Consistent and Automatic Replica Regeneration

HAIFENG YU
Intel Research Pittsburgh/Carnegie Mellon University
and
AMIN VAHDAT
University of California San Diego

Reducing management costs and improving the availability of large-scale distributed systems require automatic replica *regeneration*, that is, 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 article 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 during each regeneration. 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, so that reads in Om can be processed by any single replica. Experiments on PlanetLab show that consistent regeneration in Om completes in approximately 20 seconds.

Categories and Subject Descriptors: D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems*

General Terms: Algorithms, Design, Experimentation, Reliability

Additional Key Words and Phrases: Peer-to-peer storage systems, availability, consistency, replication, regeneration

A preliminary version of this article was presented at the Usenix Symposium on Networked Systems Design and Implementation (NSDI'04), March 2004.

Authors' addresses: H. Yu, Intel Research Pittsburgh, Pittsburgh, PA 15214; email: yhf@cs.cmu.edu; A. Vahdat, University of California, San Diego, La Jolla, CA 92093.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.
© 2004 ACM 1553-3077/04/1200-0003 $5.00

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-based Internet services [Fox et al. 1997; Saito et al. 1999]. Similarly, automatic regeneration has become a necessity in emerging large-scale distributed systems [Adya et al. 2002; Dabek et al. 2001; Kubiatowicz et al. 2000; Muthitacharoen et al. 2002; Rhea et al. 2003; Rowstron and Druschel 2001b; Saito et al. 2002]. 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 [Dabek et al. 2001] and PAST [Rowstron and Druschel 2001b]), or simply enforces consistency in a best-effort manner (e.g., Inktomi [Fox et al. 1997]; Porcupine [Saito et al. 1999]; Ivy [Muthitacharoen et al. 2002]; Pangaea [Saito et al. 2002]). Among those replication systems [Adya et al. 2002; Castro and Liskov 2000; Kubiatowicz et al. 2000; Rhea et al. 2003; Schneider 1990] that do provide strong consistency guarantees, Farsite [Adya et al. 2002] does not implement replica group reconfiguration. Oceanstore [Kubiatowicz et al. 2000; Rhea et al. 2003] mentions automatic reconfiguration as a goal, but does not detail its approach, design, or implementation. Proactive recovery [Castro and Liskov 2000] enables the same replica to leave the replica group and later rejoin, but still assumes a fixed set of replicas. Finally, replicated state-machine research [Schneider 1990] 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 [Rowstron and Druschel 2001b] and CFS [Dabek et al. 2001], 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 [Akamai Corporation 1999] 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* [Yu 2003], 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 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} out of each regeneration. This means that on average, inconsistent regeneration 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 (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 [Peterson et al. 2002], we observe that regeneration in response to replica failures only causes a 20-second service interruption.

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

Fig. 1. A configuration.

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. Section 6 uses an analytical approach to study the availability benefits of regeneration and single copy regeneration. We present unsafety (probability of violating consistency during a regeneration) and performance evaluation in Section 7. Finally, Section 8 discusses related work and Section 9 draws our conclusions.

2. SYSTEM ARCHITECTURE OVERVIEW

2.1 Naming and Configurations

Om relies on Distributed Hash Tables (DHTs) [Rowstron and Druschel 2001a; Stoica et al. 2001] for naming its objects. The current implementation of Om uses FreePastry (http://www.cs.rice.edu/CS/Systems/Pastry/FreePastry). Om invokes only two common peer-to-peer APIs [Dabek et al. 2003] 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. The **route**() API sends the request to the key's current root, which in turn uses **replicaSet**() to determine the current replica group. The root then sends back a reply containing this set. 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.

Om servers are grouped into configurations (Figure 1). Each configuration contains the set of servers holding copies of a particular object. To optimize for reads and allow the user to read the object by contacting only one server, we use replication in Om, rather than, for example, erasure coding. Thus a server in a configuration holds a full copy of the 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 article, we use *linearizability* [Herlihy and Wing 1990] 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. Such wall-clock time is defined by an imaginary, global wall-clock. 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 [Saito et al. 2002]. 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 targets (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 (Section 6) to coordinate regeneration, we adopt the witness model [Yu 2003] 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 article.

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 that are 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. We now 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 [Gray and Cheriton 1989]. 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. Also in Om, lease validity is always with respect to a particular configuration. It is possible that node A holds an unexpired lease from B, but the lease is still invalid with respect to a particular configuration. This either means that A's configuration is too old and A should initiate reconfiguration, or that the lease from B was obtained when B was in an old configuration.

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 [Goldberg 1966] 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, a 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 and sends an acknowledgment to the primary. Finally, after collecting all the acknowledgments from the replicas, 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 acknowledgment indicates that the write will ultimately be seen by all or none of the replicas. A user may choose to resubmit an unacknowledged 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. This 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).

Upon receiving a configuration notice, a replica adopts the new configuration if its current configuration has a smaller sequence number. When this happens, the replica will abort any reconfiguration that is based on older configurations, and avoid unnecessary work. We design our reconfiguration protocols so that the protocol can be aborted from any point without affecting correctness.

Instead of using dedicated messages for configuration notices, we attach the current configuration to each lease renewal request. In many cases, the first thing a replica needs to do after establishing a new configuration is to obtain leases. Thus, configuration notices incur little additional overhead. In the following, however, we assume dedicated configuration notices to simplify discussion.

