Lamport, D. Malkhi, and L. Zhou. Reconfiguring a state machine. *ACM SIGACT News*, 41(1), 2010.

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

[19] X. Li, J. Misra, and C. G. Plaxton. Concurrent maintenance of rings. *Distributed Computing*, 19(2), 2006.

[20] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell. The SMART way to migrate replicated stateful services. In *Proc. of EuroSys*, Leuven, Belgium, 2006.

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

[22] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: building efficient replicated state machines for WANs. In *Proc. of OSDI*, San Diego, California, 2008.

[23] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*. North-Holland, 1989.

[24] P. Maymounkov and D. Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 53–65, London, UK, 2002. Springer-Verlag.

[25] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10), 2001.

[26] 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.

[27] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proc. of SIGCOMM*, 2001.

[28] S. Rhea, B. Chun, J. Kubiatowicz, and S. Shenker. Fixing the embarrassing slowness of OpenDHT on PlanetLab. In *WORLDS*, Berkeley, CA, USA, 2005.

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

[30] 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.

[31] R. Rodrigues. Robust Services in Dynamic Systems. Technical Report MIT-LCS-TR, MIT, February 2005.

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

[33] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of SOSP*, 2001.

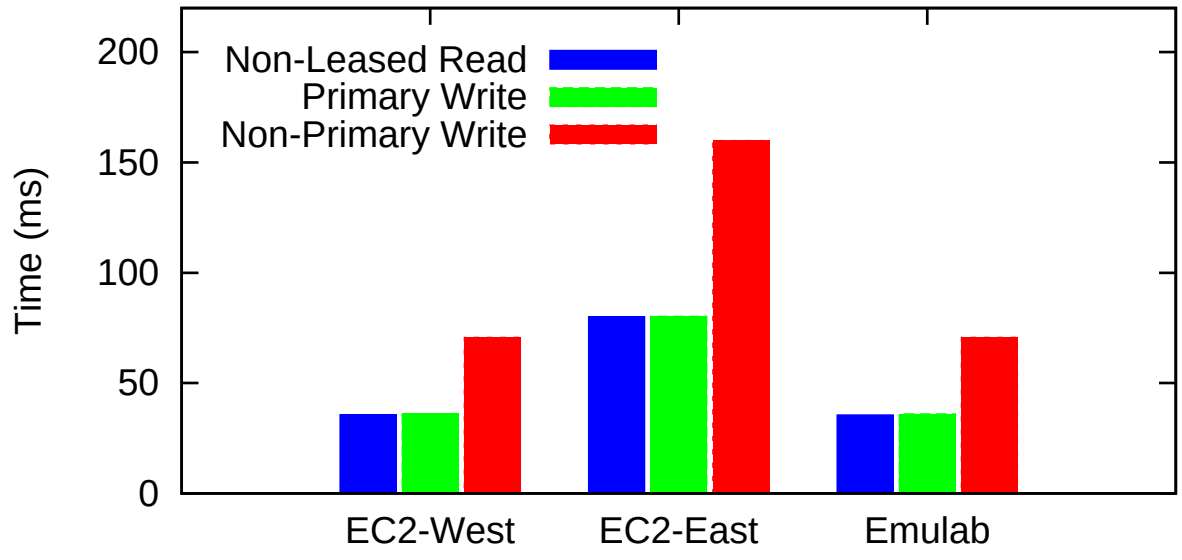
[34] S. Sankararaman, B.-G. Chun, C. Yatin, and S. Shenker. Key Consistency in DHTs. Technical Report UCB/EECS-2005-21, UC Berkeley, 2005.

[35] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4), 1990.

[36] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. Technical Report MIT-LCS-TR-819, MIT, Mar 2001.

[37] 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 TON*, 11(1), 2003.

(a) Single Cluster



(b) Multi-Cluster

Figure 4: 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.

