Local Recovery for High Availability in Strongly Consistent Cloud Services

James W. Anderson, Hein Meling, Alexander Rasmussen, Amin Vahdat, and Keith Marzullo

Abstract—Emerging cloud-based network services must deliver both good performance and high availability. Achieving both of these goals requires content replication across multiple sites. Many cloud-based services either require or would benefit from the semantics and simplicity of strong consistency. However, replication techniques for strong consistency can severely limit the availability of replicated services when recovering large data objects over wide-area links.

To address this problem, we present the design and implementation of ZORFU, a hierarchical system architecture for replication across data centers. The primary contribution of ZORFU is a *local recovery* technique that significantly increases availability of replicated strongly consistent services. Local recovery achieves this by reducing the recovery time by an order of magnitude, while imposing only a negligible latency overhead. Experimental results show that ZORFU can recover a 100MB object in 4ms.

Index Terms—Wide-area state machine replication; Hierarchical replication; Paxos; Local recovery; Dependability analysis.

1 INTRODUCTION

Traditional desktop applications, such as word processing, email, and photo management are increasingly moving to server-based deployments. However, moving applications to the cloud can reduce availability because Internet path availability averages only two-nines [1]. If a user's application state is isolated on a single server, the availability for that user is limited by the path availability between the user's desktop and that server. Hence, to improve availability, application state must be replicated across multiple servers placed in geographically distributed data centers.

Replicating state across data centers, however, makes it harder to maintain consistency across updates. Maintaining strong consistency [2] is essential for correct system behavior for many cloud-based services. Other services that tolerate weak consistency [3] may still benefit from the simplified semantics offered by strong consistency. Examples include collaborative applications, electronic commerce, and financial analysis. The need for strong consistency has also been recognized by prominent cloud providers [4], [5], [6].

Providing strong consistency for data hosted at multiple sites is difficult because different updates, possibly from different users, could be directed at servers in different data centers. Even if updates are directed to the same server, the order in which multiple updates are committed to application state must be the same across all copies of the state.

- *J. W. Anderson, A. Rasmussen, A. Vahdat and K. Marzullo are with the Department of Computer Science and Engineering, University of California San Diego. E-mail: jwanderson@gmail.com, alexras@acm.org, vahdat@cs.ucsd.edu, marzullo@cs.ucsd.edu.*
- *H. Meling is with the Department of Electrical Engineering and Computer Science, University of Stavanger, Norway. E-mail: hein.meling@uis.no.*

Strong consistency is typically achieved using a Replicated State Machine (RSM) model [2], based on a consensus algorithm such as Paxos [7], [8] to order state machine operations. In this context, a fundamental requirement for strong consistency in RSMs is that $2f + 1$ replicas are needed to tolerate f failures. Moreover, we target real-world deployment with billions of objects [9] stored across several data centers, each with tens of thousands of machines. At this scale, there is constantly a need to recover from common machine failures, and to do so without human intervention.

In this paper, we address the problem of automated recovery of RSMs across geographically distributed data centers connected by a wide-area network. In this scenario, we identify a *window of vulnerability* while recovering from a failure, during which a subsequent failure can cause the RSM to block indefinitely. This situation demands manual recovery, which would significantly reduce the system's availability. This can happen if more than f failures occur before completing recovery from previous failures. It can also happen if the RSM state is not synchronized with a replacement replica before a subsequent failure occurs. Thus, despite the availability of at least $f + 1$ replicas, application-level RSM state is not synchronized sufficiently quickly to allow the RSM to safely[1] process updates. The existence of this window of vulnerability directly affects system availability, and it becomes particularly problematic when objects are large or synchronization takes place over relatively slow or congested wide-area links. These are exactly the scenarios we target.

The principal contribution of this work is ZORFU, a system architecture for hierarchical replication designed to increase RSM availability by reducing the window of vulnerability that occurs during failure recovery. To

1. An RSM needs $f + 1$ replicas to make progress, and remain safe.

reach this goal, ZORFU uses asynchronous replication between a primary and a backup within each data center, along with synchronous replication between primaries across data centers.

A second contribution facilitated by ZORFU's architecture is a *local recovery* mechanism that can quickly replace a failed replica, insulating machine failures within a data center from replicas in other data centers. Our local recovery mechanism makes use of a scalable and fault tolerant software switch to hide the internal replica allocations within each data center. This allows us to recover replica failures locally, without incurring costly wide-area RSM reconfigurations, resulting in a significantly shorter window of vulnerability. Our experimental results show that our switch adds only a negligible delay compared to the wide-area latency. Typically, a switch is shared across multiple RSMs, depending on the load requirements of each RSM. To cope with a larger number of RSMs, we simply can deploy additional switches.

Our third contribution is an analytic dependability analysis of ZORFU's impact on system availability, compared to two other design alternatives (Section 7). We show that our local recovery mechanism provides significant availability gain over the other designs.

The paper proceeds as follows. Section 2 explains the reasons for the window of vulnerability, and argues that a traditional flat replication scheme is not suitable for wide-area replication. Section 3 describes the system model, and gives an overview of ZORFU, followed by a detailed description in Section 4. ZORFU is described as a stepwise refinement, starting from a Paxos-based state machine implementation [7]. In Section 5 we present important design details pertaining to the software switch and a controller infrastructure used by ZORFU. Section 6 discuss design choices and resource costs of ZORFU. Section 7 contains our analytic dependability analysis. In our experimental evaluation in Section 8, we emulated a data center environment with ModelNet [10] in order to test the recovery time achievable with local recovery. We show that a 100MB object can be recovered in 4ms (excluding detection time), which directly translates to increased system availability. Our analysis also show that we achieve these availability improvements while still maintaining high performance. Finally, we discuss related work in Section 9 and conclude the paper.

2 MOTIVATION

An RSM provides the illusion of a single highly available state machine by maintaining a copy of that state machine at multiple replicas [7], [11]. Any replica may receive updates at any time. To maintain the same state across replicas, an RSM uses a consistent replicated log [12], [13], [14] to totally order updates with a consensus protocol such as Paxos. The replicated log assigns to every operation an *index* number representing the position in the log at which the operation should be executed by the RSM. We say that the index number of an operation is *chosen* once a quorum of replicas agree on its index number, ensuring a totally-ordered sequence of operations. The consensus *safety* property states that at most one update will be chosen for any index. As we focus on the Paxos consensus algorithm, we refer to this safety property as *Paxos safety*. The consensus *liveness* condition states that the system can only make progress when a quorum of replicas are available. More precisely, to maintain liveness, a quorum of replicas must be available for every index.

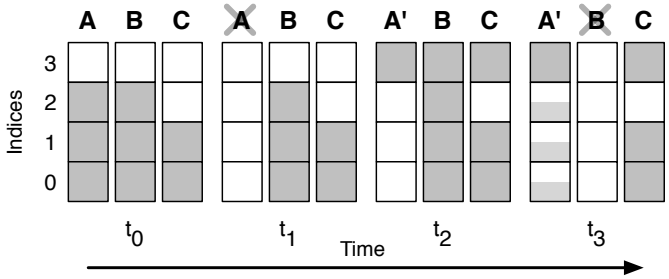


Fig. 1. State loss due to failure during recovery.

Consider the scenario in Figure 1 with replicas A, B, C. At time t_0, some instances of consensus have already completed and replica C lags behind A and B (perhaps because it never received messages for index 2 due to a transient network error). At t_1, replica A permanently fails. To regain fault tolerance, replicas B and C add a replacement replica A' to the group at t_2 for index 3. A' has only learned the chosen value for the index that added it to the group, but it can participate in consensus for indices > 3. To acquire the remaining state, A' begins copying the prior values from B and C. At t_3, B fails and loses its state. Replica C has not yet copied index 2, and replica A' has only partially copied indices $0..2$. The system has lost the value for index 2. To maintain safety, it must block permanently or request a human to resolve potential conflicts. Note that, index 3 for consensus could have blocked until A' had recovered all of its state or C has copied index 2, but B's failure would still cause the system to lose state and block permanently.

