USENIX Association

Proceedings of the First Symposium on Networked Systems Design and Implementation

San Francisco, CA, USA
March 29–31, 2004

© 2004 by The USENIX Association All Rights Reserved For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738 Email: office@usenix.org WWW: http://www.usenix.org
Rights to individual papers remain with the author or the author's employer.
Permission is granted for noncommercial reproduction of the work for educational or research purposes.
This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

Consistent and Automatic Replica Regeneration

Haifeng Yu
Intel Research Pittsburgh / Carnegie Mellon University
yhf@cs.cmu.edu, http://www.cs.cmu.edu/~yhf

Amin Vahdat
University of California, San Diego
vahdat@cs.ucsd.edu, http://www.cse.ucsd.edu/~vahdat

Abstract

Reducing management costs and improving the availability of large-scale distributed systems require automatic replica *regeneration*, i.e., creating new replicas in response to replica failures. A major challenge to regeneration is maintaining consistency when the replica group changes. Doing so is particularly difficult across the wide area where failure detection is complicated by network congestion and node overload.

In this context, this paper presents Om, the first read/write peer-to-peer wide-area storage system that achieves high availability and manageability through online automatic regeneration while still preserving consistency guarantees. We achieve these properties through the following techniques. First, by utilizing the *limited view divergence* property in today's Internet and by adopting the *witness model*, Om is able to regenerate from any single replica rather than requiring a majority quorum, at the cost of a small (10^{-6} in our experiments) probability of violating consistency. As a result, Om can deliver high availability with a small number of replicas, while traditional designs would significantly increase the number of replicas. Next, we distinguish *failure-free* reconfigurations from *failure-induced* ones, enabling common reconfigurations to proceed with a single round of communication. Finally, we use a *lease graph* among the replicas and a two-phase write protocol to optimize for reads, and reads in Om can be processed by any single replica. Experiments on PlanetLab show that consistent regeneration in Om completes in approximately 20 seconds.

1 Introduction

Replication has long been used for masking individual node failures and for load balancing. Traditionally, the set of replicas is fixed, requiring human intervention to repair failed replicas. Such intervention can be on the critical path for delivering target levels of performance and availability. Further, the cost of maintenance now dominates the total cost of hardware ownership, making it increasingly important to reduce such human intervention. It is thus desirable for the system to automatically *regenerate* upon replica failures by creating new replicas on alternate nodes. Doing so not only reduces maintenance cost, but also improves availability because regeneration time is typically much shorter than human repair time.

Motivated by these observations, automatic replica regeneration and reconfiguration (i.e., change of replica group membership) have been extensively studied in cluster-base Internet services [12, 34]. Similarly, automatic regeneration has become a necessity in emerging large-scale distributed systems [1, 10, 20, 25, 30, 33, 35]. One of the major challenges to automatic regeneration is maintaining consistency when the composition of the replica group changes. Doing so is particularly difficult across the wide-area where failure detection is complicated by network congestion and node overload. For example, two replicas may simultaneously suspect the failure of each other, form two new disjoint replica groups, and independently accept conflicting updates.

The focus of this work is to enable automatic regeneration for replicated wide-area services that require some level of consistency guarantees. Previous work on replica regeneration either assumes read-only data and avoids the consistency problem (e.g., CFS [10] and PAST [33]), or simply enforces consistency in a best-effort manner (e.g., Inktomi [12], Porcupine [34], Ivy [25] and Pangaea [35]). Among those replication systems [1, 6, 20, 30, 37] that do provide strong consistency guarantees, Farsite [1] does not implement replica group reconfiguration. Oceanstore [20, 30] mentions automatic reconfiguration as a goal but does not detail its approach, design or implementation. Proactive recovery [6] enables the same replica to leave the replica group and later re-join, but still assumes a fixed set of replicas. Finally, replicated state-machine research [37] typically also assumes a static set of replicas.

In this context, we present Om, a read/write peer-to-peer

wide-area storage system. Om logically builds upon PAST [33] and CFS [10], but achieves high availability and manageability through online automatic regeneration while still preserving consistency guarantees. To the best of our knowledge, Om is the first implementation and evaluation of a wide-area peer-to-peer replication system that achieves such functionality.

Om's design targets large, infrastructure-based hosting services consisting of hundreds to thousands of sites across the Internet. We envision companies utilizing hosting infrastructure such as Akamai [2] to provide wide-area mutable data access service to users. The data may be replicated at multiple wide-area sites to improve service availability and performance. We believe that our design is also generally applicable to a broader range of applications, including: i) a totally-ordered event notification system, ii) distributed games, iii) parallel grid computing applications sharing data files, and iv) content distribution networks and utility computing environments where a federation of sites deliver read/write network services.

We adopt the following novel techniques to achieve our goal of consistent and automatic replica regeneration.

1. Traditional designs for regeneration require a *majority* of replicas to coordinate consistent regeneration. We show that by taking advantage of the *limited view divergence* property in today's Internet and by adopting the *witness model* [40], Om is able to regenerate from any single replica at the cost of a small probability of violating consistency. As a result, Om can deliver high availability with a small number of replicas, while traditional designs would significantly increase [42] the number of replicas in order to deliver the same availability. When strict consistency is desired, Om can also trivially replace the witness model with a simple majority quorum (at the cost of reduced availability) to provide strict consistency.

2. We distinguish between *failure-free* and *failure-induced* reconfiguration, enabling common reconfigurations to proceed with a single round of communication while maintaining correctness even if a failure should occur in the middle.

3. We use a *lease graph* among all replicas and a two-phase write protocol to avoid executing a consensus protocol for normal writes. Reads in Om proceed with a single round trip to any single replica, yielding the read performance of a centralized service but with better network locality.

Om assumes a crash (stopping) rather than Byzantine failure model. While this assumption makes our approach inappropriate for a certain class of services, we argue that the performance, availability, consistency, and flexible reconfiguration resulting from our approach will make our work appealing for a range of important applications.

Through WAN measurement and local area emulation, we observe that the probability of violating consistency in Om is approximately 10^{-6}, which means that on average, inconsistency occurs once every 250 years with 5 replicas and a pessimistic 12 hours replica MTTF. At the same time, the ability to regenerate from any replica enables Om to achieve high availability using a relatively small number of replicas [42] (e.g., 99.9999% using 4 replicas with node MTTF of 12 hours, regeneration time of 5 minutes and human repair time of 8 hours). Under stress tests for write throughput on PlanetLab [27], we observe that regeneration in response to replica failures only causes a 20-second service interruption.

We provide an overview of Om in the next section. The following three sections then discuss the details of normal case operations, reconfiguration, and single replica regeneration in Om. We present unsafety (probability of violating consistency) and performance evaluation in Section 6. Finally, Section 7 discusses related work and Section 8 draws our conclusions.