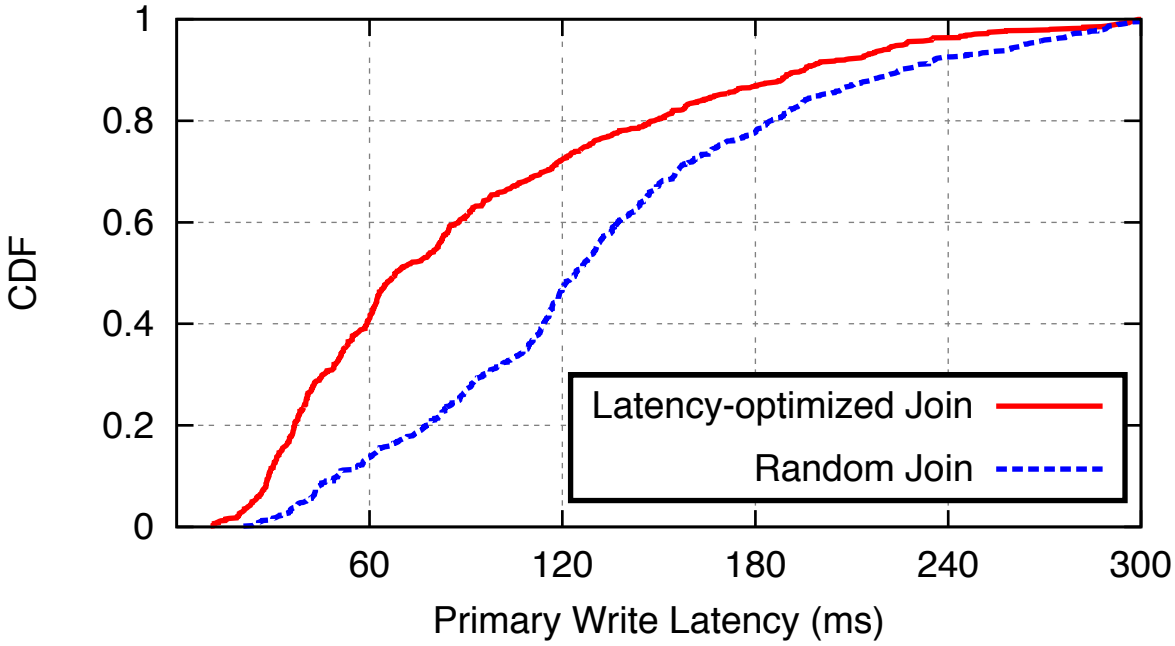
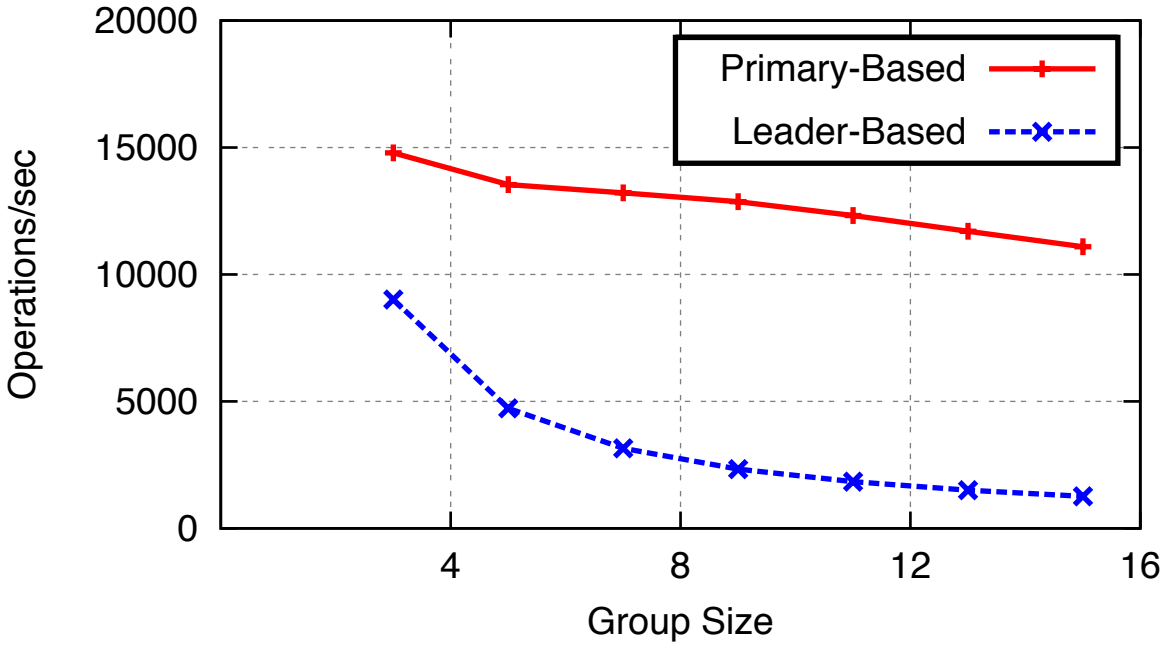