One can reduce the probability of losing state by decreasing the time between replica failure and the full availability of a new replica. This translates to improved system availability. At first glance, a simple approach to improve system availability would be to add more replicas in a flat replication scheme, by increasing f, the number of tolerable failures. However, the message complexity of consensus protocols is linear with the number of replicas, and the latency is bounded by the slowest replica in the consensus quorum. While these requirements may not matter when the replicas are on the same LAN, these factors can significantly hurt performance with replicas separated by high-latency, low bandwidth links. For example, with $f = 1$, the consensus latency is only that of the closer of the two replicas. If we use $f = 2$, consensus would require waiting for two wide-area replies, with increasing possibility of one

taking longer than the other. Note that this cost is borne even when there are no failures. Ideally, we want a system that does not incur this additional cost in latency or bandwidth.

While we consider a permanent failure model in this paper, we note that, it *may* be possible to simply wait for the machine to reboot to recover missing state. In this scenario, the amount of wide-area communication needed to re-synchronize the replica would be limited to any missing updates during the reboot period. Unfortunately, reboot can take several tens of seconds in the best case and since the fundamental problem with the window of vulnerability remains, the system could become unacceptably prone to unavailability as further analyzed in Section 7.

3 OVERVIEW

We first state our system model and assumptions, which is followed by a brief overview of ZORFU.

3.1 System Model

ZORFU provides the abstraction of a collection of highly-available RSMs, which higher-level services can read or update. Each RSM has its own Paxos-based replicated log and associated replica set. The RSM stores application state needed by cloud-based services. To meet the demands of typical cloud services, ZORFU must be able to support a large number of RSMs simultaneously; for example, one might imagine the billions of objects stored in Amazon's S3 [9] each being backed by an RSM.

To achieve its target of high availability, ZORFU replicates RSM replicas across multiple data centers. Each data center may contain up to tens of thousands of machines [15], each of which may host relevant application and replicated log state for many RSMs.

A client interacts with ZORFU by issuing requests to a data center through the Internet. Clients may submit requests to any data center hosting a replica, and the set of data centers replicating an RSM may change dynamically in response to failures. Typically, clients are directed to their nearest data center in terms of latency.

We assume links connecting data centers are well-provisioned and do not contribute to any significant downtime. For simplicity, we ignore link downtime from our analytic evaluation (Section 7). However, we note that being well-provisioned, these links are different from links between clients and data centers, for which the general Internet path availability [1] is applicable.

Failure and Recovery Model: Traditionally, consensus protocols assume nodes fail by crashing, but may later recover with their state intact. In this work, we assume a stronger notion of failure in which a machine irrevocably loses all of its state when it fails (such as a disk loss). We assume such a failure model because it is a pessimistic assumption that allows us to explore worst-case recovery times (Section 7). Specifically, we define a failed server to be one that some other server believes to have crashed and will never recover. As such, failed servers must be replaced to maintain the same degree of fault tolerance. As Paxos, ZORFU does not need a perfect failure detector.

Consistency Model: Unlike other wide-area replicated services that offer high availability but only provide weak consistency guarantees [3], ZORFU uses a consensus algorithm to provide *strong consistency*. Requests to a given RSM are executed in the same order at all replicas.

3.2 ZORFU Overview

We now give a brief overview of the core mechanisms used by ZORFU to achieve its goal of high availability for wide-area replicated services. ZORFU's design focuses on decreasing the recovery time by using:

1) Hierarchical replication and
2) Local recovery within each data center (site).

By localizing recovery within the site of the failed replica, we can keep the number of wide-area message exchanges small, both for the common case and when there are failures. This is a key contributor to keeping the recovery time small, even when the RSM state is large. Also, having fewer wide-area exchanges in the critical-path reduces the latency observable by clients.

To facilitate local recovery, we need more replicas at each site. This calls for hierarchical replication [16], [17], where we distinguish between replication across sites and replication within a site. In ZORFU, we use *synchronous replication* between sites, and *asynchronous replication* within each site. Each site has a *primary* replica and one or more *backup* replicas (we consider only one backup here). The primaries at the different sites maintain a replicated log, which is kept synchronously updated with a Paxos-based RSM protocol. Within each site, the local primary asynchronously sends updates to the backups. ZORFU uses asynchronous replication to limit the amount of bandwidth needed to update the backups; however, this means that the backups may lag slightly behind the primary. To resolve this issue, and to facilitate local recovery, we develop a fault-tolerant and scalable software switching service, as discussed next.

The switching service may consist of several switch devices, as shown in Figure 2, each acting as both a message cache and message forwarder. A single switch is typically shared across several RSMs. A switch directs messages to the appropriate RSM primary, in a similar manner to that employed by front-end load balancers present in most data centers. In addition, a switch also acts as a message cache for certain Paxos messages sent by the primary. Should a primary fail, ZORFU can promote one of its backups to become the new primary by sending it the Paxos state in the switch message cache. This local recovery is transparent to the replicas at the other sites and can be completed in milliseconds, even for very large objects, requiring no wide-area messages. This transparency is made possible by the switching service and its addressing scheme, designed to isolate local replicas from those in other sites (see Section 5.3).

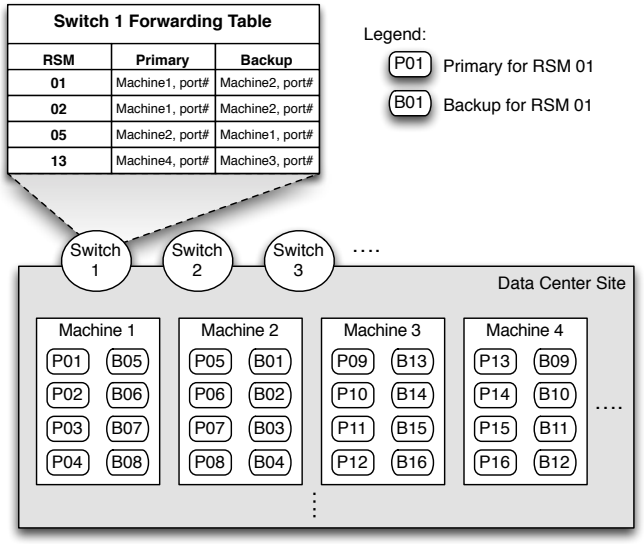


Fig. 2. An example data center configuration with primary and backup replicas, switches, and a forwarding table.

Figure 2 illustrates a simple allocation of primary and backup replicas to machines in a data center. This allocation is a dynamic optimization problem that needs to consider several constraints. These constraints may include the expected message load for each individual RSM and switch, the available compute resources on machines, and so on. Similar optimization problems have been studied extensively in previous work [18], [19]. Here the focus is on new techniques for amortizing the cost of replication across machines hosting RSMs and switches. The exact choice of allocation of RSMs and switches is highly application dependent and dynamic, and is beyond the scope of this paper.

4 The Zorfu Protocol

We describe the ZORFU protocol as a stepwise refinement starting from the Paxos state machine [7]. In Step 0 we review Paxos and some optimizations commonly used in Paxos-based RSMs. We then introduce a backup replica in Step 1 and a switch in Step 2. Then in Step 3 we explain how the switch interacts with the primary and backup replicas to facilitate local recovery. In Steps 4-6 we add mechanisms aimed at reducing overhead. In each step we argue that the additions and changes we make have no adverse effect on the correctness of Paxos.