4.1 Failure-Free Reconfiguration

Only the primary may initiate failure-free reconfiguration. Secondary replicas are involved only when i) the primary transmits data to them 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;
}
```

Fig. 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 reapply any pending writes. When reapplying 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).

4.3 Avoiding Interference Between Failure-Free and Failure-Induced Reconfigurations

A critical issue in the two reconfiguration protocols is determining the `sequenceNum` and `consensusID` for the new configuration. For `sequenceNum`, a naive design would simply increment it for each new configuration. Following, we construct a scenario to show that this affects correctness. Suppose the original configuration has sequence number i and four replicas, A (primary), B, C, and D. D notifies A that it will exit, and then does so, after receiving an acknowledgment from A. A now initiates a failure-free reconfiguration with the new configuration consisting of A (primary), B, and C. It sends to B and C the notice for the new configuration with sequenceNum $i+1$. B receives and adopts the new configuration. However, the message to C is delayed. C believes there are failures and performs a failure-induced reconfiguration. Suppose C decides that the new configuration should contain only C itself. The new configuration of C also has a sequence number of $i + 1$. Now imagine that the configuration notice from A to C finally arrives. C will have no way to determine whether it should adopt this configuration, since it has the same sequence number as its local configuration.

This problem arises from the interference between failure-free and failure-induced reconfigurations. To address this problem, an important observation is that in our design, only the primary can initiate failure-free reconfigurations, and it will do so after the previous reconfiguration finishes. As a result, at any time, there is at most one failure-free reconfiguration in progress. Thus despite message delay and loss, the `sequenceNum` of the configurations on different replicas can differ by one, at most.

Based on such observation, we can simply have failure-induced reconfigurations increase the sequence number by two (Figure 2), instead of one. Doing so ensures that the new configuration, proposed by failure-induced reconfigurations, always has a larger sequence number than the configuration proposed by failure-free ones. For our previous example, the new configuration of C will have a sequence number of $i + 2$, and C will not adopt the configuration notice from A. When A and B try to obtain leases from C, they will not be able to obtain valid leases based on their configuration (with sequence number of $i + 1$). They will then also start a failure-induced reconfiguration. Their proposal for the new configuration will have a sequence number of $i + 3$. However, the consensus protocol ensures that they reach the same decision as C, namely, the configuration (proposed by C) with sequence number of $i + 2$. At this point,

A and B will retire themselves from the configuration since they are not included in the new configuration. Excluding A and B is a desirable result since C may have received new writes, and the data on A and B can be stale. It is possible for A and B to request to rejoin the configuration, in which case C will perform a failure-free reconfiguration to include A and B as new replicas.

The previous example also reveals the subtlety in choosing `consensusID`. It is obvious that in failure-induced reconfigurations, we should choose a new `consensusID`. On the other hand, it is also important (but less clear) that we should continue to use the old `consensusID` in failure-free reconfigurations. We can use the old `consensusID` because it has never been used. Further, if we were to use a new `consensusID` and a failure-induced reconfiguration were to take place in the middle of the process, the replicas would not be running the same instance of the consensus protocol. In our example, when A and B fail to obtain valid leases from C, they should enter the same instance of the consensus protocol that C entered earlier. Only by doing so can they determine that the new configuration consists of only C.

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 [Lynch 1997] is a classic distributed computing problem and we can conceptually use any consensus protocol in Om. However, most consensus protocols such as Paxos [Lamport 1998] 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* [Yu 2003] to achieve probabilistic consensus without requiring a majority.

5.1 Probabilistic Quorum Intersection Without Majority

The witness model [Yu 2003] 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 simply Om nodes, other than the replicas in the new configuration. Since there are a large number of such nodes in the system, we only need to choose $m \times t$ of them as witnesses. Notice that witnesses for a given configuration are actually replicas of other configurations. 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 [Stoica et al. 2001].

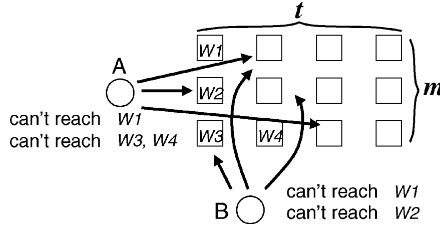

Fig. 3. Two replicas and 3 × 4 witnesses.

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 colocated 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 the 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 Yu [2003], using the RON [Andersen et al. 2001] and TACT [Yu and Vahdat 2001] 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 [Cohen et al. 2000] 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 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 [Yu 2003].

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 an 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. Nonintersection in the witness model is ultimately translated into the *unsafety* (probability of having multiple decisions) of a consensus protocol. Unsafety, in turn, means inconsistent regeneration in Om. For protocols with multiple rounds, unsafety potentially increases with every round. This precludes the application of protocols such as Paxos [Lamport 1998] 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 [Saks et al. 1991], where the expected number of rounds before termination is constant and thus helps to bound unsafety.

A straightforward way to emulate a read/write atomic shared-memory is to use the protocol in Yu [2003]. Such emulation requires one round of communication between the replicas and the witnesses for each memory write, and two rounds for each memory read. The second round for read is used to ensure the atomicity of the memory. The shared-memory randomized consensus protocol [Saks et al. 1991] always performs writes and reads in pairs, which enables us to combine the first round of a read with the corresponding write, resulting in two rounds for each write/read pair.

Interestingly, we find that the shared-memory consensus protocol actually requires weaker guarantees from the memory than atomicity. However, to exploit such an optimization, we need to abandon the standard notion of reads and writes on shared-memory. 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 implementing, where each access only takes a single round of communication between the replicas and the witnesses. If we substitute each write/read pair with a single access operation, then the

```
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;
}
```

Fig. 4. Emulating shared-memory under the witness model.

message (and time) complexity of the shared-memory emulation is reduced by half. 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. Later we will prove that such change does not impact correctness.

Following, we accurately describe the semantics of the shared-memory as implemented in Figure 4. Consider an access A_1 that writes value v to the pth element of the array. Another access A_2 *sees* A_1, if in the array returned by A_2, the pth element is v. Two accesses, A_1 and A_2, *intersect* if A_1 sees A_2, or A_2 sees A_1. A set of accesses *intersect* if any two accesses in the set intersect. Let P_{ni} be the probability of non-intersection in the witness model. The semantics of the shared-memory is abstracted in the following: *If we use the protocol in Figure* 4 *to perform two accesses,* A_1 *and* A_2*, then* A_1 *and* A_2 *intersect with probability of at least* $1 - P_{ni}$.

5.3 Application of Shared-Memory Randomized Consensus Protocol

With the shared-memory abstraction, we can now apply a previous shared-memory consensus protocol [Saks et al. 1991] (Figure 5). For simplicity, we