2 System Architecture Overview

2.1 Naming and Configurations

Om relies on Distributed Hash Tables (DHTs) [32, 38] for naming its objects. The current implementation of Om uses FreePastry [13]. Om invokes only two common peer-to-peer APIs [11] from FreePastry: **void route(key → K, msg → M, nodehandle → hint)** and **nodehandle[] replicaSet(key → K, int → max_rank)**. We use these APIs to determine the set of nodes that should hold a particular object. Om does not require any change to the FreePastry code.

DHTs do not guarantee the correctness of naming. For example, the same key may be mapped to different nodes if routing tables are stale. In Om, each node ultimately determines whether it is a replica of a certain Om object. With inconsistent routing in DHTs, user requests may be routed to the wrong node. Instead of returning an incorrect value, the node will tell the user that it does not have the data.

```
public class Configuration {
    boolean valid;
    int sequenceNum;
    LogicalAddr primary;
    LogicalAddr[] secondary;
    String consensusID;
}
```

Figure 1: A configuration.

Om servers are grouped into configurations (Figure 1). Each configuration contains the set of servers holding copies of a particular object. A physical node may belong to multiple configurations. Conceptually, the total number of configurations equals the number of objects in Om. However, multiple objects residing on the same set of replicas share the same configuration, which significantly reduces the number of configurations and overall regeneration activity.

2.2 Two Quorum Systems for Maintaining Consistency

Throughout this paper, we use *linearizability* [18] as the definition for consistency. An *access* to an Om object is either a *read* or a *write*. Each access has a *start time*, the wall-clock time when the user submits the access, and a *finish time*, the wall-clock time when the user receives the reply. Linearizability requires that: i) each access has a unique *serialization point* that falls between its start time and finish time, and ii) the results of all accesses and the final state of the replicas are the same as if the accesses are applied sequentially by their serialization points.

To maintain consistency, Om uses two different quorum systems in two different places of the design. The first is a read-one/write-all quorum system for accessing objects on the replicas. We choose to use this quorum system to maximize the performance of read operations. In general, however, our design supports an arbitrary choice of read/write quorum. Each configuration has a *primary* replica responsible for serializing all writes and transmitting them to *secondary* replicas. The failure of any replica causes regeneration. Thus both primary and secondary replicas correspond to *gold replicas* in Pangaea [35]. It is straightforward to add additional *bronze replicas* (which are not regenerated) into our design. Distinguishing these two kinds of replicas helps to decrease the overhead of maintaining the lease graph, liveness monitoring and performing two-phase writes among the gold replicas.

Reads can be processed by any replica without interacting with other replicas. A write is always forwarded to the primary, which uses a two-phase protocol to propagate the write to all replicas (including itself). Even though two-phase protocols in WAN can incur high overhead, we limit this overhead because Om usually needs a relatively small number of replicas to achieve certain availability target [42] (given its single replica regeneration mechanism).

The second quorum system is used during reconfiguration to ensure that replicas agree on the membership of the new configuration. In wide-area settings, it is possible for two replicas to simultaneously suspect the failure of each other and to initiate regeneration. To maintain consistency, the system must ensure a unique configuration for the object at any time. Traditional approaches for guaranteeing unique configuration require each replica to coordinate with a majority before regeneration, so that no simultaneous conflicting regeneration can be initiated.

Given the availability cost of requiring a majority [42] to coordinate regeneration, we adopt the witness model [40] that achieves similar functionality as a quorum system. In the witness model, quorum intersection is not always guaranteed, but is extremely likely. In return, a quorum in the witness model can be as small as a single node. While our implementation uses the witness model, our design can trivially replace the witness model with a traditional quorum system such as majority voting.

2.3 Node Failure/Leave and Reconfiguration

The membership of a configuration changes upon the detection of node failures or explicit reconfiguration requests. Failures are detected in Om via timeouts on messages or heartbeats. By definition, accurate failure detection in an environment with potential network failure and node overload, such as the Internet, is impossible. Improving failure detection accuracy is beyond the scope of this paper.

There are two types of reconfigurations in Om: *failure-free reconfiguration* and *failure-induced reconfiguration*. Failure-free reconfiguration takes place when a set of nodes gracefully leave or join the configuration. "Gracefully" means that there are no node failures or message timeouts during the process. On the other hand, Om performs failure-induced reconfiguration when it (potentially incorrectly) detects a node failure (in either normal operation or reconfiguration).

Failure-free reconfiguration is lightweight and requires

only a single round of messages from the primary to all replicas, a process even less expensive than writes. Failure-induced reconfiguration is more expensive because it uses a *consensus protocol* to enable the replicas to agree on the membership of the next configuration. The consensus protocol, in turn, relies on the second quorum system to ensure that the new configuration is unique among the replicas.

Under a denial of service (DoS) attack, all reconfigurations will become failure-induced. One concern is that an Om configuration must be sufficiently over-provisioned to handle the higher cost of failure-induced reconfiguration under the threat of such attacks. However, the reconfiguration functionality of Om actually enables it to dynamically shift to a set of more powerful replicas (or expand the replica group) under DoS attacks, making static over-provisioning unnecessary.

2.4 Node Join and Reconfiguration

New replicas are always created by the primary in the background. To achieve this without blocking normal operations, the primary replica creates a snapshot of the data and transfers the snapshot to the new replicas. During this process, new reads and writes are still accepted, with the primary logging those writes accepted after creating the snapshot. After the snapshot has been transferred, the primary will send the logged writes to the new replicas, and then initiate a failure-free reconfiguration to include them in the configuration. Since the time needed to transfer the snapshot tends to dominate the total regeneration time, Om enables online regeneration without blocking accesses.

Each node in the system maintains an incarnation counter in stable storage. Whenever a node loses its state in memory (due to a crash or reboot), it increments the incarnation number. After the node rejoins the system, it should discard all messages intended for older incarnations. This is necessary for a number of reasons: For example, otherwise a primary that crashes and then recovers immediately will not be able to keep track of the writes in the middle of the two-phase protocol.

3 Normal Case Operations

Given the overall architecture described above, we now discuss some of the complex system interactions in Om. Despite the simplicity of the read-one/write-all approach for accessing objects, failures and reconfigurations may introduce several anomalies in a naive design. Below we describe two major anomalies and our solutions.

The first anomaly arises when replicas from old configurations are slow in detecting failures, and continue servicing stale data after reconfiguration (initiated by other replicas). We address this scenario by leveraging leases [17]. In traditional client-server architectures, each client holds a lease from the server. However, since Om can regenerate from any replica, a replica needs to hold valid leases from all other replicas.

Requiring each replica to contact every other replica for a lease can incur significant communication overhead. Fortunately, it is possible for a replica to *sublease* those leases it already holds. As a result, when a replica A requests a lease from B, B will not only grant A a lease for B, it can also potentially grant A leases for other replicas (with a shorter lease expiration time, depending on how long B has been holding those leases).