Step 0: Paxos

Paxos [7] is a consensus algorithm that can be used to consistently order client requests for an RSM.

Paxos is often described in terms of three separate agent roles: *proposers* that can propose values for consensus, *acceptors* that accept a value among those proposed, and *learners* that learn the chosen value. A process may implement multiple roles, and in a typical configuration the proposer, acceptor, and learner combine to form a Paxos server, or *replica*.

For a given index in a replicated log, Paxos provides the safety property that at most one operation will ever be chosen among a set of $2f + 1$ replicas. Due to this property, we can say that Paxos is safe for any number of crash failures. Moreover, an RSM based on Paxos can remain operational despite f crashes.

Paxos is used to consistently order operations for an RSM. For every index, Paxos will try to decide on a value that represents the operation to be executed at that index. To decide on a value, Paxos may run one or more rounds. Typically, one round is enough, but due to asynchrony and failures, multiple rounds may be necessary.

Every round is associated with a single proposer replica, which is the *leader* for that round. A proposer can start a new round by sending a PREPARE message to the acceptors, requesting that they promise not to accept any old messages. Essentially, every round runs in two phases: (1) A proposer collects a quorum of PROMISE messages from acceptors in response to a previously sent PREPARE message; and (2) the proposer then sends ACCEPT messages for some value v to acceptors, who respond by sending ACCEPTED messages to learners. The proposer selects the value v for the ACCEPT message as the value with the highest round among those provided in the PROMISE messages. Or if no values are provided in the PROMISE messages, any value can be chosen for v. A value is said to be *chosen* when it has been *accepted* by a quorum of replicas. A replica learns a value once it has seen a quorum of ACCEPTED messages for that value. If more than f replicas fail simultaneously, then the RSM cannot make progress, in order to preserve safety.

Paxos optimizations: If a stable leader starts a new index, knowing that no other leader will send PREPARE or ACCEPT messages for that index, the leader can safely propose any value, and thus can skip the first phase. The leader also serves as a distinguished learner that sends CHOSEN messages to the other replicas after receiving a quorum of ACCEPTED responses. Next, Paxos also allows for a simple optimization; if a replica receives a PREPARE or ACCEPT for which it knows the chosen value, it responds with a CHOSEN message containing this value. Finally, if a replica believes that another replica S has failed, it may propose a reconfiguration command to have S removed from the RSM and replaced by a new replica S'. Eventually, S is removed and S' is added to the RSM. This is called a Paxos *reconfiguration* [7], [20], [21] and is essential to ZORFU.

Step 1: Introducing a Backup

In the ZORFU model, the *primary* implements the traditional Paxos replica. Each *replica site* (data center) has one primary replica per RSM. Our first change is to introduce a backup replica for each primary, i.e. one backup per replica site. The backup runs on a separate machine in the same site as its primary. We assume that each site has a pool of machines available to host replicas and that each machine may act as both primary or backup for multiple RSMs. Figure 2 shows an example where

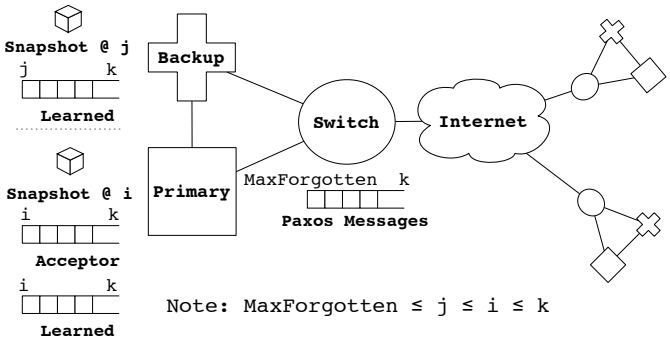


Fig. 3. ZORFU components and their queues. Each queue may hold different sets of messages, hence the different indices.

multiple primaries are allocated to the same machine, and their corresponding backups are mapped to other machines in the same site.

The backup is a standby replica used to speed up the replacement of a failed primary. It is updated asynchronously by the primary, and hence may lag behind. The primary simply stores received Paxos messages in a queue and later forwards them to the backup. Thus, the addition of a backup does not introduce any protocol changes from the perspective of the primary. In the remaining steps, we introduce mechanisms to recover messages lost due to asynchronous updates, should the backup be promoted to primary.

Step 2: Introducing a Switch

We next introduce a switch for each RSM (primary and backup) in a replica site. Multiple RSMs may share the same switch. Figure 3 illustrates our model for three sites, each with one primary replica, backup, and a switch. In case of switch failure, a new switch can replace the failed switch as we explain in Step 3.

The purpose of the switch is to intercept and forward Paxos messages between its local primary and the switches at the other replica sites. In this step, the switch simply forwards messages, and thus does not introduce any changes. The switch works with the backup to enable *local recovery* without the usual reconfiguration and state transfer required in Paxos.

Step 3: Role of the Switch

In this step of the refinement, the backup is not updated with any state until the primary fails. The switch intercepts Paxos messages to recover the primary should it fail. To recover a primary, the new replica must have the same learned values and acceptor state as the failed primary. This can be determined from the messages that the failed primary has sent to the switch. Note that if the failed acceptor has made a promise or accepted a value, but has not sent any messages reflecting this change, then we do not need to recover this update, as it could not have been observed externally. The switch saves outgoing PROMISE and ACCEPTED messages, and incoming CHOSEN messages. We will eliminate the latter requirement in Step 5.

If the **switch is notified that the primary has failed**, recovery is initiated by isolating the primary. During recovery, this site will not participate in consensus or respond to client requests. The switch begins to store messages and suspends message forwarding to/from the "failed" primary, so that even if it has not actually failed, it cannot interfere with the remaining replicas. The primary will eventually be notified of its removal and terminate. The switch then *promotes* the backup to become primary by sending it the stored messages. The backup applies these messages to its state, in the order received by the switch. Algorithm 1 shows the state maintained by a backup, and how it is updated from the switch. After applying these messages, the backup will have the same state as the failed primary. This new primary notifies the switch that recovery has completed, and the switch resumes forwarding messages. Client requests and Paxos messages from other sites will now be forwarded to the new primary.

Algorithm 1 Backup Update to Become Primary

1: **Initialization:**
2: $rnd[i] \leftarrow \perp$ {Current round number of index i}
3: $vrnd[i] \leftarrow \perp$ {Round number of last vote for index i}
4: $vval[i] \leftarrow \perp$ {Value voted for in last round of index i}
5: $learned[i] \leftarrow \perp$ {Learned state of index i}
6: **on** $\langle$PROMISE$, i, n\rangle$ from switch
7: $rnd[i] \leftarrow n$
8: **on** $\langle$ACCEPTED$, i, n, v\rangle$ from switch
9: $vrnd[i] \leftarrow n$
10: $vval[i] \leftarrow v$
11: **on** $\langle$CHOSEN$, i, v\rangle$ from switch
12: $learned[i] \leftarrow v$

If the **primary is notified that its switch has failed**, a new switch is initialized to replace the failed one. The new switch blocks message forwarding until it receives the necessary Paxos state from the primary. This is important: if the switch tries to recover a backup before having all the primary's state, the backup will not have the appropriate state to maintain safety. For each index for which the primary has learned the chosen value, it sends a CHOSEN message. For all other indices, the primary sends ACCEPTED and/or PROMISE messages.

If the **primary is notified that the backup has failed**, it initializes a new one by sending it all of its chosen values. With this information the new backup can, with the help of the switch, replace the primary should it fail.

If **both switch and primary fail**, they will eventually be replaced through Paxos reconfiguration and wide-area state transfer.