```
// 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';
    }
}
```

Fig. 5. Randomized consensus protocol for shared-memory.

assume that the proposals and decisions are all integer values, though they are actually configurations. In the figure, we have already substituted the read and write operations in the original protocol with our new access operations.

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 [Chow and Johnson 1998]). Since the protocol itself is not a contribution of this article, we only focus on its high-level properties. Proof will be provided in the next section. The protocol proceeds in successive iterations, and 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. Nonintersection 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.

For the following discussion, we will focus on the case where there are only two distinct proposals. With n replicas, the number of distinct proposals can be as large as n. More distinct proposals increase the complexity of the protocol. For example, with $\theta(n)$ distinct proposals, the protocol will have $O(n)$ complexity, instead of $O(1)$ complexity as proven later in Theorem 2. It is possible to optimize the protocol so that the complexity is $O(\log n)$, instead of $O(n)$, even with n distinct proposals. However, such optimization will result in $O(\log n)$ complexity for two distinct proposals as well. We choose not to use such optimization, and also choose to focus our analysis for two distinct proposals because we expect, in most cases, there will be no more than two distinct proposals. This is because i) the proposals are proposals for the next configuration, and many replicas will likely have the same proposal; ii) the total number of replicas in

each configuration tends to be small given Om's single replica regeneration functionality; and iii) with the optimization in Section 4.2, in many cases only one replica will enter the consensus protocol.

With two distinct proposals with symbolic names "0" and "1", we can simplify the statement "myvalue = a random element in prop_view indexed by **coinFlip**()" to be "myvalue = **coinFlip**()", where **coinFlip**() simply returns "0" or "1" at random. When **coinFlip**() is implemented with a local random number generator, at each iteration, there is at least $1/2^n$ probability that all replicas choose the same new value for the next iteration, and thus terminate the protocol. A simple calculation can show that the expected number of iterations before termination is 2^n.

To reduce the exponential complexity, many researchers have investigated *shared coins* [Aspnes and Waarts 1996; Bracha and Rachman 1991; Saks et al. 1991], which enable the replicas to choose the same new value with a constant probability. As a result, a shared coin protocol helps the consensus protocol to terminate in a constant expected number of iterations. However, shared coin protocols require communication and incur at least $O(n^2)$ time complexity.

In our design, we implement **coinFlip**() using a local random number generator, *initialized using a common seed shared by all replicas*. All replicas will thus generate, the same random sequence. This construct provides a weaker guarantee than a standard shared coin protocol. Namely, it needs to assume that the timing of the system (including instruction execution speed and message propagation) is not affected by the value of the random seed until the coin flip result is used by the program. Otherwise, the property on probabilistic termination will be compromised. However, we argue that this is a reasonable assumption, since the seed is only used in the random number generator before the random result is generated. As long as the random number generator only uses integer addition, integer multiplication and bit-wise operations, the timing will be independent of the seed value. (Notice that we need to exclude division operations since they may incur division-by-zero interrupts.) Many pseudo random number generators satisfy such properties, including the one used in Java 1.4.

With such design and 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.

We also apply an optimization to the protocol to remove a significant assumption in Yu [2003], where the intersection of access pairs in different rounds are assumed to be independent. Such assumption may or may not hold in practice, since a past nonintersection may indicate a higher probability of nonintersection in the future. To remove this assumption, we simply force the protocol to terminate after R iterations, where R is a tunable constant. Doing this slightly increases unsafety, but enables us to bound unsafety, even with correlated nonintersections. This also achieves a better termination property (in fact, deterministic termination).

5.4 Proof

This section will show that our proposed optimization on shared-memory emulation (Section 5.2) does not compromise correctness. We will also analyze the complexity and unsafety of the consensus protocol.

An iteration of the consensus protocol is *correct* if i) all accesses on `proposed[i]` intersect; and ii) all accesses on `check[i]` intersect. Similarly, a round is *correct* if all accesses in a particular round (either on `proposed[i]` or on `check[i]`) intersect. A replica is *correct* if it does not crash. We prove the following lemma, which is adopted from the original lemma for atomic shared-memory [Saks et al. 1991].

LEMMA 5.1. *For any iteration i:*

(1) *If all replicas that start iteration i have the same* `myvalue`*, then all correct replicas will decide on that value in iteration i.*
For any correct iteration i:
(2) *All replicas that write* `agree` *to* `check[i]` *wrote the same value to* `proposed[i]`.
(3) *If any replica decides on a value v in iteration i, then each correct replica that completes iteration i sees at least one* `agree` *in* `check_view`.
(4) *Every replica that sees at least one* `agree` *in its* `check_view` *completes iteration i with the same* `myvalue`.
(5) *If any replica decides on a value v, in iteration i, then each correct replica will decide in some iteration* $i' \leq i + 1$.

PROOF.

(1) If all replicas in iteration i have the same `myvalue`, then all correct replicas will write `agree` into `check[i]`, and the array `check_view` on any replica will contain only `agree`. As a result, all correct replicas decide at the end of iteration `i`.
(2) Prove by contradiction. Suppose both p and q write `agree` to `check[i]`, but they wrote different values to `proposed[i]`. Let A_p denote p's access to `proposed[i]`, and A_q denote q's access to `proposed[i]`. Since iteration i is correct, by definition, A_p and A_q intersect. Without loss of generality, suppose A_p sees A_q. Then p must have seen different values in its `prop_view`, and it will not write `agree` to `check[i]`. Contradiction.