Following we abstract the problem by considering replicas to be nodes in a *lease graph*. If a node A directly requests a lease from node B, we add an arc from B to A in the graph. A lease graph must be strongly connected to avoid stale reads. Furthermore, we would like the layers of recursive subleasing to be as small as possible because each layer of sublease decreases the effective duration of the lease. Define the diameter of a lease graph to be the smallest integer d, such that any node A can reach any other node B via a directed path of length at most d. In our system, we would like to limit d to 2 to ensure the effectiveness of subleasing. Overhead of lease renewal is determined by the number of arcs in the lease graph. It has been proven [16] that with $n \geq 4$ nodes, the minimal number of arcs to achieve $d = 2$ is $2(n-1)$. For $n \geq 5$, we can show that the *only* graph reaching this lower bound is a star-shaped graph. Thus, our lease graphs are all star-shaped, with every node having two arcs to and from a central node. The central node does not have to be the primary of the configuration, though it is in our implementation.

A second problem results from a read seeing a write that has not been applied to all replicas, and the write may be lost in reconfiguration. In other words, the read observes temporary, inconsistent state. To avoid this scenario, we employ a two-phase protocol for writes. In the first *prepare* round, the primary propagates the writes to the replicas. Each replica records the write in a *pending queue* and sends back an acknowledgment. After receiving all acknowledgments, the primary will start the second *commit* round by sending commits to all replicas. Upon receiving a commit, a replica applies the corresponding write to the data object. Finally, the primary sends back an acknowledgment to the user. A write becomes "stable" (applied to all replicas) when the user receives an acknowledgment. The lack of an acknowl-

edgment indicates that the write will ultimately be seen by all or none of the replicas. A user may choose to re-submit an un-acknowledged write, and Om performs appropriate duplicate detection and elimination.

After a failure-induced reconfiguration and before a new primary can serialize any new writes, it first collects all pending writes from the replicas in the new configuration and processes the writes again using the normal two-phase protocol. Each replica performs appropriate duplicate detection and elimination in this process. Such design solves the previous problem because if any read sees a write, then the write must be either applied or in the pending queue on all replicas.

4 Reconfiguration

Each configuration has a monotonically increasing sequence number, increased with every reconfiguration. For any configuration and at any point of time, a replica can only be in a single reconfiguration process (either failure-free or failure-induced). It is however, possible that different replicas in the same configuration are simultaneously in different reconfiguration processes.

Conceptually, a replica that finishes reconfiguration will try to inform other replicas of the new configuration by sending *configuration notices*. In failure-free reconfigurations, only the primary does this, because the other replicas are passive. In failure-induced reconfigurations, all replicas transmit configuration notices to aid in completing reconfiguration earlier. In many cases, most replicas do not even need to enter the consensus protocol—they simply wait for the configuration notice (within a timeout).

4.1 Failure-free Reconfiguration

Only the primary may initiate failure-free reconfiguration. Secondary replicas are involved only when i) the primary transmits to them data for creating new replicas; and ii) the primary transmits configuration notices.

The basic mechanism of failure-free reconfiguration is straightforward. After transferring data to the new replicas in two stages (snapshot followed by logged writes as discussed earlier), the primary constructs a configuration for the new desired membership. This new configuration will have a new `sequenceNum` by incrementing the old `sequenceNum`. The `consensusID` of the configuration remains unchanged.

The primary then informs the other replicas of the new configuration and waits for acknowledgments. If timeout occurs, a failure-induced reconfiguration will follow.

```
// A snapshot of the current configuration must be passed in.
void shrink(Configuration dupconf)
    throws InterruptedException {
    //Stop granting leases for current configuration.
    current_configuration.valid = false;

    newmember = set of replicas I can reach in dupconf;
    newconf = new Configuration(newmember);
    newconf.sequenceNum = dupconf.sequenceNum + 2;
    newconf.consensusID = dupconf.name + "_" +
        newconf.sequenceNum;

    decision = consensus(newconf, dupconf.consensusID);
    leaseManager.waitForLeaseExpire(dupconf);

    Block writes and configuration notices;
    if (current_configuration.sequenceNum <
        decision.sequenceNum) {
        current_configuration = decision;
        send configuration notices;
        if (I am primary in decision) applyPendingWrites();
    }
    // If not, then configuration notice received.
    // dupconf is no longer current and reconfig is obsolete.
    Unblock writes and configuration notices;
}
```

Figure 2: Failure-induced reconfiguration.

4.2 Failure-induced Reconfiguration

In contrast to failure-free reconfigurations, failure-induced reconfigurations can only shrink the replica group (potentially followed by failure-free reconfigurations to expand the replica group as necessary). Doing this simplifies design because failure-induced reconfigurations do not need to create new replicas and request them to participate in the consensus protocol. Failure-induced reconfigurations can take place during normal operations, failure-free reconfigurations or even failure-induced reconfigurations.

A replica initiates failure-induced reconfiguration (Figure 2) upon detecting a failure. The replica first disables the current configuration so that leases can no longer be granted for the current configuration. This reduces the time we need to wait for lease expiration later. Next, it will perform another round of failure detection for all members of the configuration. The result (a subset of the current replicas) will be used as a *proposal* for the new configuration. The replica then invokes a consensus protocol, which returns a *decision* that is agreed upon by all

replicas entering the protocol. When invoking the consensus protocol, the replica needs to pass a unique ID for this particular invocation of the consensus protocol. Otherwise, since nodes can be arbitrarily slow, different invocations of the consensus protocol may interfere with one another.

Before adopting a decision, each replica needs to wait for all leases to expire with respect to the old configuration. Finally, the primary of the new configuration will collect and re-apply any pending writes. When re-applying pending writes, the primary only waits for a certain timeout. If a subsequent failure were to take place, the replicas will start another failure-induced reconfiguration.

One important optimization to the previous protocol is that after a replica determines `newmember`, it checks whether it has the smallest ID in the set. If it does not, the replica will wait (within a timeout) for a configuration notice. With this optimization, in most cases, only a single replica enters the consensus protocol, which can significantly improve the time complexity of the randomized consensus protocol (see Section 5.3).

When a failure-induced reconfiguration is invoked in the middle of a failure-free reconfiguration, they may interfere with each other and result in inconsistency. Such issue is properly addressed in our complete design [42].

5 Single Replica Regeneration

Failure-induced reconfigurations depend on a consensus protocol to ensure the uniqueness of the new configuration and in turn, data consistency. Consensus [22] is a classic distributed computing problem and we can conceptually use any consensus protocol in Om. However, most consensus protocols such as Paxos [21] rely on majority quorums and thus cannot tolerate more than $n/2$ failures among n replicas. To reduce the number of replicas required to carry out regeneration (as a desirable side-effect, this also reduces the overhead of acquiring leases and of performing writes), we adopt the *witness model* [40] to achieve probabilistic consensus without requiring a majority.