Step 4: Limiting Switch State by Discarding Prefixes

We now reduce the amount of state that the switch needs to store. Upon learning a chosen value, the primary sends that value to the backup. Once it receives chosen values for all contiguous indices through i, the backup notifies the switch. The switch keeps this value i in a variable *MaxForgotten*, and discards its Paxos messages for all indices through i.

If the **switch is notified that the primary has failed**, it first checks that the backup is not in the process of recovering so that local recovery may be performed. It does this by sending *MaxForgotten* to the backup. If the backup has all the chosen values through *MaxForgotten*, it will copy the switch messages as discussed in Step 3 and assume the role of the primary. If the backup is missing chosen values for any of these indices (because it was replaced after a failure but had not completed recovery) then local recovery cannot be performed. In this case, the backup will be promoted to primary through a Paxos reconfiguration and initialized through state transfer from the other replica sites.

After a backup assumes the role of primary—either through local recovery or a Paxos reconfiguration—a new backup will be initialized. Recovery from switch failure is identical to Step 3.

Step 5: Limiting Switch State by Discarding CHOSEN

We can further reduce switch state by removing the requirement that the switch store CHOSEN messages. We expect the backup has most of the chosen values, but that it might be missing some of them. This is because the backup is not kept fully synchronized with the primary, but is instead updated asynchronously. However, the switch will contain the Paxos messages for all of these indices, as they are guaranteed to be greater than *MaxForgotten*. Thus, when being promoted to primary, the backup will acquire the same acceptor state as the failed primary even if it does not have the same learned state for some indices. This new primary will be able to learn the chosen values after recovery completes, e.g., by running Paxos instances for these indices.

Step 6: Limiting Replica State with Snapshots

Now that we have shown how to limit the state stored by the switch, we introduce application snapshots that allow primaries and backups to discard Paxos state. We define the *snapshot* at log index i as the state resulting from executing all RSM commands through index i. Replicas periodically exchange the indices of their snapshots. When a replica learns that a quorum of replicas have a snapshot at index at least i, then it may discard all Paxos state up to i. Replicas will no longer respond to Paxos messages for indices that have been discarded; rather, we use a snapshot state transfer mechanism similar to that of [13], [22].

Let k be the highest index proposed by the system, let i be the primary's snapshot index, and let j be the backup's snapshot index. ZORFU maintains the invariant that for any site, $\mathit{MaxForgotten} \leq j \leq i \leq k$ (illustrated in Figure 3). This property states that the messages stored in the switch plus the snapshot and chosen values stored by the backup will equal the required primary state for local recovery. When a primary initializes a new backup, it copies its snapshot and any chosen values larger than the snapshot index. Should the primary fail before this copy completes, the backup notifies the switch that it does not have the state prior to *MaxForgotten* and will fall back to a heavyweight Paxos reconfiguration and state transfer from remote replicas across the wide-area.

5 DESIGN DETAILS

We now discuss certain design details important for an implementation not covered by the protocol description.

5.1 Failure Detection

ZORFU uses a heartbeat-based failure detector [23] as part of a failure detection service that monitors every machine within a site. If a machine has not sent a heartbeat within a given timeout, the system considers it failed; we do not distinguish between actual failure and slowness for local failure detection. The failure detector will notify interested parties of detected failures, which then respond appropriately. For instance, the failure detector will notify a primary that its associated switch has failed and vice versa. If a failed server resumes sending heartbeats, the failure detector instructs it to deallocate its local state so that it can be reassigned to another RSM. Because the switch forwards all messages to the new primary, a server that was incorrectly declared to have failed cannot interfere with consensus; it will never receive any new consensus messages or RSM operations. Switches believed to have failed are similarly isolated.

The failure detection service is also used by a *controller* component to assist with recovery, as described next.

5.2 Controller

As described in Section 3.1, in ZORFU each site will have its own primary, backup, and switch servers for every RSM. To coordinate the assignment of server replicas, we use a logically centralized controller per site that has knowledge of the site's pool of available machines. The controller maintains two tables. The first maps RSM identifiers to the set of machines hosting the RSM's primary, backup, and switch. The second table is the inverse, mapping from a machine to a list of RSMs hosted on that machine.

A centralized implementation does not bottleneck the system because the controller does not participate in most RSM requests. Rather, the controller is only consulted when an RSM is created or destroyed or when a switch needs to look up an RSM-to-machine mapping. The controller also initiates recovery when machines fail, by creating new replicas or switches. Table 1 summarize

TABLE 1
Summary of recovery actions taken by the controller, switch, and replicas. *Italic indicates remote actions.*

Failed	Primary Actions	Secondary Actions
S	Create new switch; Recover Paxos state from primary.	
B	Create new backup; Recover RSM state from primary.	
P	Promote backup; Recover missing Paxos state from switch.	Create new backup; Recover RSM state from primary.
P, B	Create new primary; *Paxos reconfiguration and wide-area state transfer*.	Create new backup; Recover RSM state from primary.
S, B	Create new switch; Recover Paxos state from primary.	Create new backup; Recover RSM state from primary.
P, S	Create new switch and primary; *Paxos reconfiguration and wide-area state transfer*.	Synchronize backup from primary.
P, B, S	Create new switch and primary; *Paxos reconfiguration and wide-area state transfer*.	Create new backup; Recover RSM state from primary.

the recovery actions taken by the controller, switch, and replicas for different failure scenarios.

We assume the controller itself is replicated for fault tolerance. Should all the controller replicas fail, the mapping from RSMs-to-machines at that site would be lost. In the rare case where a majority of replicas fail and the controller state cannot quickly be recovered from stable storage, we fall back to a relatively heavyweight mechanism. The new controller broadcasts requests for mappings to all the RSM servers in the site and uses the responses to bootstrap the required state information.

5.3 Distributed Switches

A subtle yet important consideration for replicated services deployed across multiple sites is the naming of the replicas in a replica set. In an architecture where every machine makes direct network connections to every other, each replica in the replica set is an explicitly named (i.e. by its IP address and port) server in a particular site. The primary implication of this naming scheme is that information local to one site, i.e., the servers in a given replica set, must be exposed to remote sites. This precludes the possibility of masking failures in one site from remote sites and necessitates a sophisticated global failure detection system to detect server failures and a global machine allocation system to replace failed servers. The wide-area latencies separating sites make accurate, prompt failure detection in the face of transient network errors difficult. Any global server allocation strategy would potentially have to track hundreds of thousands of machines. Both would necessitate a marked increase in wide-area traffic.

The ZORFU switches serve two primary purposes: forwarding messages to the appropriate RSM replicas and storing messages to enable local recovery. The switching service allows replicas in a replica set to be named as sites (specifically, a single IP address and port for each site but not assigned to an actual server) rather than as servers within a site. This design also allows all information pertaining to an individual site, including failure detection and machine allocation, to remain local within that site.

Our switch servers run on commodity hardware and implement a scalable software switching service. Clients and remote sites may send messages to any switch at the destination site. When a switch receives an outgoing message, the message is immediately forwarded to the destination. For incoming messages, the switch checks whether it is responsible for the RSM named in the message, consulting the controller if it does not have an entry in its forwarding table. If so, the switch forwards the message to the appropriate server. Otherwise, it forwards the message to the switch responsible for the RSM, which it learns from the controller.

We note that all switch state, including saved Paxos messages, constitute soft state and need not be persistently stored in the switches for correctness. Should the switch lose any state, it may retrieve it from the authoritative source for the data, either the controller for forwarding entries or appropriate primaries for Paxos state. If the switch does not have the necessary Paxos messages to locally recover from a primary failure, its backup will direct the switch to perform wide-area recovery. We also note that while a server must process both Paxos logic and application-level RSM logic, the switch's processing and storage requirements are limited to inspecting the message type and saving certain messages. Thus, depending on the RSM resource requirements, each switch server may forward messages for many RSMs.