(3) Suppose replica p decides on a value in iteration i. Prove by contradiction. Suppose replica q does not see `agree` in its `check_view`. Obviously, p writes `agree` into `check[i][p]`, otherwise it would not decide. Also, q writes `disagree` into `check[i][q]`, otherwise q would have seen one `agree` (from itself). Let A_p denote p's access to `check[i]`, and A_q denote q's access to `check[i]`. Since A_p and A_q must intersect and q does not see `agree`, A_p must have seen A_q. However, this means that p sees at least one `disagree`, and it will not decide. Contradiction.
(4) Directly follows from Claim 2.

(5) From Claim 3, every correct replica sees at least one `agree`. From Claim 4, all correct replicas complete iteration i with the same `myvalue`. Finally, Claim 1 tells us that all replicas will decide in or before iteration $i+1$. □

LEMMA 5.2. *Suppose all iterations before protocol termination are correct, then all correct replicas decide on the same value.*

PROOF. Prove by contradiction. Suppose p and q decide on different values, and consider two cases:

(1) p decides in an iteration before q decides. From Lemma 5.1 (Claim 3, Claim 4, and Claim 1), we know that q will decide on the same value in the next iteration. Contradiction.
(2) p and q decide in the same iteration. Obviously, both p and q write `agree` to `check[i]`. From Lemma 5.1 (Claim 2), they write the same value to `proposed[i]`, which means they have the same `myvalue` in that iteration. As a result, they decide on the same value. Contradiction. □

THEOREM 5.3. *Let* $P_{iter} = n(n-1)P_{ni}$:

(1) *Our consensus protocol has expected time complexity of* $3 + (R^2 + 3R - 2) \times P_{iter}/2$ *iterations and unsafety of* $(R+1)P_{iter} + 1/2^{R-1}$.
(2) *When* $R = 0.47 - 1.44 ln P_{iter}$, *unsafety reaches its global minimum of* $2.88P_{iter} - 1.44P_{iter} ln P_{iter}$.
(3) *When* $R = 0.47 - 1.44 ln P_{iter}$ *and* $P_{iter} < 0.001$, *expected time complexity is smaller than* 3.1 *iterations* (6.2 *rounds*).

PROOF. The probabilistic analysis in this proof is pessimistic because we cannot assume independence among various events. We first analyze time complexity. Lemma 5.1, Claim 4, shows that if iteration i is correct, then there is probability of 0.5 that all replicas finishing iteration i have the same `myvalue`. In such case, iteration i becomes a *deciding iteration*, and the protocol will terminate in or before iteration $i+1$.

Let $P_{round} = n(n-1)/2 \times P_{ni}$. Clearly, a round is correct with at least probability of $1 - P_{round}$, since there are at most $n(n-1)/2$ pairs of accesses that need to intersect in a correct round. Similarly, an iteration is correct with at least probability of $1 - 2P_{round} = 1 - P_{iter}$.

Let B_i be the event that iteration i is correct, that is, $B_i = 1$, if and only if iteration i is correct. For iteration i, define A_i to be the event that all replicas have the same `myvalue` at the end of iteration i *as if iteration i were correct*. All A_i's are mutually independent by property of the random number generator, but there may exist dependence among B_i's, and also between B_i and A_j ($j \leq i$). Let C_i denote whether iteration i is a deciding iteration. Define $x_i = P[C_1 = C_2 = \cdots = C_i = 0]$. We have:

$$x_0 = 1$$
$$x_1 = 1/2 + P_{iter}/2.$$

For x_2, notice that:

$$\begin{aligned} x_2 &= P[C_1 = C_2 = 0] \\ &= P[A_2 = 0 \text{ and } C_1 = 0] \\ &\quad + P[A_2 = 1 \text{ and } B_2 = 0 \text{ and } C_1 = 0] \\ &= P[A_2 = 0] \times P[C_1 = 0] + P[B_2 = 0] \\ &\quad \times P[A_2 = 1 | B_2 = 0] \\ &\quad \times P[C_1 = 0 | (B_2 = 0 \text{ and } A_2 = 1)] \\ &= x_1/2 + P_{iter}/2 \\ &\quad \times P[C_1 = 0 | (B_2 = 0 \text{ and } A_2 = 1)]. \end{aligned}$$

Since the value of $P[C_1 = 0|(B_2 = 0 \text{ and } A_2 = 1)]$ must be within 0 and 1, we have $x_1/2 \le x_2 \le x_1/2 + P_{iter}/2$. Such relation can be extended to all iterations, and we will have the following:

$$\begin{aligned} x_0 &= 1 \\ x_1 &= 1/2 + P_{iter}/2 \\ x_1/2 \le x_2 &\le x_1/2 + P_{iter}/2 \\ &\cdots \\ x_{R-1}/2 \le x_R &\le x_{R-1}/2 + P_{iter}/2. \end{aligned}$$

Some simple calculation can show that $1/2^i \le x_i \le 1/2^i + P_{iter}$. Define $y_i = P[C_1 = C_2 \cdots = C_{i-1} = 0 \text{ and } C_i = 1]$. We have:

$$\begin{aligned} y_i &= x_{i-1} - x_i \\ &\le 1/2^{i-1} + P_{iter} - 1/2^i \\ &= 1/2^i + P_{iter}. \end{aligned}$$

The expected number of iterations before termination is then:

$$\begin{aligned} &\sum_{i=1}^{R} P[\text{Terminate in iteration } i] \times i \\ &\quad + R \times P[\text{Does not terminate within iteration } R] \\ &= R \times x_{R-1} + \sum_{i=1}^{R-1} (i+1) \times y_i \\ &\le R \times (1/2^{R-1} + P_{iter}) + \sum_{i=1}^{R-1} (i+1)/2^i \\ &\quad + P_{iter} \sum_{i=1}^{R-1} (i+1) \\ &= R/2^{R-1} + \sum_{i=1}^{R-1} (i+1)/2^i + \frac{R^2 + 3R - 2}{2} \times P_{iter}\ . \end{aligned}$$

Next, we will show that $R/2^{R-1} < \sum_{i=R}^{\infty}(i+1)/2^i$. This is true because for any z:

$$\begin{aligned}\sum_{i=R}^{\infty}(i+1)z^i &= \left(\sum_{i=R}^{\infty} z^{i+1}\right)' \text{ (derivative for } z\text{)}\\ &= \left(\frac{z^{R+1}}{1-z}\right)'\\ &= \frac{(R+1)z^R - Rz^{R+1}}{(1-z)^2}.\end{aligned}$$

When $z = 1/2$, we have $\sum_{i=R}^{\infty}(i+1)/2^i = (2R+4)/2^R > R/2^{R-1}$. From this result, we have:

$$\begin{aligned}& R/2^{R-1} + \sum_{i=1}^{R-1}(i+1)/2^i + \frac{R^2+3R-2}{2} \times P_{iter}\\ <& \sum_{i=1}^{\infty}(i+1)/2^i + \frac{R^2+3R-2}{2} \times P_{iter}\\ =& \ 3 + \frac{R^2+3R-2}{2} \times P_{iter}.\end{aligned}$$

For unsafety, notice that the probability of all iterations from iteration 1 to iteration R being correct is at least $1 - R \times P_{iter}$. The protocol does not terminate within iteration R with probability of at most $x_{R-1} \le 1/2^{R-1} + P_{iter}$. From Lemma 5.2, the total unsafety is bounded from above by:

$$\begin{aligned}& R \times P_{iter} + 1/2^{R-1} + P_{iter}\\ =& \ (R+1) \times P_{iter} + 1/2^{R-1}.\end{aligned}$$

It can easily be shown that the above unsafety reaches its minimal when:

$$R = \frac{ln ln 4 - lnP_{iter}}{ln 2} = 0.47 - 1.44 lnP_{iter},$$

and the minimal unsafety achieved is $2.88P_{iter} - 1.44P_{iter}lnP_{iter}$. □

6. AVAILABILITY ANALYSIS OF OM

This section uses analytical approaches to study the availability of Om, and in particular, the benefits of regeneration and single replica regeneration. We choose to use an analytical approach because experimental evaluation of availability for regenerative systems poses fundamental challenges. For example, at high availability levels, the experiments may require unrealistically long duration to observe enough failure events. Furthermore, a human subject may need to be involved throughout the evaluation to repair the system as necessary [Brown et al. 2002].