5.1 Probabilistic Quorum Intersection without Majority

The witness model [40] is a novel quorum design that allows quorums to be as small as a single node, while ensuring probabilistic quorum intersection. In our system, for each new configuration, the primary chooses $m \times t$ witnesses and communicates their identities to all secondary replicas. Witnesses are periodically probed by the primary and refreshed as necessary upon failure. This refresh is trivial and can be done in the form of a two-phase write. If failure occurs between the first and the second phase, a replica will use both old and new witnesses in the consensus protocol. The primary may utilize a variety of techniques to choose witnesses, with the goal of choosing witnesses with small failure correlation and diversity in the set of network paths from the replicas to individual witnesses. For example, the primary may simply use entries from its finger table under Chord [38].

For now, we will consider replicas that are not in *singleton partitions*, where a single node, LAN, or perhaps a small autonomous system is unable to communicate with the rest of the network. Later we will discuss how to determine singleton partitions. We say that a replica can *reach* a witness if a reply can be obtained from the witness within a certain timeout. The witness model utilizes the following *limited view divergence* property:

Consider a set S of functioning randomly-placed witnesses that are not co-located with the replicas (e.g., not in the same LAN). Suppose one replica A can reach the subset S_1 of witnesses and cannot reach the subset S_2 of witnesses (where $S_1 \cup S_2 = S$). Then the probability that another replica B cannot reach any witness in S_1 and can reach all witnesses in S_2 decreases with increasing size of S.

Intuitively, the property says that two replicas are unlikely to have a completely different view regarding the reachability of a set of randomly-placed witnesses. The size of S and the resulting probability are thoroughly studied in [40] using the RON [4] and TACT [41] traces. Later we will also present additional results based on PlanetLab measurements.

The validity of limited view divergence can probably be explained by the rarity [9] of large-scale "hard partitions", where a significant fraction of Internet nodes are unable to communicate with the rest of the network. Given that witnesses are randomly placed, if the two replicas have completely different views on the witnesses, this tends to indicate a "hard partition". Further, the more witnesses, the larger-scale the partition would have to be to result in entirely disjoint views from the perspective of two independent replicas.

To utilize the limited view divergence property, all replicas logically organize the witnesses into an $m \times t$ matrix. The number of rows, m, determines the probability of intersection. The number of columns, t, protects

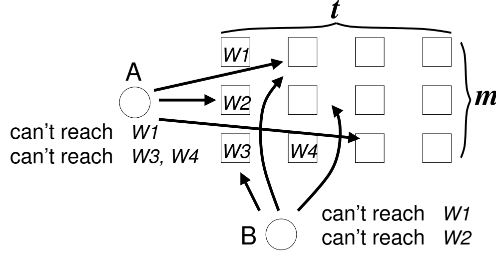

Figure 3: Two replicas and 3×4 witnesses.

against the failure of individual witnesses, so that each row has at least one functioning witness with high probability. Each replica tries to coordinate with one witness from each row. Specifically, a replica uses the first witness from left to right that it can reach for each row (Figure 3). The set of witnesses used by a replica is its quorum. Now consider two replicas A and B. The desirable outcome is that A's quorum intersects with B's. It can be shown that if the two quorums do not intersect, with high probability (in terms of t), A and B have completely different views on the reachability of m witnesses [40].

Replicas behind singleton partitions will violate limited view divergence. However, if the witnesses are not colocated with the replica, then the replica behind the partition will likely not be able to reach *any* witness. As a result, it cannot acquire a quorum and will thus block. This is a desirable outcome as the replicas on the other side of the partition will reach consensus on a new configuration that excludes the node behind the singleton partition. To better detect singleton partitions, a replica may also check whether all reachable witnesses are within its own LAN or autonomous system.

5.2 Emulating Probabilistic Shared-Memory

We intend to substitute the majority quorum in traditional consensus protocols with the witness model, so that the consensus protocol can achieve probabilistic consensus without requiring majority. To do this however, we need a consensus protocol with "good" termination properties for the following reason. Non-intersection in the witness model is ultimately translated into the *unsafety* (probability of having multiple decisions) of a consensus protocol. Unsafety in turn, means violation of consistency in Om. For protocols with multiple rounds, unsafety potentially increases with every round. This precludes the application of protocols such as Paxos [21] that do not have good termination guarantees.

To address the previous issue, we first use the witness model to emulate a *probabilistic shared-memory*, where reads may return stale values with a small probability. We then apply a shared-memory randomized consensus protocol [36], where the expected number of rounds before termination is constant and thus helps to bound unsafety.

To reduce the message complexity of the shared-memory emulation, we choose not to directly emulate [40] the standard notion of reads and writes. Rather, we define an *access* operation on the shared-memory to be an update to an array element followed by a read of the entire array. The element to be updated is indexed by the replica's identifier. The witnesses maintain the array. Upon receiving an access request, a witness updates the corresponding array element and returns the entire array. Such processing is performed in isolation from other access requests on the same witness. Figure 4 provides the pseudo-code for such emulation.

For Replicas:

```
static int version = 0;
int[] access(String arrayname, int newvalue) {
    Record[][] replies = new Record[m][];
    replies[1..m] = null; version++; j = 1;
    while ((∃ i, replies[i] == null) and ( j ≤ t)) {
        send (myindex, version, newvalue) to all witness[i][j]
            where (replies[i] == null);
        wait until all replies received or time out;
        replies[i] = the reply from witness[i][j];
        j++;
    }

    if (replies[1..m] == null) block;
    int[] result = new int[n]; // combine all replies
    for (int k = 1; k ≤ n; k++)
        result[k] = replies[i][k].value, where replies[i][k]
            has the largest version in replies[1..m][k]
    return result;
}
```

For Witnesses:

```
Record[] processAccess(String arrayname, int index,
                       int version, int newvalue) {
    let record[1..n] be the array corresponding to arrayname;
    if (record[index].version < version) {
        record[index].version = version;
        record[index].value = newvalue;
    }
    return record;
}
```

Figure 4: Emulating shared-memory under the witness model.

```
// Shared data: The ith iteration uses two arrays,
// proposed[i] and check[i]. Each array has n entries,
// one for each replica. All entries initialized to null.
int randCons(int proposal) {
    i = 0; myvalue = proposal;
    while (true) {
        i++;
        prop_view = access(proposed[i], myvalue);
        if (different proposals appear in prop_view)
            check_view = access(check[i], 'disagree');
        else
            check_view = access(check[i], 'agree');

        if (check_view only contains 'agree')
            return myvalue; //this is the decision
        if (check_view only contains 'disagree')
            myvalue = a random element in prop_view
                indexed by coinFlip();
        if (check_view has both 'agree' and 'disagree')
            myvalue = prop_view[q],
                ∀ q, where check_view[q] == 'agree';
    }
}
```

Figure 5: Randomized consensus protocol for shared-memory.