6 DISCUSSION OF ZORFU'S DESIGN CHOICES

We now contrast ZORFU's design choices with several design alternatives and examine their relative costs. We use two of these designs also in our analysis in Section 7.

6.1 Design Alternatives

In the *NoBackup* design, an RSM's state resides on one machine in each site. If a replica in one site fails, a Paxos reconfiguration is executed adding a replica to replace the failed one. After reconfiguration, the new replica will restore its state from replicas at other sites.

The *WARBackup* design reduces recovery time by having several machines host the RSM at each site—one serving as the primary and the others acting as backups. This design is essentially ZORFU's *LocalRecovery* without the switches, so every primary failure entails a Paxos reconfiguration. Since the promoted backup may not have the complete state of the failed primary, some state may still need to be copied from a remote site before the new primary can participate in processing RSM operations. If all of the backups fail before the new

primary can be brought up-to-date, the system will need to fall back to the *NoBackup* design.

A slight variation on ZORFU's *LocalRecovery* would be to use site-internal synchronous replication instead of asynchronous replication, allowing a simpler switch design without the need to maintain Paxos state. This would impose a marginal latency overhead compared to the wide-area latencies involved. However, asynchronous replication makes more efficient use of network and compute resources, and this can help avoid latency increases in highly loaded data center networks.

We also describe a *Paxos-over-Paxos* design which is compatible with recovering locally. As such, this design has recovery performance similar to *LocalRecovery*. The design uses two separate *synchronous replication* protocols, one between sites and another within each site. That is, replicas within a site run Paxos to agree upon messages sent to the other sites that participate in the global Paxos protocol. This is similar to Steward [16] but without Byzantine fault tolerance, and would need $d(2f+1)$ replicas with d sites. *Paxos-over-Paxos* has the benefit of simplicity and symmetry. To support local recovery, the primary of a site can integrate our switch's functionality to isolate local replicas from other sites. In ZORFU, we chose to keep these concerns in the switch, separated from the primary. This keeps our primary simple, and allows our switch to be implemented in a programmable hardware switch, or as a module of a middlebox component together with other front-end functions [24], such as modules for intrusion detection, caching, and load balancing.

6.2 Cost Analysis

The benefits of *LocalRecovery* do come with a cost in terms of using more machine resources. Below we compare the costs for the different design alternatives. Clearly, a plain *NoBackup* approach is the least costly, but as we show in Section 7, lacks in other respects.

To compare the machine resource costs for the various design alternatives we introduce the following notation. Let d denote the number of sites used to host RSMs, let R be the total number of RSMs hosted by each site, and let β be the average number of RSMs that share a machine. Thus, each site needs R/β machines. The number of sites d supported by a particular design is typically tightly coupled with f, the number of replica failures that can be masked. It is clearly possible to configure multiple RSM replicas to reside in the same site, but that would lead to asymmetry in the links between RSM replicas.

For *WARBackup* and *LocalRecovery*, there are exactly *one* primary and b backups in each site. For *LocalRecovery* we also rely on switches. Let γ be the average number of RSMs per switch. Thus, each site needs R/γ switches. Then, consider the resources N needed for *LocalRecovery*:

$$\begin{aligned} N &= d(1+b)R/\beta + dR/\gamma \\ &= 6R/\beta + 3R/\gamma \qquad (b=1,\ d=3) \end{aligned}$$

For a *Paxos-over-Paxos* design, each site needs $2f+1$ replicas, since Paxos needs to collect $f+1$ replies to make progress with synchronous replication. However, with ZORFU's switch and asynchronous replication, we can reduce the number of replicas that need to maintain full RSM state. ZORFU's switch only stores Paxos state, typically a small fraction of the RSM state. For a single RSM, both ZORFU and *Paxos-over-Paxos* would require the same amount of machine resources. However, a single switch can be amortized over multiple RSMs, as expressed with the γ parameter. If $\beta > \gamma$ however, i.e., there are more RSMs per machine than there are RSMs per switch, then ZORFU is disadvantaged. But this is unlikely, since the CPU overhead of the switch is much lower than a full RSM replica. Furthermore, if the machines used for switches are dedicated to this task, these machines could be provisioned with additional network IO resources. We note that d and b relate to fault tolerance, while β and γ relate to the load on machines and switches. The machine resource costs for the different designs are summarized in Table 2.

TABLE 2
Comparing resource costs for the design alternatives.

Design	**Equation** for N	**Cost**[a] N
NoBackup ($f=1$)	$(2f+1)R/\beta$	$3R/\beta$
NoBackup ($f=2$)	$(2f+1)R/\beta$	$5R/\beta$
WARBackup	$d(1+b)R/\beta$	$6R/\beta$
Paxos-over-Paxos	$d(2f+1)R/\beta$	$9R/\beta$
LocalRecovery	$d(1+b)R/\beta + dR/\gamma$	$6R/\beta + 3R/\gamma$

[a] Cost is given in terms of RSMs, R, average number of RSMs per machine, β, and average number of RSMs that share a switch, γ. Where relevant we use $d=3$ sites, $b=1$ backup, and $f=1$.

7 DEPENDABILITY ANALYSIS

Next, we analyze the benefits of ZORFU's *LocalRecovery* design with respect to reducing the window of vulnerability. In *LocalRecovery*, the application and Paxos state stored in primaries is replicated either on a local backup or on a local switch. We evaluate the significance of each of these components by comparing *LocalRecovery* with two other designs—*NoBackup* and *WARBackup*.

Figure 4 shows the advantages of our *LocalRecovery* design by comparing the total time taken to recover, as well as the window of vulnerability for losing state during recovery. We note that contention for the next available index may prolong the reconfiguration window (Section 8.2) for *NoBackup* and *WARBackup*.

7.1 Markov Dependability Models

To compare *LocalRecovery* with the other designs, we constructed a Markov dependability model [25] for each design. The goal of these models is to compare the designs with respect to the expected time until first RSM failure, denoted $\text{MTFF} = E[T_{FF}]$, and the RSM availability. An RSM becomes unavailable when it reaches

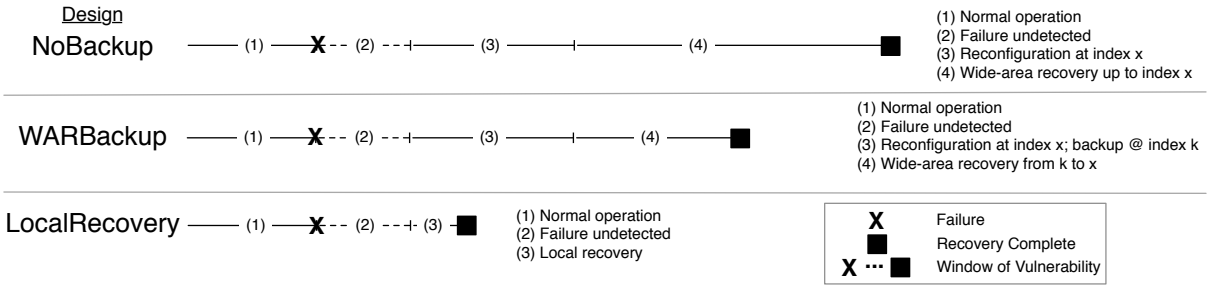
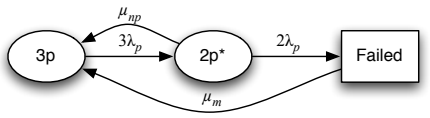
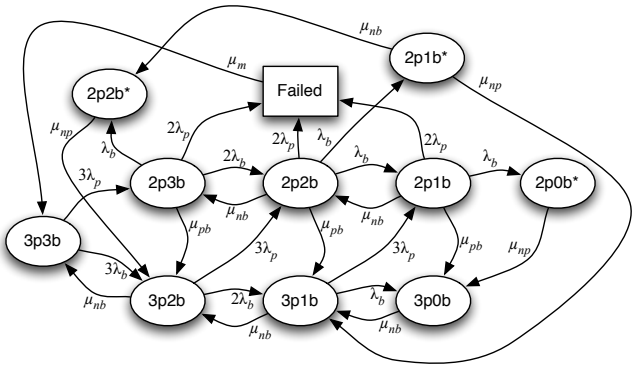