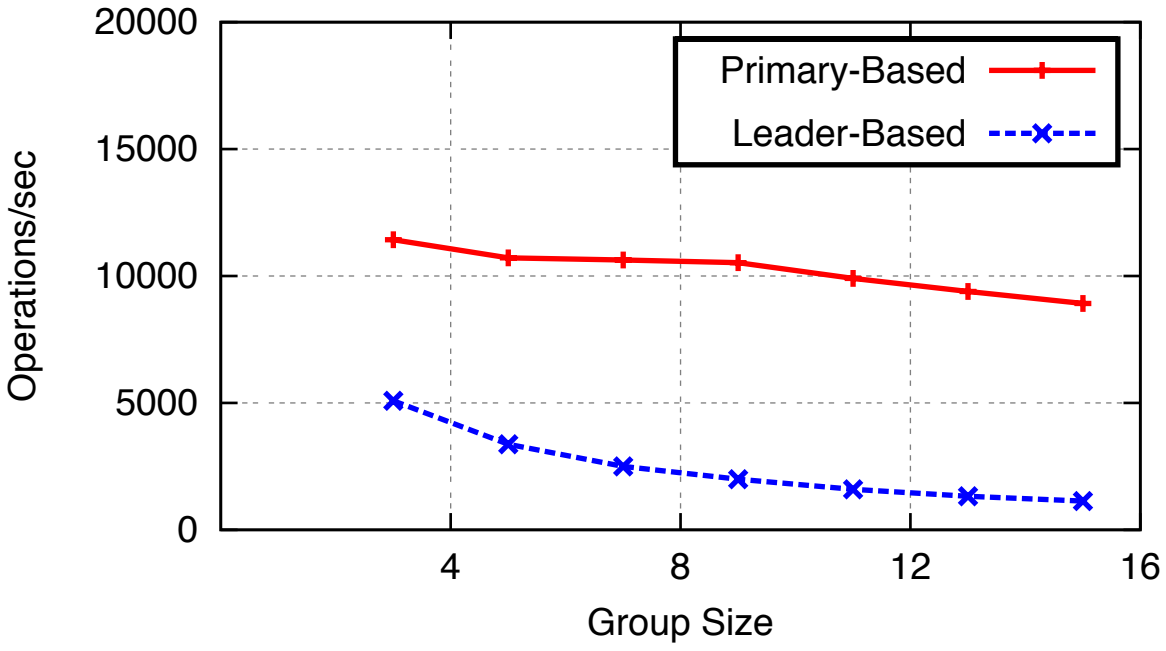