We consider two types of systems: *regenerative systems* with automatic regeneration functionality, and *static replication systems* without such functionality. We consider three types of events: *failure*, *regeneration*, and *repair*. Regeneration is the system's action to heal after replica failures. Repair by a human

being is necessary for static replication systems, or regenerative systems when they cannot regenerate (for example, because of the loss of majority). We assume that once human repair is invoked, all failures in the system are repaired when the repair completes. Failure, regeneration, and repair are all modeled using exponential distributions.

We consider n replicas that fail independently with a failure rate of λ. Let the regeneration rate be μ_g, and the repair rate be μ_p. For regenerative systems, we assume that the system can regenerate with k or more functioning replicas. The value of k is determined by the quorum system used during the regeneration (Section 2.2). For example, if we use a majority quorum system to enforce strict consistency, then $k = \lceil (n+1)/2 \rceil$. With the witness model and single replica regeneration in Om, $k = 1$.

For regenerative systems, availability analysis must account for both regeneration and repair. Here we assume that the system remains available during regeneration (enabled by background regeneration in Om), but becomes unavailable when human repair is in progress. To deal with two streams of repairs, we first analyze the mean time to the first human repair (MTTF). To derive such MTTF, we only consider regeneration and analyze the MTTF of an imaginary system that can always regenerate. The MTTF of this imaginary system will be the same as the mean time to the first human repair of the real system. (See Sahner et al. [1996] for a comprehensive treatment on reliability analysis and full discussion on such technique.) In the imaginary system, the steady state availability of each replica is $A = \mu_g/(\lambda + \mu_g)$. System availability is $A_{eq} = \sum_{i=k}^{n} \binom{n}{k} (1-A)^{n-i} A^i$. Let $1/\lambda_{eq}$ be the MTTF of the imaginary system. We have:

$$\lambda_{eq} = \frac{P[k \text{ replicas are up}] \times k\lambda}{\text{System Availability}}, \\ = \frac{\binom{n}{k}(1-A)^{n-k}A^k \times k\lambda}{A_{eq}}.$$

Given that the mean time to human repair is $1/\lambda_{eq}$, the availability of real regenerative system is then $\mu_p/(\lambda_{eq} + \mu_p)$.

The availability of a static replication system can be computed by simply replacing μ_g with μ_p in the previous analysis. We also make the advantageous assumption for static replication systems that $k = 1$.

Based on the previous analysis, Figure 6 plots the availability of three replication systems. We intentionally use a small MTTF for the nodes, because our regeneration design targets recovery from not only real node failures, but also long network outages. Based on a diverse set of traces, Dahlin et al. [2003] show that for network failures lasting more than 30 seconds, the MTTF is 48111 seconds (or roughly 12 hours). The jagged curve for the majority regenerative system results from the majority requirement for regeneration. For example, three replicas can tolerate one failure, and having four replicas will not increase such fault tolerance (since a majority is three). On the other hand, the probability of having more that one failure out of three replicas is smaller. This is why the availability under three replicas is actually better than the availability under four replicas.

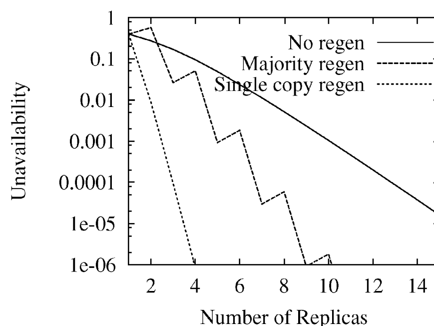

Fig. 6. The unavailability of three replication systems: static replication with no regeneration, regeneration from any majority of replicas, and regeneration from any single replica (such as Om). Node MTTF: 12 hours. Mean time to regenerate: 5 minutes. Mean time for human repair: 8 hours.

The graph shows that regeneration, in general, has significant advantage over static replication systems. Further notice that the amount of human repair is also decreased in regenerative systems, resulting in better manageability. Compared with regenerative systems that require a majority, single replica regenerative systems (such as Om) need only one half or one third of the replicas in order to achieve the same availability. Smaller number of replicas not only means improved write performance, but also reduces regeneration activities. On the other hand, with the same number of replicas, Om can achieve much better availability (and incurs less human repair). Such benefit is significant even with a small number of replicas (such as two or three). Similar observations are valid for other reasonable values for the parameters.

7. EXPERIMENTAL EVALUATION

This section evaluates the performance and unsafety of Om. 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. Our design does not rely on TCP's reliable and in-order delivery, and we use TCP to simplify the fragmentation and composition of serialized objects. Because of the use of serialization, our message size tends to be larger than strictly necessary. The core of Om uses an event-driven architecture.

7.1 Unsafety Evaluation

Om is able to regenerate from any single replica at the cost of a small probability of inconsistent regeneration. 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 [Yu and Vahdat 2001]. 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 Yu and Vahdat [2001] 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 noneducational 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 noneducational sites, while the other half are U.S. educational sites. For the results presented in this article, 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.

7.2 Unsafety Results

The key property utilized by the witness model is that P_{ni} (probability of nonintersection) can be quite small, even with a small number of witnesses. Earlier work [Yu 2003] verifies this assumption using a number of existing network measurement traces [Andersen et al. 2001; Yu and Vahdat 2001]. 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 nonintersection 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 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}, which will be discussed later. Larger timeout values decrease the possibility that a replica cannot reach a functioning witness, and thus decreases P_{ni}. On the other hand, it also degrades performance under true witness failures since the replica cannot return until the timeout expires.

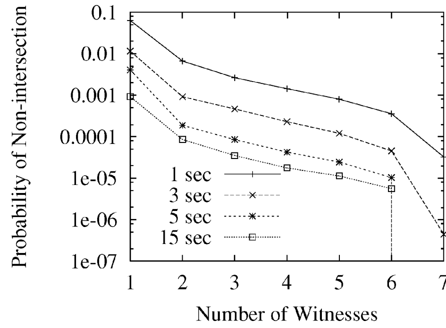

Fig. 7. P_{ni} for different timeout values.

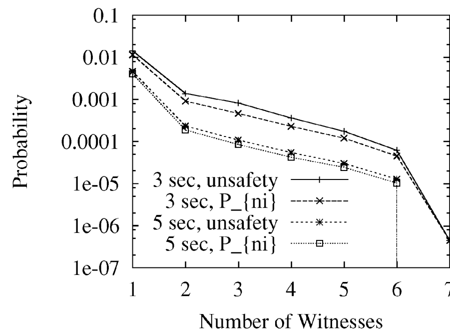

Fig. 8. Unsafety and P_{ni}.