Fig. 4. Comparison across design alternatives of timelines for recovery from failure.

a Markov state wherein processing operations would violate Paxos safety. An RSM is distributed over $2f + 1$ sites, where f is the number of replica failures that can be masked. Moreover, since replicas are distributed across the sites, f also corresponds to the number of site failures that can be masked. We assume a fixed f=1, except for *NoBackup* where we also consider f=2, i.e., replicas at five sites. All application state is stored on one primary per site. For *WARBackup* and *LocalRecovery*, there is also one backup for each RSM per site.



(a) *NoBackup* ($f = 1$)



(b) *WARBackup* ($f = 1$)

Fig. 5. Markov model for two design alternatives.

In modeling the different designs (Figure 5), we examine all the operational states (circles) and the down states (squares). Each up state is identified by $XpYb$, where X and Y represent the number of primaries and backups, respectively. States annotated with a * indicate that wide-area recovery is necessary. In the models, the order in which the replicas fail is not important, and hence we can represent several failure modes with fewer states. In each model different failure rates, denoted λ, lead to different states, weighted by the number of entities that may fail from that state. The restore rates, denoted μ, represent the expected time to recover from a state with failures to another state with fewer failures. To determine the availability of a design we introduce μ_m, denoting manual restore from a down state to the fully operational state. We assume an idealized scenario in which no logical failures or failure of the recovery mechanism occur. We omit a visual representation of the Markov model for *LocalRecovery* because it involves three different entities with quite complicated failure patterns. Instead, we have programmatically generated an accurate model representation based on the behaviors of *LocalRecovery* (partially illustrated in Table 1). The generated model is used in our analysis below, and consists of 512 states. The model could have been simplified significantly, but to obtain numerical solutions this is not necessary. See [25] for additional details on the modeling framework.

TABLE 3
Failure rates and restore times for the different designs. NB=*NoBackup*, WB=*WARBackup*, LR=*LocalRecovery*.

	Parameters		NB	WB	LR
Failure rate (λ)	Primary	λ_p	$(50\text{d})^{-1}$ and $(100\text{d})^{-1}$		
	Backup	λ_b	$(50\text{d})^{-1}$ and $(100\text{d})^{-1}$		
	Switch	λ_s	$(50\text{d})^{-1}$ and $(100\text{d})^{-1}$		
Restoration time (μ^{-1})	Promote backup to primary	μ_{pb}^{-1}	—	10s - 5m	—
	Replace primary (remote copy)	μ_{np}^{-1}	5m - 5h	5h	5h
	Replace backup (local copy)	μ_{nb}^{-1}	—	1h	1h
	Replace switch (local init)	μ_{ns}^{-1}	—	—	10s
	Local recovery	μ_{lr}^{-1}	—	—	3.6s
	Manual restore	μ_m^{-1}	5h	5h	5h

7.2 Analysis Results

We now evaluate the expected time to first failure and unavailability for the Markov models above, under three different operating regimes. The results were obtained using a *Mathematica* package for symbolic and numerical dependability analysis of Markov models [26]. Table 3 describes the notation used in the models and the parameter values used in the analysis.

We conduct our analysis for two values of λ^{-1}=50 and 100 days. The former is representative of machine reliability observed in cloud infrastructures [27], while the latter represents more reliable and costly hardware. As our analysis show, more reliable hardware benefits *NoBackup* more than *LocalRecovery*. However, since we

TABLE 4
Unavailability and MTFF of an RSM for the different design alternatives.

(a) *NoBackup*, $\lambda^{-1}=50$d

$f=1$			$f=2$	
μ_{np}^{-1}	**MTFF**	U	**MTFF**	U
5h	5.6y	$1.0 \cdot 10^{-3}$	136y	$4.2 \cdot 10^{-6}$
1h	27.5y	$2.1 \cdot 10^{-4}$	3310y	$1.7 \cdot 10^{-7}$
5m	329y	$1.7 \cdot 10^{-6}$	473687y	$1.2 \cdot 10^{-9}$

(b) $\lambda^{-1}=100$d

$f=1$		$f=2$	
MTFF	U	**MTFF**	U
22.1y	$2.6 \cdot 10^{-5}$	1070y	$5.3 \cdot 10^{-7}$
110y	$5.2 \cdot 10^{-6}$	26389y	$2.2 \cdot 10^{-8}$
1315y	$4.3 \cdot 10^{-7}$	$3.8 \cdot 10^{6}$y	$1.5 \cdot 10^{-10}$

(c) *WARBackup*, $\lambda^{-1}=50$d

$f=1$		
μ_{pb}^{-1}	**MTFF**	U
5m	329y	$1.7 \cdot 10^{-6}$
1m	1644y	$3.5 \cdot 10^{-7}$
10s	9863y	$5.8 \cdot 10^{-8}$

(d) $\lambda^{-1}=100$d

$f=1$	
MTFF	U
1315y	$4.3 \cdot 10^{-7}$
6576y	$8.7 \cdot 10^{-8}$
39452y	$1.4 \cdot 10^{-8}$

(e) *LocalRecovery*, $\lambda^{-1}=50$d

P=Primary, B=Backup, S=Switch, _=Down.	$f=1$	
Up states for one Data Center	**MTFF**	U
$\mathcal{P}$: $\{P\_\_, P\_S, PB\_, PBS\}$	2961y	$1.5 \cdot 10^{-10}$
$\mathcal{BS}$: $\{\_BS, P\_\_, P\_S, PB\_, PBS\}$	$1.97 \cdot 10^{6}$y	$7.5 \cdot 10^{-11}$

(f) $\lambda^{-1}=100$d

$f=1$	
MTFF	U
21291y	$1.5 \cdot 10^{-11}$
$3.15 \cdot 10^{7}$y	$4.7 \cdot 10^{-12}$

specifically target a cloud environment, our analysis focus on the most representative failure rate. The manual restore rate is fixed to μ_m^{-1}=5h for all designs. A higher manual restore rate would negatively impact *NoBackup*, while *LocalRecovery* see almost no negative impact, even for very long restore rates. For the remaining parameters, we refer to Table 3, and argue that the various restoration times are representative. For example in *WARBackup* and *LocalRecovery*, we fix the time to replace a primary across the wide-area to μ_{np}^{-1}=5h, and to replace a backup from a local primary to μ_{nb}^{-1}=1h. These numbers could represent an RSM with a 2GB state size, and link capacities reserved for recovery of 1Mbps for WAN and 5Mbps for LAN. For *NoBackup*, μ_{np}^{-1} varies from 5 hours to 5 minutes, representing different RSM state sizes to be recovered over the wide-area. For *WARBackup*, we employ a μ_{pb}^{-1} that varies from 5 minutes to 10 seconds, representing the time to promote a backup to become primary. Note that, although our experimental evaluation (Section 8) shows that *LocalRecovery* can be completed in a few milliseconds, we use μ_{lr}^{-1}=3.6s to account for failure detection time. Table 4 gives the results of our analysis.