Figure 5: The impact of join policy on write latency in two P2P deployments. The latency-optimized policy is described in Section 7. The random join policy directs nodes to join a group at random.

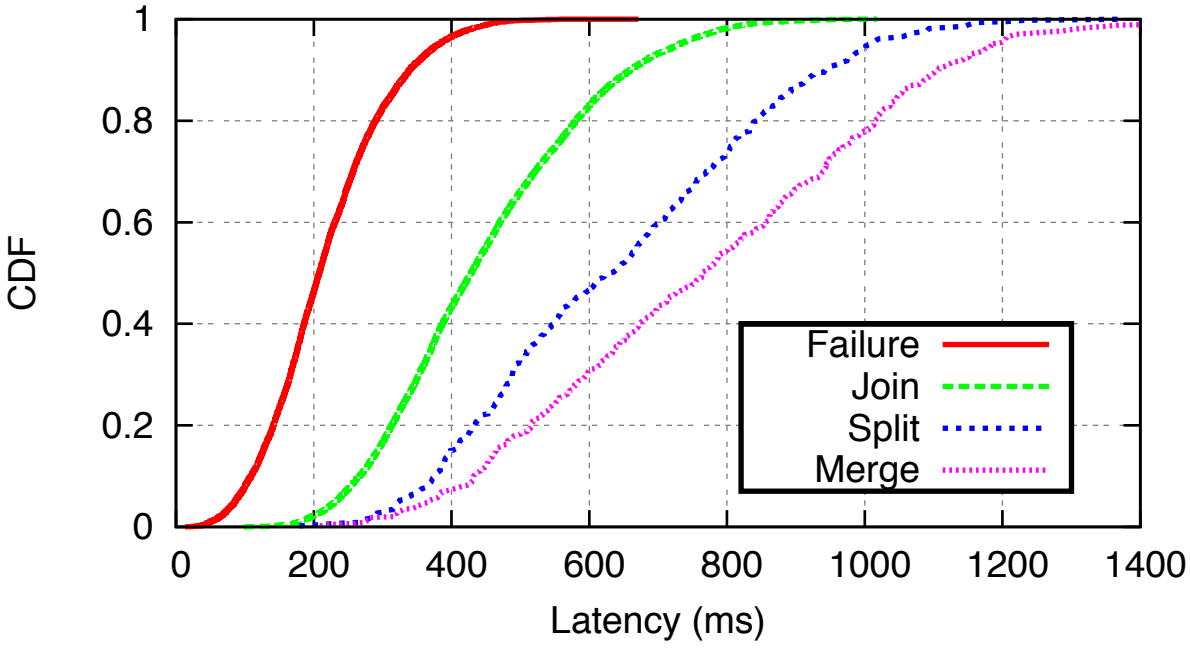


(a) Single Cluster

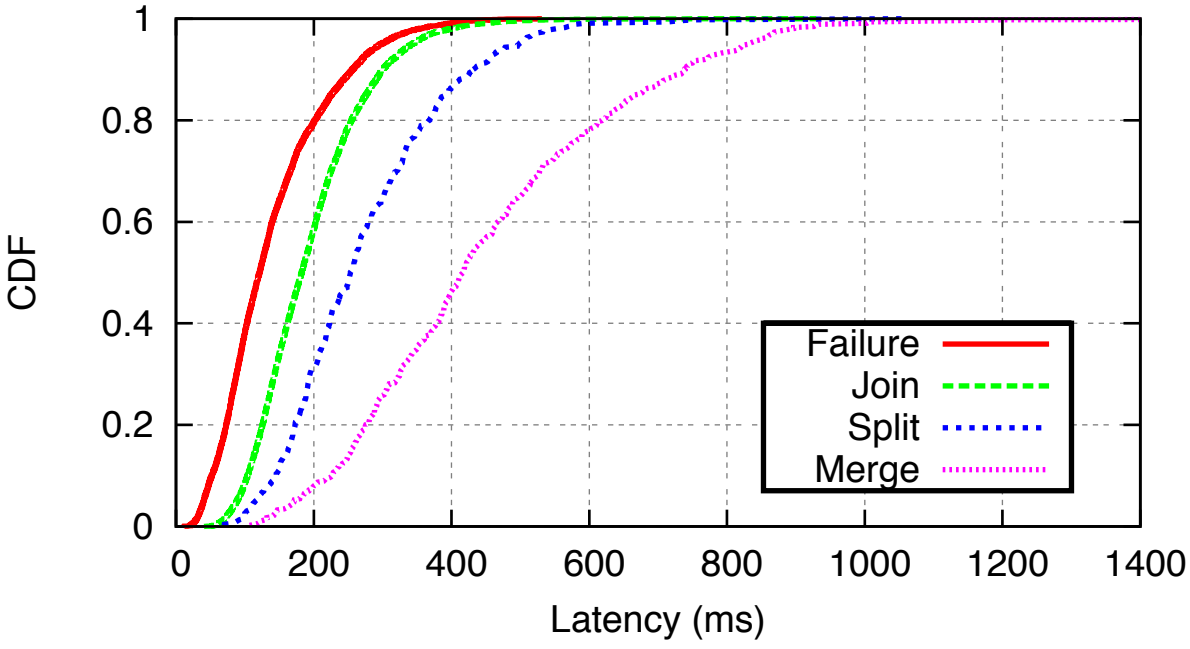


(b) Multi-cluster

Figure 6: Harmony Group throughput in single cluster and multi-cluster settings.



(a) Unoptimized



(b) Join and Leader Optimized for Latency

Figure 7: 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.