Figure 7 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. Our results are slightly optimistic since witnesses do not crash in the measurements. Witness failures before the consensus protocol starts are masked using a $t > 1$. Formal analysis is available in Yu [2003]. On the other hand, witness failures in the middle of the protocol will slightly increase P_{ni}. However, even with a pessimistic 12-hour MTTF and 6-second protocol execution time, witness failures will occur during protocol execution with a probability of below 0.001. Figure 7 shows that each witness contributes roughly a 0.1 factor in decreasing P_{ni}. Taking witness failures into account will only bring such factor down to 0.099.

Having determined the timeout value, we now use emulation to measure unsafety. We first consider the simple case of two replicas. Figure 8 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-peer system with a large number of nodes, reconfiguration can occur much more frequently. For example, for a Pastry ring with

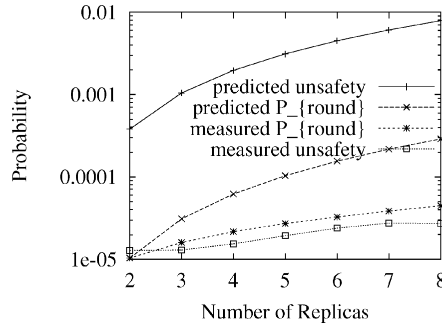

Fig. 9. Unsafety and P_{ni} for more than two replicas. The predicted results are computed based on Theorem 2. Notice that we intentionally use 6 witnesses with 5 second timeout value, which results in relatively large unsafety. This allows us to observe more rare events and reduce inaccuracy. Thus, we focus on the trends in these results rather than the absolute values.

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.

Figure 8 also shows that unsafety is very close to P_{ni}, implying that the consensus protocol does not significantly amplify the error. On the other hand, our analysis (Theorem 2) shows that unsafety can be much larger that P_{ni}. The discrepancy comes from the fact that, in many cases, nonintersection in a round does not necessarily result in unsafety. Our analysis further overstates unsafety because of the pessimistic analysis on termination. Recall that unsafety can potentially be amplified at each round. Thus the larger the total number of rounds, the higher the unsafety. Theorem 2 shows that the expected number of rounds before termination is below 6.2. In our experiments, the average number of rounds before termination is roughly 4, and this value is rather insensitive to the number of witnesses and replicas. Also notice that in our emulation, we do not use the optimization of having replicas wait before entering the protocol (Section 4.2), which will further reduce the complexity to 2 rounds (Section 7.3.2).

Figure 9 confirms our previous observations beyond two replicas. P_{round} is defined as the possibility of those rounds where nonintersection occurs. Clearly, P_{round} equals P_{ni} for two replicas, and it increases with the number of replicas. For these experiments, each replica proposes a distinct value. This makes it more difficult for the protocol to terminate than the case with just two distinct proposals (as assumed by Theorem 2). Interestingly, the measured unsafety is even below P_{round} with three or more replicas. This does not occur for two replicas because nonintersection in the *first* round of the protocol will always result in unsafety for two replicas. With more than two replicas, even if some pair of replicas do not intersect, it is possible that a third replica first makes a decision. Then the two replicas will both adopt the proposal of the third replica, thus preserving safety. Such effects become increasingly significant with more

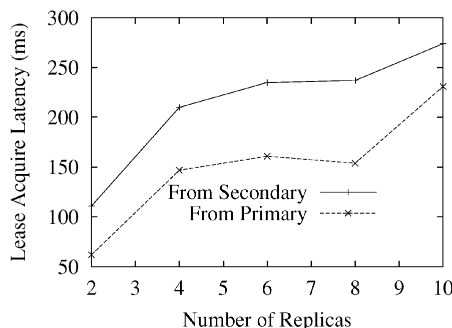

Fig. 10. Latency for renewing leases based on our lease graph.

replicas, which explains why the gap between measured unsafety and measured P_{round} increases with the number of replicas.

7.3 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 (Section 6) 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.

7.3.1 *Normal Case Operations.* We first provide basic latency results for individual read and write operations using 10 PlanetLab nodes as replicas. In Pastry, node IDs are created using a one-way hash and replicas must be those nodes with the closest IDs to the object ID. To control the nodes on which data is replicated, we only start a Pastry ring with the same number of nodes as the replication degree. We intentionally choose a mixture of US educational sites, US noneducational 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 10 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 11 presents the latency for writes of different sizes. In all three cases, the latency increases linearly with the number of replicas,

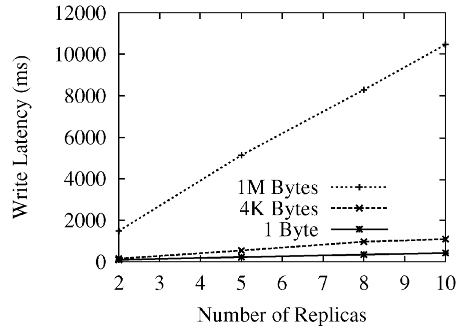

Fig. 11. Latency for a write.

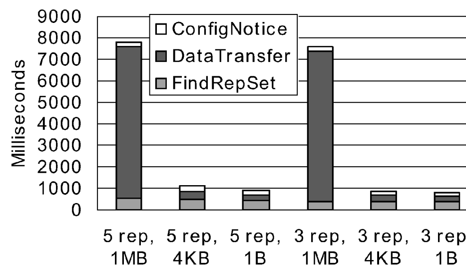

Fig. 12. The cost of creating new replicas and invoking a failure-free reconfiguration. All experiments start from a single replica with a data object of a particular size, and then expand to either 3 or 5 replicas.

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. Because of the reconfiguration capability of Om, such a tree can be easily established by the primary for each new configuration, significantly decreasing the complexity of tree construction.

7.3.2 *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 12 shows the cost of failure-free reconfiguration. In all cases, the two components of "finding replica set" 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 believe, therefore, that new replicas should be regenerated in the background using bandwidth consumption-controlling techniques such as TCP Nice [Venkataramani et al. 2002]. This will, of course, increase the time to regenerate and affect availability. But even with 2-hour regeneration time, based on our analysis in Section 6 and with the same values for other parameters, we still

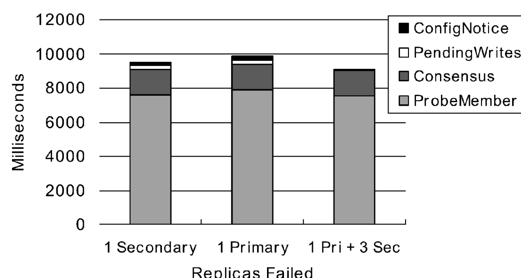

Fig. 13. The cost of failure-induced reconfigurations as observed by the primary of the *new* configuration. All experiments start from a five-replica configuration, and then we kill a particular set of replicas.

only need 8 replicas to achieve 99.9999% availability. Also note that failure-free reconfigurations do not block accesses, so the system is still available during the regeneration process.

The cost of failure-induced reconfiguration is higher. Figure 13 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. As a result, the total reconfiguration delay observed on all the other replicas is approximately the same as the primary, except that after probing members (7.5 seconds), they wait for the decision. In Figure 13, 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 13). 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. Finally, when the new configuration contains only one replica (last experiment in Figure 13), applying pending writes and sending out configuration notices involve only local processing.

7.3.3 *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.

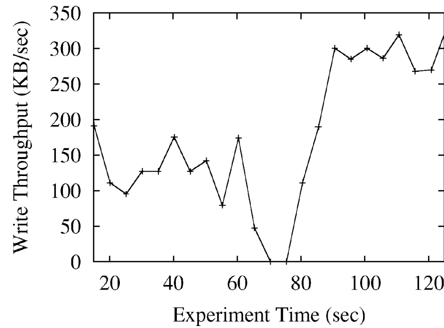