While the access primitive appears to be a simple wrapper around reads and writes, it actually violates atomicity and qualitatively changes the semantics of the shared-memory. It reduces the message (and time) complexity of the shared-memory emulation in [40] by half. More details are available in [42].

5.3 Application of Shared-memory Randomized Consensus Protocol

With the shared-memory abstraction, we can now apply a previous shared-memory consensus protocol [36] (Figure 5). For simplicity, we assume that the proposals and decisions are all integer values, though they are actually configurations. In the figure, we already substitute the read and write operations in the original protocol with our new access operations. We implement **coinFlip**() using a local random number generator initialized using a common seed shared by all replicas. Such implementation is different and simpler than the design for standard shared-memory consensus protocols, and it reduces the complexity of the protocol by a factor of $\theta(n^2)$. See [42] for details on why such optimization is possible.

The intuition behind the shared-memory consensus protocol is subtle and several textbooks have chapters devoted to these protocols (e.g., Chapter 11.4 of [8]). Since the protocol itself is not a contribution of this paper, we only enumerate several important properties of the protocol. Proofs are available in [42].

- The protocol proceeds in successive iterations, each iteration has two accesses. Each access requires one round of communication (between the replicas and the witnesses), and needs to coordinate with a quorum. Non-intersection for any access may result in unsafety.
- Each iteration has a certain probability of terminating. The number of iterations before termination is a random variable.
- With two distinct proposals, the expected time complexity of the protocol is below 3.1 iterations (6.2 rounds).
- If all replicas entering the protocol have the same proposal (or if only one replica enters the protocol), the protocol terminates (deterministically) after one iteration. With the optimization in Section 4.2, this will be the situation when the new primary does not crash in the middle of reconfiguration.

6 Experimental Evaluation

This section evaluates the performance and unsafety of Om. Availability of Om and the benefit of single replica regeneration is studied separately [42]. Om is written in Java 1.4, using TCP and nonblocking I/O for communication. All messages are first serialized using Java serialization and then sent via TCP. The core of Om uses an event-driven architecture.

6.1 Unsafety Evaluation

Om is able to regenerate from any single replica at the cost of a small probability of consistency violation. We first quantify such unsafety under typical Internet conditions.

Unsafety is about rare events, and explicitly measuring unsafety experimentally faces many of the same challenges as evaluating service availability [41]. For instance, assuming that each experiment takes 10 seconds to complete, we would need on average over four years to observe a single inconsistency event for an unsafety of 10^{-7}. Given these challenges, we follow the methodology in [41] and use a real-time emulation environment for our evaluation. We instrument Om to add an artificial delay to each message. Since the emulation is performed on a LAN, the actual propagation delay is negligible. We

determine the distribution of appropriate artificial delays by performing a large-scale measurement study of PlanetLab sites. For our emulation, we set the delay of each message sent across the LAN to the delay of the corresponding message in our WAN measurements.

Our WAN sampling software runs with the same communication pattern as the consensus protocol except that it does not interpret the messages. Rather, the replicas repeatedly communicate with all witnesses in parallel via TCP. The request size is 1KB while the reply is 2KB. We log the time (with a cap of 6 minutes) needed to receive a reply from individual witnesses. The sampling interval (time between successive samples) for each replica ranges from 1 to 10 seconds in different measurements. Notice that we do not necessarily wait for the previous probe's reply before sending the next probe. All of our measurements use 7 witnesses and 15 replicas on 22 different PlanetLab sites. To avoid the effects of Internet2 and to focus on the pessimistic behavior of less well-connected sites, we locate the witnesses at non-educational or foreign sites: Intel Research Berkeley, Technische Universitat Berlin, NEC Laboratories, Univ of Technology, Sydney, Copenhagen, ISI, Princeton DSL. Half of the nodes serving as replicas are also foreign or non-educational sites, while the other half are U.S. educational sites. For the results presented in this paper, we use an 8-day long trace measured in July 2003. The sampling interval in this trace is 5 seconds, and the trace contains $150,000$ intervals. Each interval has $7 \times 15 = 105$ samples, resulting in over 15 million samples.

6.2 Unsafety Results

The key property utilized by the witness model is that P_{ni} (probability of non-intersection) can be quite small even with a small number of witnesses. Earlier work [40] verifies this assumption using a number of existing network measurement traces [4, 41]. In the RON1 trace, 5 witness rows result in 4×10^{-5} P_{ni}, while it takes 6 witness rows to yield similar P_{ni} under the TACT trace.

Given these earlier results, this section concentrates on the relationship between P_{ni} and unsafety, namely, how the randomized consensus protocol amplifies P_{ni} into unsafety under different parameter settings. This is important since the protocol has multiple rounds, and non-intersection in any round may result in unsafety.

Unsafety can be affected by several parameters in our system: the message timeout value for contacting witnesses, the size of the witness matrix and the number of

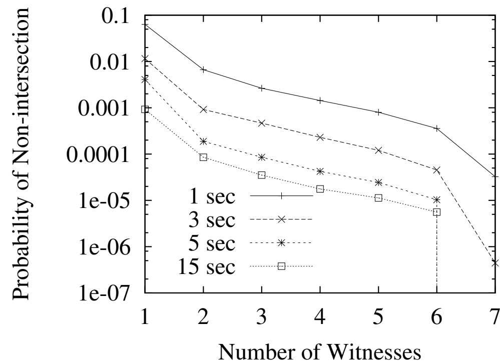

Figure 6: P_{ni} for different time-out values.

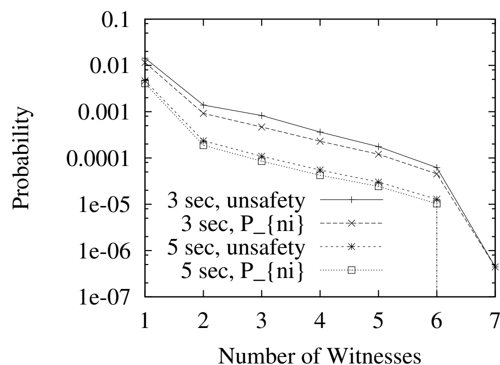

Figure 7: Unsafety and P_{ni}.

replicas. Since a larger t value in the witness matrix is used to guard against potential witness failures and witnesses do not fail in our experiments, we use $t = 1$ for all our experiments. Witness failures between accesses may slightly increase P_{ni}, but a simple analysis can show that such effects are negligible [42] under practical parameters. Larger timeout values decrease the possibility that a replica cannot reach a functioning witness and thus decrease P_{ni}. Figure 6 plots P_{ni} for different timeout values. In our finite-duration experiments, we cannot observe probabilities below 10^{-7}. This is why the curves for 5 and 15 second timeout values drop to zero with seven witnesses. The figure shows that P_{ni} quickly approaches its lowest value with the timeout at 5 seconds.