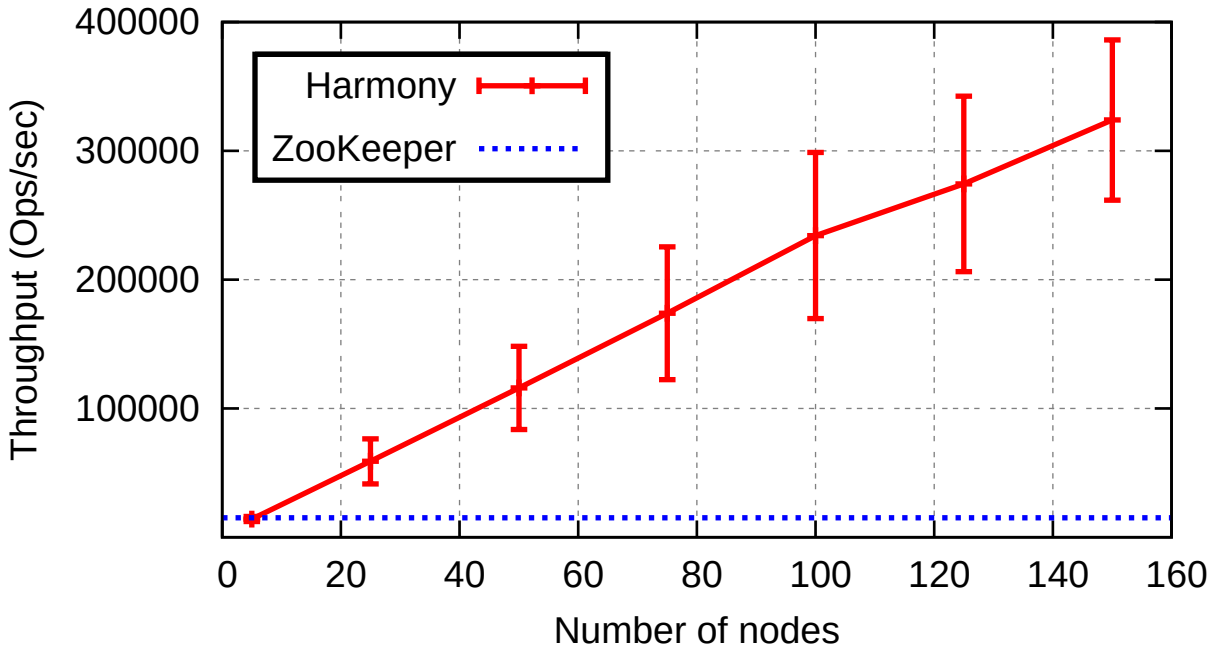


Figure 8: Comparison of Harmony and ZooKeeper throughput scalability in a single cluster setting with varying number of nodes.

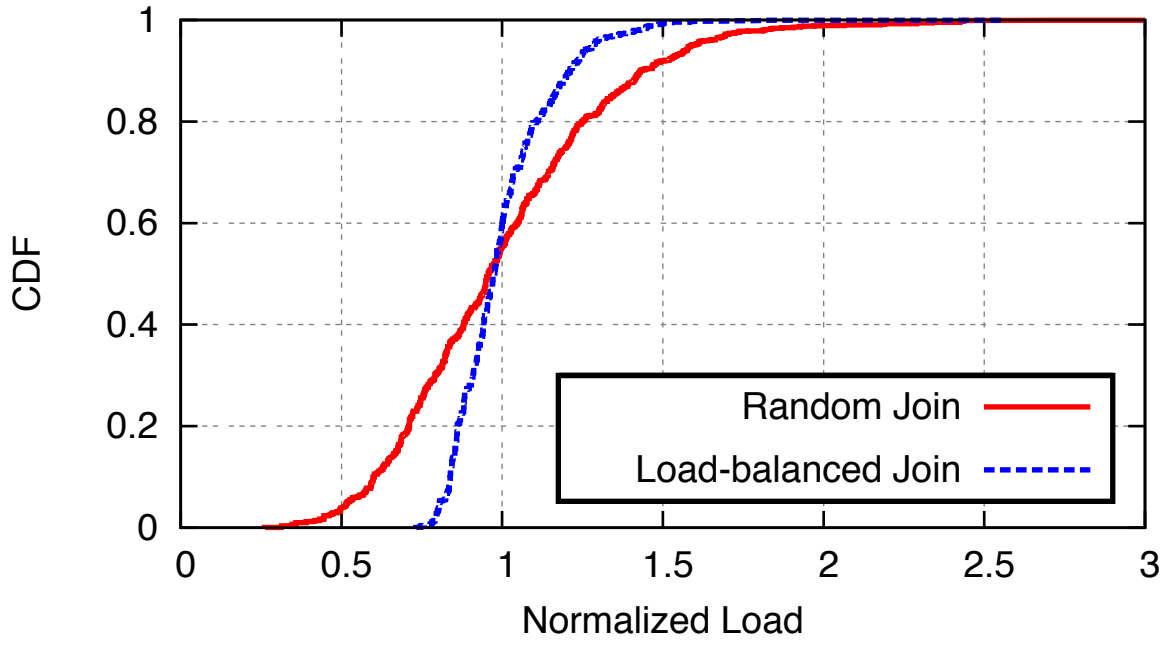


Figure 9: The impact of join policy on load in a single cluster setting.