Fig. 14. Measured write throughput under regeneration.

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`, causing a 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 14 plots the measured throughout of the system over time. The absolute throughput in Figure 14 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 $t = 62$. In the configuration with the `inria.fr` site, that site's bandwidth is clearly the bottleneck. This explains why after regeneration (when the `inria.fr` site is replaced by `lbl.gov`), the system throughput roughly doubles. Furthermore, the throughput in the first configuration is less stable than the second because the available bandwidth is more variable from `inria.fr`.

The throughput between $t = 60$ and $t = 85$ in Figure 14 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

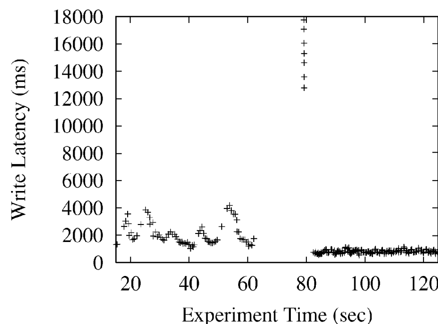

Fig. 15. Measured latency of writes. For each write submitted at time t_1 and returning at time t_2, we plot a point $(t_2, t_2 - t_1)$ in the graph.

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 15. For the first configuration, the variation in latency confirms our previous argument that the network path to the `inria.fr` site is unstable. 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. Notice that even though the system only starts accepting writes again at $t = 82$, the throughput at $t = 80$ is already nonzero because those pending writes are applied and count toward throughput.

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.

8. RELATED WORK

RAMBO [Gilbert et al. 2003; Lynch and Shvartsman 2002] 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 [Lamport 1998]) to uniquely determine 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. Next, RAMBO does not have a primary to serialize writes, rather a client directly contacts a quorum. Because of this, RAMBO only allows

primitive write operations, that is, writes that simply write a new value. In our system, a write can be an arbitrary update transaction that reads and then modifies the data based on some computation. 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 configuration, a replica needs to acquire both a read and a write quorum of that configuration. This means that whenever a *read* quorum of replicas fails, 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. Under such design, however, Om would require at least a read quorum of replicas to regenerate. But this is still strictly better than RAMBO where regeneration requires both a read and a write quorum.

Related to replica group management, there has been extensive study on group communication [Amir et al. 1992; Birman and Joseph 1987; Kaashoek and Tanenbaum 1991; Mishra et al. 1993; Prisco et al. 1999; Renesse et al. 1993; Ricciardi and Birman 1991] in asynchronous systems. A comprehensive survey [Chockler et al. 2001] is available. 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. This helps to explain the relationship between Om (which fundamentally reduces replica group reconfiguration to consensus) and the impossibility result in Schiper and Sandoz [1994], which proves that primary-partition group communication *cannot* be reduced to consensus. We avoid this impossibility in a similar fashion as in Babaoglu et al. [1994] and Bartoli and Babaoglu [1997], where even though multiple primary views can be delivered, the application will be able to perform action in only one of them.

Om's reconfiguration protocol can also be used in group communication. We here compare our reconfiguration component with previous group communication systems. Depending on their behavior under partitions, group communication services can be classified into *partitionable* group communication services and *primary-partition* group communication services. One exception to such classification is Amoeba [Kaashoek and Tanenbaum 1991], which allows the user to choose between a partitionable service and a primary-partition service. Partitionable group communication services [Amir et al. 1992; Kaashoek and Tanenbaum 1991; Renesse et al. 1993] allow simultaneous progress in the

face of network partitions or false failure detection, potentially introducing inconsistency. On the other hand, primary-partition group communication services [Birman and Joseph 1987; Kaashoek and Tanenbaum 1991; Mishra et al. 1993; Prisco et al. 1999; Ricciardi and Birman 1991] allow progress only in a single partition, but require a majority of nonfaulty nodes to proceed. In comparison, Om can regenerate even with a single nonfaulty node, while allowing a small probability of violating consistency.

While many early group communication systems (e.g., ISIS [Birman and Joseph 1987], Amoeba [Kaashoek and Tanenbaum 1991], and Consul [Mishra et al. 1993]) claim to allow network partitions, they only allude to using majority quorums to guarantee the uniqueness of the next configuration in case of failures. However, such a step in fact requires a consensus protocol such as Paxos, and is nontrivial. Recent group communication systems [Prisco et al. 1999] all use variants of three-phase commit [Keidar and Dolev 1995; Skeen 1982] (similar to Paxos) for reconfiguration. None of these systems distinguishes between failure-free and failure-induced reconfigurations.

The group membership design in Ricciardi and Birman [1991] uses ideas similar to failure-free reconfiguration (called *update*) and failure-induced reconfiguration (called *reconfiguration*). However, updates in Ricciardi and Birman [1991] involve two phases, rather than a single phase as in our failure-free reconfiguration. In fact, their updates are similar to Om writes. Furthermore, the reconfiguration process in Ricciardi and Birman [1991] involves reapplying pending "updates". Our design avoids this overhead by using appropriate manipulation manipulation on the sequence numbers proposed by failure-free and failure-induced reconfigurations.

In standard replicated state machine techniques [Schneider 1990], 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 [Paris 1986] 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 [Gafni and Lamport 2000], where a majority of disks is needed.

9. CONCLUSIONS

Motivated by the need for consistent replica regeneration, this article 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.

REFERENCES

ADYA, A., BOLOSKY, W. J., CASTRO, M., CERMAK, G., CHAIKEN, R., DOUCEUR, J. R., HOWELL, J., LORCH, J. R., THEIMER, M., AND WATTENHOFER, R. P. 2002. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*.

AKAMAI CORPORATION. 1999. http://www.akamai.com.

AMIR, Y., DOLEV, D., KRAMER, S., AND MALKI, D. 1992. Transis: A communication subsystem for high availability. In *Proceedings of the 22nd International Symposium on Fault Tolerant Computing*. 76–84.