Having determined the timeout value, we now use emulation to measure unsafety. We first consider the simple case of two replicas. Figure 7 plots both P_{ni} and unsafety for two different timeout values. Using just 7 witnesses, Om already achieves an unsafety of 5×10^{-7}. With 5 replicas and a pessimistic replica MTTF of 12 hours, reconfiguration takes place every 2.4 hours. With unsafety at 5×10^{-7}, an inconsistent reconfiguration would take place once every 500 years. In a peer-to-

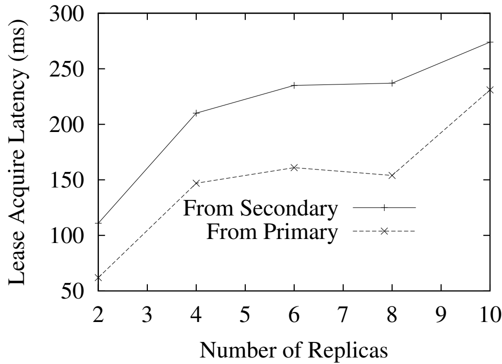

Figure 8: Latency for renewing leases based on our lease graph.

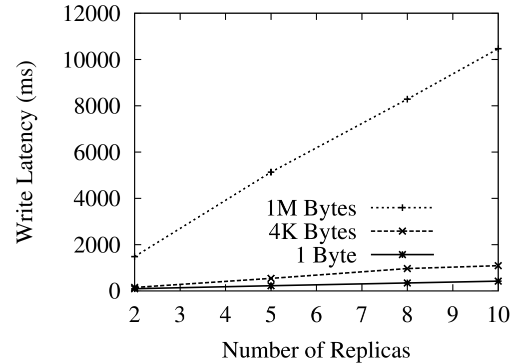

Figure 9: Latency for a write.

peer system with a large number of nodes, reconfiguration can occur much more frequently. For example, for a Pastry ring with $1,000$ nodes and replication degree of 5, each node may be shared by 5 different configurations. As a result, reconfiguration in the entire system occurs every 8.64 seconds. In this case, inconsistent regeneration will take place once every half year system-wide. It may be possible to further reduce unsafety with additional witnesses, though the benefits cannot be quantified with the granularity of our current measurements.

The extended version of this paper [42] further discusses the relationship between unsafety and P_{ni}, and also generalizes the results to more than two replicas. Due to space limitations, we will move on to the performance results.

6.3 Performance Evaluation

We obtain our performance results by deploying and evaluating Om over PlanetLab. In all our performance experiments, we use the seven witnesses used before in our WAN measurement. With single replica regeneration, Om can achieve high availability with a small number of replicas. For example, our analysis [42] shows that Om can achieve 99.9999% availability with just 4 replicas under reasonable parameter settings. Thus, we focus on small replication factors in our evaluation.

6.3.1 Normal Case Operations

We first provide basic latency results for individual read and write operations using 10 PlanetLab nodes as replicas. We intentionally choose a mixture of US educational sites, US non-educational sites and foreign sites. To isolate the performance of Om from that of Pastry, we inject reads and writes from the replicas, instead of having client nodes injecting accesses via peer-to-peer routing.

Since a read in Om is processed by a single replica (as long as it holds all necessary leases), a read involves only a single request/response pair. However, additional latency is incurred when lease renewal is required. To separate these effects, we directly study the latency of lease renewal. However, notice that though not implemented in our prototype, leases can be renewed proactively, which will hide most of this latency from the critical path. Figure 8 plots the time needed to renew leases based on our lease graph. Obviously, the primary incurs smaller latency to renew all of its leases. Secondary replicas need to contact the primary first to request the appropriate set of subleases.

Processing writes is more complex because it involves a two-phase protocol among the replicas. Figure 9 presents the latency for writes of different sizes. In all three cases, the latency increases linearly with the number of replicas, indicating that the network bandwidth of the primary is the likely bottleneck for these experiments. For 1MB writes, the latency reaches 10 seconds for 10 replicas. We believe such latency can be improved by constructing an application-layer multicast tree among the replicas.

6.3.2 Reconfiguration

We next study the performance of regeneration. For these experiments, we use five PlanetLab nodes as replicas: `bu.edu`, `cs.duke.edu`, `hpl.hp.com`, `cs.arizona.edu` and `cs-ipv6.lancs.ac.uk`. Figure 10 shows the cost of failure-free reconfiguration. In all cases, the two components of “finding replica set”

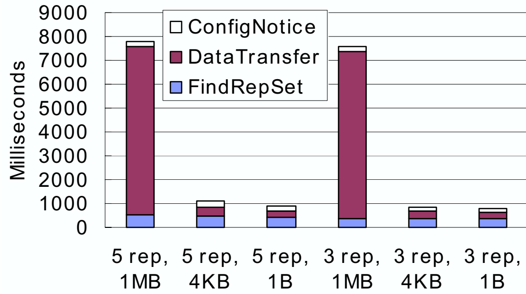

Figure 10: The cost of creating new replicas and invoking a failure-free reconfiguration. All experiments start from a single replica with a data object of a particular size and then expand to either 3 or 5 replicas.

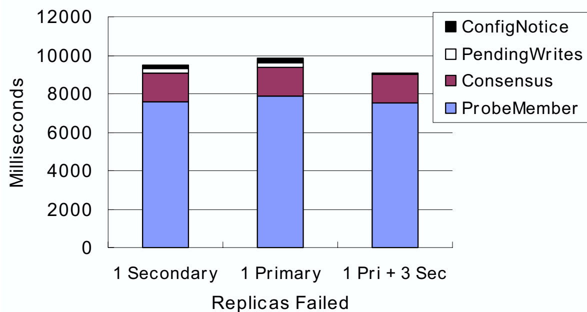

Figure 11: The cost of failure-induced reconfigurations as observed by the primary of the *new* configuration. All experiments start from a five-replica configuration and then we kill a particular set of replicas.

and "sending configuration notices" take less than one second. This is also the cost of failure-free reconfigurations when we shrink instead of expand the replica group. The latency of "finding replica set" is determined by Pastry routing, the only place where Pastry's performance influences the performance of reconfiguration. The time needed to transfer the data object begins to dominate the overall cost with 1MB of data. We thus believe that new replicas should be regenerated in the background using bandwidth consumption controlling techniques such as TCP Nice [39].

The cost of failure-induced reconfiguration is higher. Figure 11 plots the cost of failure-induced reconfiguration as observed by the primary of the *new* configuration. Using optimizations in Section 4.2, only one replica (the one with the smallest ID, which is also the primary of the new configuration) enters the consensus protocol immediately, while other replicas wait for a timeout (10 seconds in our case). As a result of this optimization, in all three cases, the consensus protocol terminates after one iteration (two rounds) and incurs an overhead of roughly 1.5 seconds. The new primary then notifies the other replicas of the resulting configuration. In Figure 11, the time needed to determine the live members of the old configuration dominates the total overhead. This step involves probing the old members and waiting for replies within a timeout (7.5 seconds in our case). A smaller timeout would decrease the delay, but would also increase the possibility of false failure detection and unnecessary replica removal.