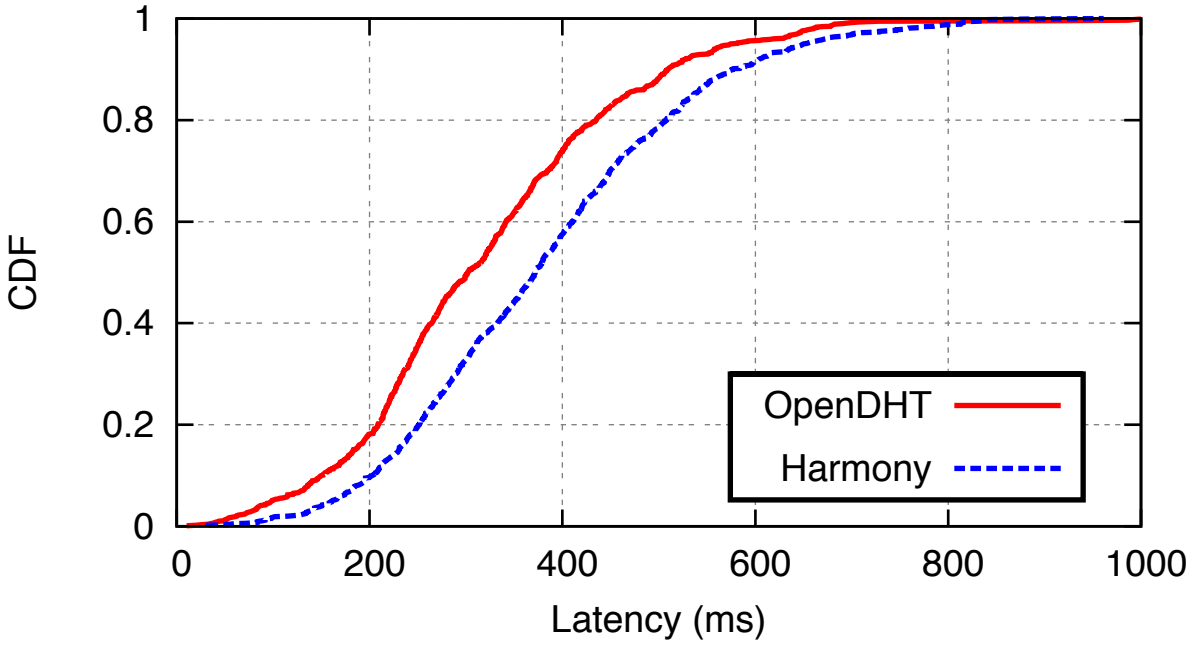


Figure 10: Write latency for OpenDHT and Harmony, in a P2P setting.