ANDERSEN, D., BALAKRISHNAN, H., KAASHOEK, F., AND MORRIS, R. 2001. Resilient overlay networks. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP)*.

ASPNES, J. AND WAARTS, O. 1996. Randomized consensus in expected $O(n\ log^2 n)$ operations per processor. *SIAM J. Comput. 25*, 5 (Oct.), 1024–1044.

BABAOGLU, O., BARTOLI, A., AND DINI, G. 1994. Replicated file management in large-scale distributed systems. In *Workshop on Distributed Algorithms*. 1–16.

BARTOLI, A. AND BABAOGLU, O. 1997. Selecting a "primary partition" in partitionable asynchronous distributed systems. In *Symposium on Reliable Distributed Systems*. 138–145.

BIRMAN, K. AND JOSEPH, T. 1987. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst. 5*, 47–76.

BRACHA, G. AND RACHMAN, O. 1991. Randomized consensus in expected $O(n^2 logn)$ operations. *In Proceedings of the 5th International Workshop on Distributed Algorithms*.

BROWN, A., CHUNG, L., AND PATTERSON, D. 2002. Including the human factor in dependability benchmarks. In *DSN Workshop on Dependability Benchmarking*.

CASTRO, M. AND LISKOV, B. 2000. Proactive recovery in a Byzantine-fault-tolerant system. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*.

CHOCKLER, G. V., KEIDAR, I., AND VITENBERG, R. 2001. Group communication specifications: A comprehensive study. *ACM Comput. Surv. 33*, 1–43.

CHOW, R. AND JOHNSON, T. 1998. *Distributed Operating Systems & Algorithms*. Addison Wesley Longman, Inc.

COHEN, R., EREZ, K., BEN AVRAHAM, D., AND HAVLIN, S. 2000. Resilience of the Internet to random breakdowns. *Phys. Rev. Letters 85*, 21 (Nov.).

DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. 2001. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*.

DABEK, F., ZHAO, B., DRUSCHEL, P., KUBIATOWICZ, J., AND STOICA, I. 2003. Towards a common API for structured peer-to-peer overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*.

DAHLIN, M., CHANDRA, B., GAO, L., AND NAYATE, A. 2003. End-to-end WAN service availability. *ACM/IEEE Trans. Network. 11*, 2 (April).

FOX, A., GRIBBLE, S., CHAWATHE, Y., AND BREWER, E. 1997. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. Saint-Malo, France.

FREEPASTRY. http://www.cs.rice.edu/CS/Systems/Pastry/FreePastry.

GAFNI, E. AND LAMPORT, L. 2000. Disk paxos. In *Proceedings of the International Symposium on Distributed Computing*. 330–344.

GILBERT, S., LYNCH, N., AND SHVARTSMAN, A. 2003. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*.

GOLDBERG, M. K. 1966. The diameter of a strongly connected graph (Russian). *Doklady 170*, 4.

GRAY, C. AND CHERITON, D. 1989. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*. 202–210.

HERLIHY, M. AND WING, J. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (July).

KAASHOEK, M. F. AND TANENBAUM, A. S. 1991. Group communication in the Amoeba distributed operating system. In *Proceedings of the 10th International Conference on Distributed Computing Systems*. 222–230.

KEIDAR, I. AND DOLEV, D. 1995. Increasing the resilience of atomic commit, at no additional cost. In *Proceedings of the ACM Symposium of Principles of Database Systems*.

KUBIATOWICZ, J., BINDEL, D., CHEN, Y., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. 2000. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*.

LAMPORT, L. 1998. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 133–169.

LYNCH, N. 1997. *Distributed algorithms*. Morgan Kaufmann Publishers.

LYNCH, N. AND SHVARTSMAN, A. 2002. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC)*.

MISHRA, S., PETERSON, L., AND SCHLICHTING, R. 1993. Consul: A communication substrate for fault-tolerant distributed programs. *Distrib. Syst. Engineer. 1*, 87–103.

MUTHITACHAROEN, A., MORRIS, R., GIL, T., AND CHEN, B. 2002. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*.

PARIS, J.-F. 1986. Voting with Witnesses: A consistency scheme for replicated files. In *Proceedings of the 6th International Conference on Distributed Computer Systems*. 606–612.

PETERSON, L., ANDERSON, T., CULLER, D., AND ROSCOE, T. 2002. A blueprint for introducing disruptive technology into the internet . In *Proceedings of the ACM HotNets-I Workshop*.

PRISCO, R. D., FEKETE, A., LYNCH, N., AND SHVARTSMAN, A. 1999. A dynamic primary configuration group communication service. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC)*.

RENESSE, R., BIRMAN, K., COOPER, R., GLADE, B., AND STEPHENSON, P. 1993. The Horus System. In *Reliable Distributed Computing with the Isis Toolkit*, K. P. Birman and R. van Renesse, Eds. 133–147.

RHEA, S., EATON, P., GEELS, D., WEATHERSPOON, H., ZHAO, B., AND KUBIATOWICZ, J. 2003. Pond: the OceanStore prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*.

RICCIARDI, A. AND BIRMAN, K. 1991. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the 10th ACM Symposium of Principles of Distributed Computing*. 341–352.

ROWSTRON, A. AND DRUSCHEL, P. 2001a. 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)*.

ROWSTRON, A. AND DRUSCHEL, P. 2001b. 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*. 188–201.

SAHNER, R. A., TRIVEDI, K. S., AND PULIAFITO, A. 1996. *Performance and Reliability Analysis of Computer Systems*. Kluwer Academic Publishers.

SAITO, Y., BERSHAD, B., AND LEVY, H. 1999. Manageability, availability and performance in Porcupine: A highly scalable internet mail service. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*.

SAITO, Y., KARAMANOLIS, C., KARLSSON, M., AND MAHALINGAM, M. 2002. Taming aggressive replication in the Pangaea wide-area file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*.

SAKS, M., SHAVIT, N., AND WOLL, H. 1991. Optimal time randomized consensus—Making resilient algorithms fast in practice. In *Proceedings of the 2nd Symposium on Discrete Algorithms*. 351–362.

SCHIPER, A. AND SANDOZ, A. 1994. Primary partition “virtually-synchronous communication” harder than consensus. In *Proceedings of the 8th International Workshop on Distributed Algorithms (WDAG-8)*. 39–52.

SCHNEIDER, F. B. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 299–319.

SKEEN, D. 1982. A quorum-based commit protocol. In *Proceedings of the Berkeley Workshop on Distributed Data Management and Computer Network*.

STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, F., AND BALAKRISHNAN, H. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM 2001*. 149–160.

VENKATARAMANI, A., KOKKU, R., AND DAHLIN, M. 2002. TCP nice: A mechanism for background transfers. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*.

YU, H. 2003. Overcoming the majority barrier in large-scale systems. In *Proceedings of the 17th International Symposium on Distributed Computing (DISC)*.

YU, H. AND VAHDAT, A. 2001. The costs and limits of availability for replicated services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*.

Received September 2004; revised September 2004; accepted September 2004