LocalRecovery (Table 4(e)) is evaluated for two interpretations of availability. In the first model ($\mathcal{P}$), a site is considered to be *up* iff its primary is up. This is technically accurate, because only the primary can respond to requests. However, the backup and switch together have the same state as the failed primary, so we also evaluate a model ($\mathcal{BS}$) treating a site as up if its backup and switch are up. This is reasonable since *LocalRecovery* will restore the primary in at most a few seconds; clients would experience a slight delay, typically accepted by users of Internet services.

As the results show, *NoBackup* ($f=1$) has what seems to be "good enough" MTFF numbers for a practical deployment. But recall that these numbers represent idealized failure assumptions, and thus it is desirable to further increase the MTFF. This is accomplished by reducing the time to recover from a failure, which is a distinguishing characteristic between the designs. Thus as expected, *WARBackup* and *LocalRecovery* significantly increase MTFF. Note the difference in MTFF between the $\mathcal{P}$ and $\mathcal{BS}$ models. This is due to the $\mathcal{P}$ model accounting for a small delay, while a backup and switch is recovering a primary, as a down state. On the other hand, there is only a relatively minor difference in unavailability for the two models.

NoBackup ($f=1$) only provides 2-3 nines of availability for recovery times of 1-5 hours, which is much less than the target of five nines desired by many services. *NoBackup* ($f=2$) does provide this availability, but at the expense of potentially longer latencies for every request, significantly increased message complexity and bandwidth requirements, and higher processing requirements, as the resources needed by Paxos grows linearly with f. Note that both *LocalRecovery* models yield better availability than *NoBackup* ($f=2$).

Beyond the benefits of fewer messages and processing for every request relative to *NoBackup* ($f=2$), *WARBackup* and *LocalRecovery* greatly reduce the bandwidth needed between sites for most types of recovery. If services already use most of their bandwidth running consensus for normal operations, then having to periodically transfer large amounts of state between sites for recovery may cause contention for bandwidth and disrupt system throughput. Services may avoid a drop in operational throughput by throttling recovery traffic, but this will further increase the window of vulnerability, prolonging the time to recover.

WARBackup and *LocalRecovery* also minimize disruption experienced by clients. Services try to direct clients to the site that provides the best performance, significantly reducing the latency for client requests. While the primary at that site is recovering, client requests must be redirected to a different replica, increasing the network delay for those requests. Because *LocalRecovery* can recover very quickly, on the order of milliseconds, a small amount of buffering in the switches will allow services to completely mask a primary failure from the client, providing significantly improved responsiveness and performance.

Other factors not accounted for in our analysis, such as correlated failures, logical failures, power outages, and security threats will likely lead to lower availability than what is shown in Table 4. However, what our analysis does show is that by using *LocalRecovery*, we can shift the "bottleneck" for RSM availability to these other factors.

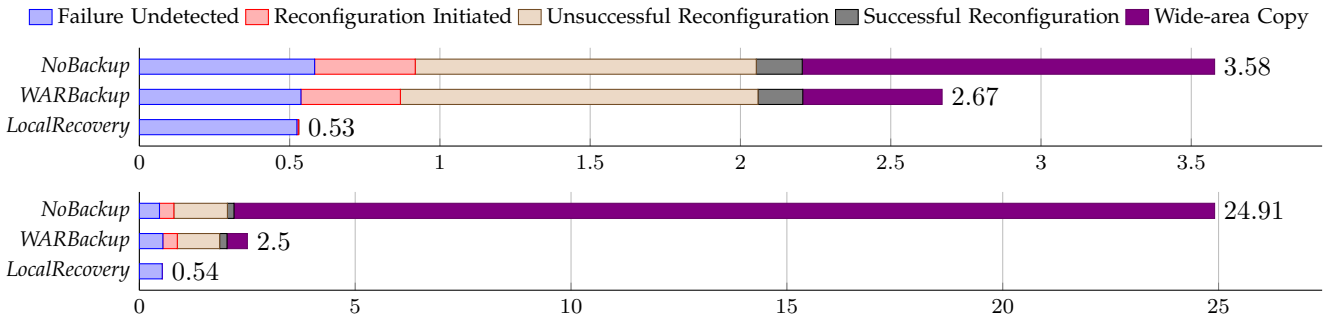


Fig. 6. Recovery time for a single RSM (seconds). Local recovery took 4ms. Top: 1MB object. Bottom: 100MB object.

8 Evaluation

Our experimental evaluation aim to demonstrate that ZORFU significantly *reduces the recover time* relative to other techniques and that it *increases the availability of all RSMs*. We implemented ZORFU and a replicated object store using the C++-based MACE [28] toolkit for building distributed systems. The object store provides highly available storage for arbitrary blob objects on top of ZORFU. Clients use it through the following API:

Create(obj)	create a new object with name *obj*
Read(obj, offset, size)	read *size* bytes from *offset*
Update(obj, offset, buffer)	write the *buffer* to *offset*
Truncate(obj, size)	truncate the object to *size* bytes

8.1 Experimental Setup

We use ModelNet [10] to emulate the environment necessary to perform our evaluation. In our setup, three data centers (A, B, and C) are pairwise-connected by 100Mbps links. To emulate a realistic geographic separation between data centers, we assign 10ms of latency between A-B, 20ms between A-C, and 50ms between B-C. Each data center consists of a switch and four servers each running on dedicated physical machines. For experiments using backups, we use one backup per RSM for each data center. Clients are connected to data centers over an 8Mbps, 2ms link, and are multiplexed onto three machines, up to four clients per machine. We configure ModelNet so that traffic local to a data center uses the physical GbE switch rather than going through ModelNet. Each machine has a 2.13GHz quad-core Intel Xeon CPU, 4GB RAM, and 1Gbps Ethernet.

To evaluate the three different recovery designs, we use the same system implementation with varying configurations for the number of backup servers (including none). Because we use the same system for all designs, our evaluation includes a version of *NoBackup* and *WARBackup* using switches (that do not maintain Paxos state), even though these designs could also operate without a switch. However, the overhead of the switch in these experiments is negligible.

8.2 Time to Recover

We measure the breakdown of recovery time for the three designs *NoBackup*, *WARBackup*, and *LocalRecovery*, for two different object sizes of 1MB and 100MB. For each of these six experiments, a single client connects to data center B and begins issuing 4KB pipelined updates with up to 64 outstanding, up to a total of 15000 updates. At approximately 10 seconds into the run, we kill the server acting as the primary replica at data center B. Figure 6 shows the resulting recovery timeline for the six experiments.

These graphs illustrate how quickly ZORFU recovers relative to the other two design choices. Because no Paxos reconfiguration or wide-area data transfer is required, the switch can promote a backup to replace the failed replica in just a few milliseconds, a time too small to be pictured on the graphs. For both 1MB and 100MB objects, the recovery completes in 4ms; because local recovery only copies the most recent Paxos messages, it is independent of object size. Indeed, local recovery is dominated by the failure detection time, or approximately 500ms for our failure detector.

Figure 6 plot timeline graphs for each experiment, split into different time shares for each phase of recovery. Local recovery completes in just two phases: 1) time to detect failure, and 2) copy Paxos state (too small to plot on a second time scale). For the other two designs, we break wide-area recovery time into five phases:

1) Failure Detection Approximately 500ms.

2) Reconfiguration Initiation This phase accounts for the switch querying the remote replicas for their highest chosen index for the relevant RSM. We note that the *NoBackup* results reflect expected recovery times for ZORFU when it experiences simultaneous failures of the primary and all backups, and that *WARBackup* reflects the scenario when the switch has lost its Paxos state, preventing local recovery.