Waiting for lease expiration, interestingly, does not cause any delay in our experiments (and thus is not shown in Figure 11). Since we disable lease renewal at the very beginning of the protocol and our lease duration is 15 seconds, by the time the protocol completes the probing phase and the consensus protocol, all leases have already expired. In these experiments, we do not inject writes. Thus, the time for applying pending writes only includes the time for the new primary to collect pending writes from the replicas and then to realize that the set is empty. The presence of pending writes will increase the cost of this step, as explored in our later experiments.

6.3.3 End-to-end Performance

Our final set of experiments study the end-to-end effects of reconfiguration on users. For this purpose, we deploy a 42-node Pastry ring on 42 PlanetLab sites, and then measure the write throughput and latency for a particular object during reconfiguration.

For these experiments, we configure the system to maintain a replication degree of four. To isolate the throughput of our system from the potential bottleneck on a particular network path, we directly inject writes on the primary. Both the writes and the data object are of 80KB size. In the two-phase protocol for writes, the primary sends a total of 240KB data to disseminate each write to the three secondary replicas. For each write, the primary also incurs roughly 9KB of control message overhead.

The experiment records the total number of writes returned for every 5 second interval, and then reports the average as the system throughput. Our test program also records the latency experienced by each write. Writes are rejected when the system is performing a failure-induced reconfiguration.

For our experiment, we first replicate the data object at `cs.caltech.edu`, `cs.ucla.edu`, `inria.fr` and `csres.utexas.edu` (primary). Notice that this replica set is determined by Pastry. Next we manually kill the process running on `inria.fr`, thus causing a

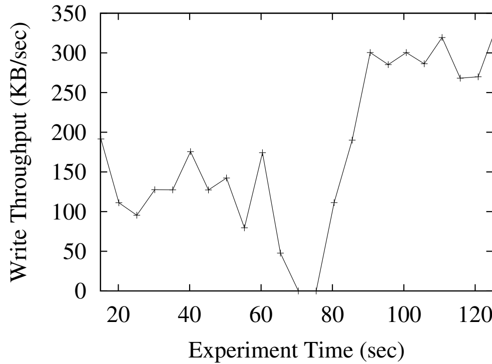

Figure 12: Measured write throughput under regeneration.

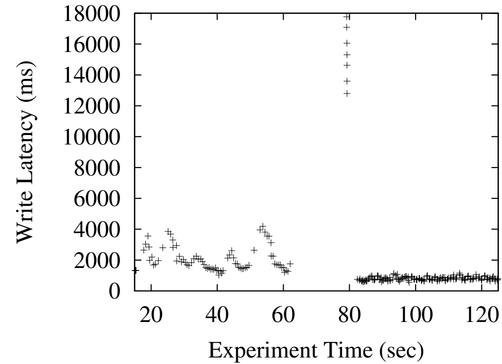

Figure 13: Measured latency of writes. For each write submitted at time t_1 and returning at time t_2, we plot a point $(t_2, t_2 - t_1)$ in the graph.

failure-induced reconfiguration to shrink the configuration to three replicas. Next, to maintain a replication factor of 4, Om expands the configuration to include `lbl.gov`.

Figure 12 plots the measured throughout of the system over time. The absolute throughput in Figure 12 is largely determined by the available bandwidth among the replica sites. The jagged curve is partly caused by the short window (5 seconds) we use to compute throughput. We use a small window so that we can capture relatively short reconfiguration activity. We manually remove `inria.fr` at $t = 62$.

The throughput between $t = 60$ and $t = 85$ in Figure 12 shows the effects of regeneration. Because of the failure at $t = 62$, the system is not able to properly process writes accepted shortly after this point. The system begins regeneration when the failure is detected at $t = 69$. The failure-induced reconfiguration shrinking the configuration takes 13 seconds, of which 3.7 is consumed by the application of pending writes. The failure-free reconfiguration that expands the configuration to include `lbl.gov` takes 1.3 seconds. After the reconfiguration, the throughput gradually increases to its maximum level as the two-phase pipeline for writes fills.

To better understand these results, we plot per-write latency in Figure 13. The gap between $t = 62$ and $t = 82$ is caused by system regeneration when the system cannot process writes (from $t = 62$ to $t = 69$) or rejects writes (from $t = 69$ to $t = 82$). At $t = 80$, those seven writes submitted between $t = 62$ and $t = 69$ return with relatively high latency. These writes have been applied as pending writes in the new configuration.

We also perform additional experiments showing similar results when regenerating three replicas instead of one replica. Overall, we believe that regenerating in 20 seconds can be highly effective for a broad array of services. This overhead can be further reduced by combining the failure detection phase (7 seconds) with the "ProbeMember" phase in failure-induced reconfiguration, potentially reducing the overhead to 13 seconds.

7 Related Work

RAMBO [15, 23] explicitly aims to support reconfigurable quorums, and thus shares the same basic goal as Om. In RAMBO, configuration not only refers to a particular set of replicas, but also includes specific quorum definitions used in accessing the replicas. In our system, the default scheme for data accessing is read-one/write-all. RAMBO also uses a consensus protocol (Paxos [21]) to uniquely determining the next configuration. Relative to RAMBO, our design has the following features. First, RAMBO only performs failure-induced reconfigurations. Second, RAMBO requires a majority of replicas to reconfigure. On the other hand, Om can reconfigure from any single replica at the cost of a small probability of violating consistency. Finally, in RAMBO, both reads and writes proceed in two phases. The first phase uses read quorums to obtain the latest version number (and value, in the case of reads), while the second phase uses a write quorum to confirm the value. Thus, reads in RAMBO are much more expensive than ours. Om avoids this overhead for reads by using a two-phase protocol for write propagation.

A unique feature of RAMBO is that it allows accesses even during reconfiguration. However, to achieve this, RAMBO requires reads or writes to acquire appropriate quorums from all previous configurations that have not been garbage-collected. To garbage-collect a configura-

tion, a replica needs to acquire both a read and a write quorum of that configuration. This means that whenever a *read* quorum of replicas fail, the configuration can never be garbage-collected. Since both reads and writes in RAMBO need to acquire a write quorum, this further implies that RAMBO *completely* blocks whenever it loses a *read* quorum. Om uses lease graphs to avoid acquiring quorums for garbage-collection. If Om uses the same read/write quorums as in RAMBO, Om will regenerate (and thus temporarily block accesses) *only* if RAMBO blocks.

Related to replica group management, there has been extensive study on group communication [3, 5, 19, 24, 28, 29, 31] in asynchronous systems. A comprehensive survey [7] is available in this space. Group communication does not support read operations, and thus does not need leases or a two-phase write protocol. On the other hand, Om does not deliver membership views and does not require view synchrony. The membership in the configuration can not be considered as a view, since we do not impose virtual synchrony relationship between the configurations and writes.