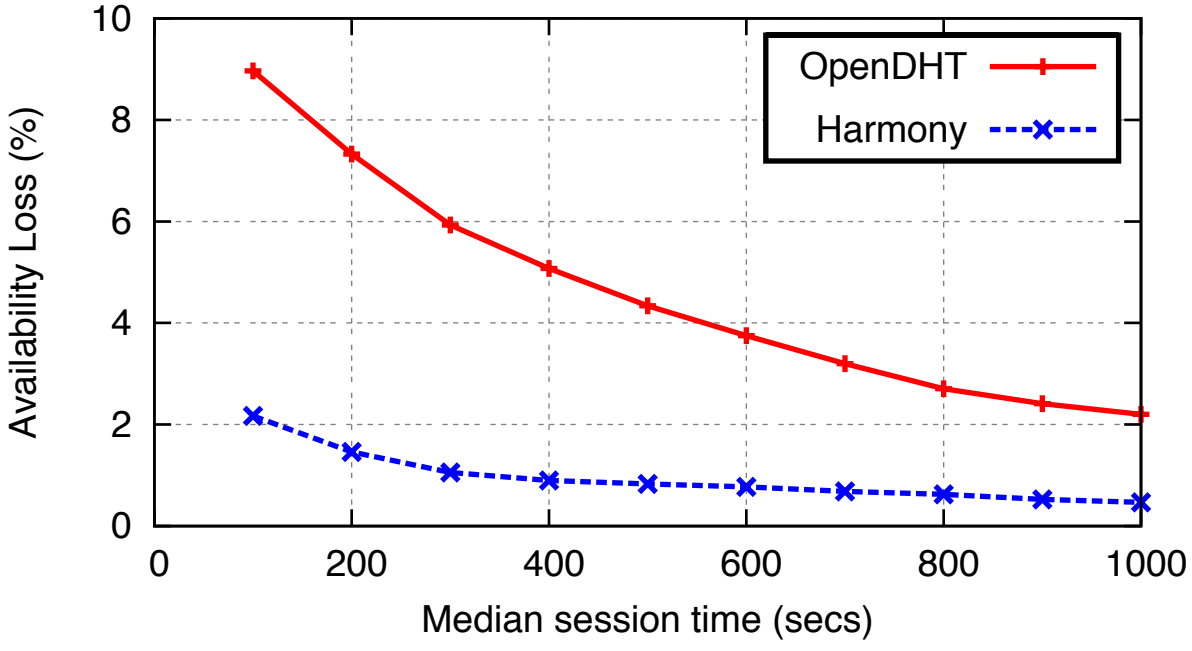


Figure 11: Availability loss in OpenDHT and Harmony at different churn rates in a P2P setting.

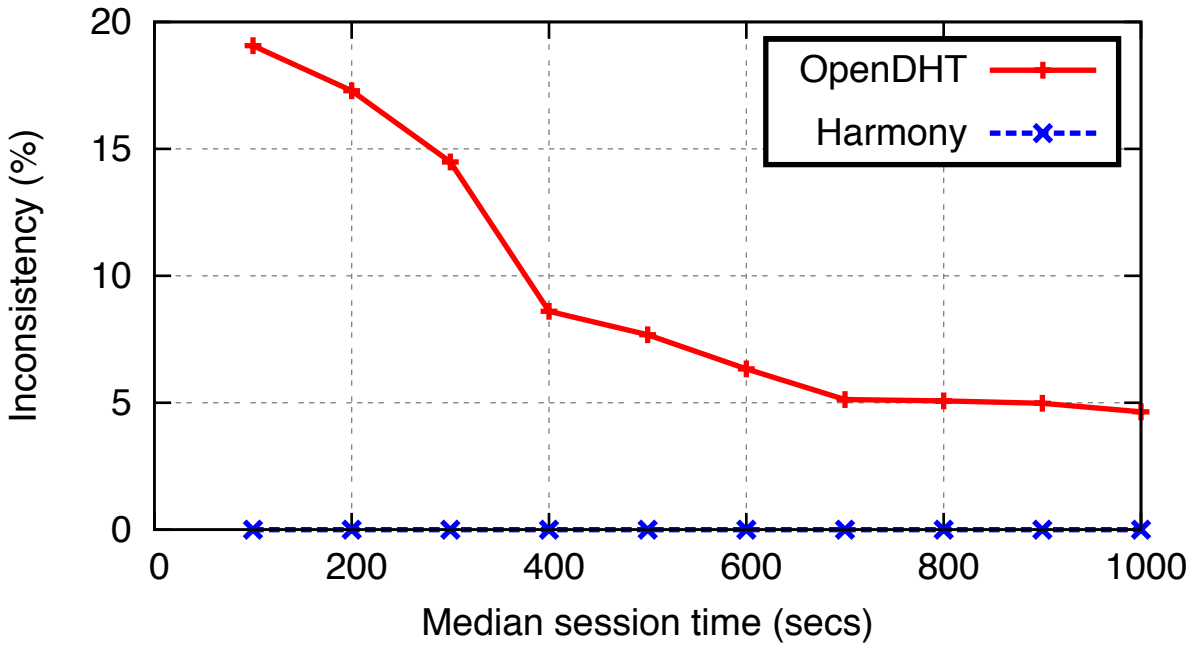


Figure 12: Inconsistency in OpenDHT and Harmony at different churn rates in a P2P setting.