The group membership design in [31] uses ideas similar to failure-free reconfiguration (called *update*) and failure-induced reconfiguration (called *reconfiguration*). However, updates in [31] involve two phases rather than a single phase in our failure-free reconfiguration. In fact, their updates are similar to Om writes. Furthermore, the reconfiguration process in [31] involves re-applying pending "updates". Our design avoids this overhead by using appropriate manipulation [42] on the sequence numbers proposed by failure-free and failure-induced reconfigurations.

In standard replicated state machine techniques [37], all writes go through a consensus protocol and all reads contact a read quorum of replicas. With a fixed set of replicas, a read quorum here usually cannot be a single replica. Otherwise the failure of any replica will disable the write quorum. In comparison, with regeneration functionality and the lease graph, Om is able to use a small read quorum (i.e., a single replica). Om also uses a simpler two-phase write protocol in place of a consensus protocol for normal writes. Consensus is only used for reconfiguration.

Similar to the witness model, voting with witnesses [26] allows the system to compose a quorum with nodes other than the replicas themselves. However, voting with witnesses still uses the simple majority quorum technique and thus always requires a majority to proceed. The same is true for Disk Paxos [14] where a majority of disks is needed.

8 Conclusions

Motivated by the need for consistent replica regeneration, this paper presents Om, the first read/write peer-to-peer wide-area storage system that achieves high availability and manageability through online automatic regeneration while still preserving consistency guarantees. We achieve these properties through the following three novel techniques: i) single replica regeneration that enables Om to achieve high availability with a small number of replicas; ii) failure-free reconfigurations allowing common-case reconfigurations to proceed within a single round of communication; and iii) a lease graph and two-phase write protocol to avoid expensive consensus for normal writes and also to allow reads to be processed by any replica. Experiments on PlanetLab show that consistent regeneration in Om completes in approximately 20 seconds, with the potential for further improvement to 13 seconds.

9 Acknowledgments

We thank the anonymous reviewers and our shepherd, Miguel Castro, for their detailed and helpful comments, which significantly improved this paper.

References

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002.

[2] Akamai Corporation, 1999. `http://www.akamai.com`.

[3] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Subsystem for High Availability. In *Proceedings of the 22nd International Symposium on Fault Tolerant Computing*, pages 76–84, July 1992.

[4] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP)*, October 2001.

[5] K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5:47–76, February 1987.

[6] M. Castro and B. Liskov. Proactive Recovery in a Byzantine-Fault-Tolerant System. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, October 2000.

[7] G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computing Surveys*, 33:1–43, December 2001.

[8] R. Chow and T. Johnson. *Distributed Operating Systems & Algorithms*. Addison Wesley Longman, Inc., 1998.

[9] R. Cohen, K. Erez, D. ben Avraham, and S. Havlin. Resilience of the Internet to Random Breakdowns. *Physical Review Letters*, 85(21), November 2000.

[10] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, October 2001.

[11] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a Common API for Structured Peer-to-peer Overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, February 2003.

[12] A. Fox, S. Gribble, Y. Chawathe, and E. Brewer. Cluster-Based Scalable Network Services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997.

[13] FreePastry. `http://www.cs.rice.edu/CS/Systems/Pastry/FreePastry`.

[14] E. Gafni and L. Lamport. Disk Paxos. In *Proceedings of the International Symposium on Distributed Computing*, pages 330–344, 2000.

[15] S. Gilbert, N. Lynch, and A. Shvartsman. RAMBO II: Rapidly Reconfigurable Atomic Memory for Dynamic Networks. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2003.

[16] M. K. Goldberg. The Diameter of a Strongly Connected Graph (Russian). *Doklady*, 170(4), 1966.

[17] C. Gray and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 202–210, 1989.

[18] M. Herlihy and J. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3), July 1990.

[19] M. F. Kaashoek and A. S. Tanenbaum. Group Communication in the Amoeba Distributed Operating System. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 222–230, May 1991.

[20] J. Kubiatowicz, D. Bindel, Y. Chen, 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 *Proceedings of ACM ASPLOS*, November 2000.

[21] L. Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16:133–169, May 1998.

[22] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1997.

[23] N. Lynch and A. Shvartsman. RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC)*, October 2002.

[24] S. Mishra, L. Peterson, and R. Schlichting. Consul: A Communication Substrate for Fault-tolerant Distributed Programs. *Distributed Systems Engineering*, 1:87–103, December 1993.

[25] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A Read/Write Peer-to-peer File System. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002.

[26] J.-F. Paris. Voting with Witnesses: A Consistency Scheme for Replicated Files. In *Proceedings of the 6th International Conference on Distributed Computer Systems*, pages 606–612, 1986.

[27] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet . In *Proceedings of the ACM HotNets-I Workshop*, 2002.

[28] R. D. Prisco, A. Fekete, N. Lynch, and A. Shvartsman. A Dynamic Primary Configuration Group Communication Service. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC)*, September 1999.

[29] R. Renesse, K. Birman, R. Cooper, B. Glade, and P. Stephenson. The Horus System. *K.P. Birman and R. van Renesse, editors, Reliable Distributed Computing with the Isis Toolkit*, pages 133–147, September 1993.

[30] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore Prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, March 2003.

[31] A. Ricciardi and K. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *Proceedings of the 10th ACM Symposium of Principles of Distributed Computing*, pages 341–352, 1991.

[32] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-peer Systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, November 2001.

[33] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 188–201, October 2001.

[34] Y. Saito, B. Bershad, and H. Levy. Manageability, Availability and Performance in Porcupine: A Highly Scalable Internet Mail Service. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, December 1999.

[35] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming Aggressive Replication in the Pangaea Wide-area File System. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002.

[36] M. Saks, N. Shavit, and H. Woll. Optimal Time Randomized Consensus – Making Resilient Algorithms Fast in Practice. In *Proceedings of the Second Symposium on Discrete Algorithms*, pages 351–362, January 1991.

[37] F. B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, pages 299–319, December 1990.

[38] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM 2001*, pages 149–160, August 2001.

[39] A. Venkataramani, R. Kokku, and M. Dahlin. TCP Nice: A Mechanism for Background Transfers. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[40] H. Yu. Overcoming the Majority Barrier in Large-Scale Systems. In *Proceedings of the 17th International Symposium on Distributed Computing (DISC)*, October 2003.

[41] H. Yu and A. Vahdat. The Costs and Limits of Availability for Replicated Services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, October 2001.

[42] H. Yu and A. Vahdat. Consistent and Automatic Replica Regeneration. Technical Report IRP-TR-04-01, Intel Research Pittsburgh, 2004. Also available at `http://www.intel-research.net/pittsburgh/publications.asp`.