3) Unsuccessful Reconfiguration Attempts A reconfiguration proposal may not be chosen at the requested index because of contention with other proposals. This contention is still possible even when using a leader that limits its number of outstanding proposals to some fixed amount α. One of the other replicas may be lagging, then believe that the leader has failed and attempt to initiate reconfiguration. Because the replica was lagging, it does not know the last index at which the leader issued a proposal, so it cannot compute the correct next

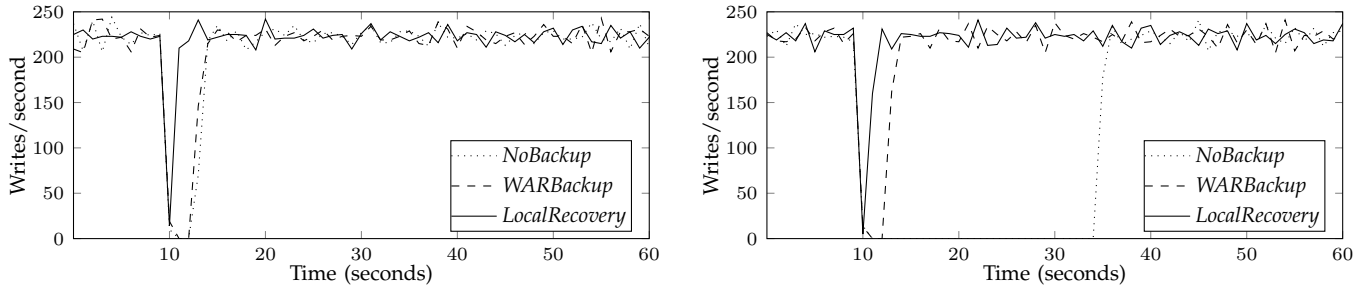


Fig. 7. Client throughput during failure of its local replica. Left: 1MB object. Right: 100MB object.

uncontested index, even though it knows that should be α after the last index.

4) Successful Reconfiguration This represents the time for three-phase Paxos to choose the Paxos configuration for an uncontested index. Once completed, the replacement replica has been added to the configuration.

5) Wide-area Data Copy As soon as the reconfiguration completes, the new replica begins transferring state from other wide-area replicas to bring itself up-to-date. Even without *LocalRecovery*, having a local backup vastly reduces the amount of data that must be copied—and the window of vulnerability—for larger objects.

Figure 7 shows the effect of the replica's failure on the client's throughput. The failure of the client's local replica represents the worst possible failure condition from that client's perspective, as some of the client's outstanding requests are lost when the failure occurs. Throughput is sampled once per second. The effect of the failure is much more pronounced for a 100MB object, as the service is unavailable from the client's perspective while the wide-area data transfer portion of recovery is taking place. We note that for any of the recovery scenarios, the client could have maintained throughput even during recovery by sending its requests to one of the other replicas.

9 Related Work

We build on a large body work, in particular protocols for building replicated state machines for fault tolerance [7], [11], virtual synchrony for ordering requests [29], and replicated logs [12]. Recent efforts have focused on making Paxos perform well in real-world usage scenarios, and [30] provides a thorough specification and evaluation of Paxos. Researchers at Google [14] have provided insight into how a real-world Paxos implementation handles different classes of failures not explored in the original protocol. While Paxos can be made very efficient in LAN environments, latency becomes an issue in wide-area settings. Mencius [31] delivers high performance for Paxos in a wide-area setting by partitioning the leader role for proposals among replicas in a round-robin fashion. However, the focus of Mencius is on wide-area performance, and it provides the same availability as classic Paxos [7]; however, their approach could be combined with ZORFU. EPaxos [32] is another Paxos variant designed for high performance in the wide-area. EPaxos takes advantage of the insight that for many workloads, it is not necessary to enforce a consistent ordering for commands that do not interfere with each other. They avoid the bottleneck of a global leader, and instead use a separate leader for each command together with a fast-path quorum to detect conflicting commands. Without conflicts, commands can be committed quickly. Steward [16] addresses the problem of providing Byzantine fault tolerance in a wide-area setting consisting of a number of wide-area sites containing several server replicas. Steward also takes a hierarchical approach in which a BFT protocol is used within each site, and a Paxos-like protocol is used across wide-area sites. Steward requires $3f + 1$ replicas at *each* site to tolerate f failures at *any* site. More generally, given the large number of RSMs and machines in our target environment, we believe the overhead for variations of hierarchical replication to be prohibitively high. As we discussed in Section 6, a scaled-down version of Steward could use $2f + 1$ replicas in each site, but with our switch we can more easily share machine resources.

Previous systems have addressed the challenge of achieving high availability over wide-area networks by relaxing consistency guarantees [3]. ZORFU guarantees strong consistency, which is important for a range of cloud computing services. Windows Azure Storage (WAS) [5] is a highly-available cloud storage service claiming to provide all three properties of the CAP theorem [33] by using an underlying append-only storage together with mechanisms for sealing append data only when enough replicas has seen the update. In WAS, synchronous replication is used within a data center, while asynchronous replication is used across the wide-area. This removes the wide-area latency from the critical-path, making the trade-off that a data center failure may lead to data loss. In ZORFU we took the opposite design choice, requiring a majority of replica sites commit to a request. The wide-area recovery performance of WAS is not evident from [5]. Similar to ZORFU, Megastore from Google [4] is also based on a synchronous wide-area Paxos, but does not asynchronously update a local backup in each data center. Instead they provide a spe-

cial coordinator service to facilitate fast reads. BigTable is used for storage. Like ZORFU, SMART [13] attempts to decrease the amount of state transferred during recovery. It does this by letting new replicas use part of all the other co-located replicas' shared state. Similarly, [34] uses linear programing to compute a schedule for faster state transfer from the remote replicas. ZORFU could leverage such techniques, but our local recovery mechanism will render these wide-area copies rare.

Our main contribution is on reducing the window of vulnerability during failure recovery. Recovery-oriented computing [35] shares our goal of increased availability by decreasing recovery time from inevitable failures, rather than by trying to prevent them. Recovery has also been studied extensively in the context of group communication and CORBA. FT CORBA [36] standardizes mechanisms such as a generic factory, a logically centralized replication manager (our controller), and a fault monitoring architecture. Several frameworks [37], [38], [39], [40] that draw inspiration from the standard have been implemented. These frameworks use a centralized controller like ZORFU, except [40] where the controller is distributed among the replicas. Although [39], [40] also target wide-area replication, none of these frameworks provide a local recovery mechanism.

10 CONCLUSION

We consider a practical problem in the deployment of a distributed service with strong consistency: how does one recover from server failures? We show that the solution to this problem has a significant impact on the availability of services, such as a data store. Our solution, ZORFU, provides a scalable implementation of *local recovery* that significantly reduces the window of vulnerability during failure recovery. We show that this results in much lower probability of system failure than to simply increase the number of replicas. We derive ZORFU from Paxos and evaluate its availability both analytically and experimentally. Our results show that the recovery time for individual failures is reduced by more than an order of magnitude over conventional recovery techniques, thus providing high availability. In summary, ZORFU provides cloud-based services with the simplicity of strong consistency semantics and high availability, while still maintaining high performance.

ACKNOWLEDGEMENTS

We thank Harsha V. Madhyastha, Meg Walraed-Sullivan, Roman Vitenberg, and Leander Jehl for feedback on early drafts of the paper. Prof. Meling was partially supported by the Tidal News project under grant no. 201406 from the Research Council of Norway.

REFERENCES

[1] K. P. Gummadi, H. V. Madhyastha, S. D. Gribble, H. M. Levy, and D. Wetherall, "Improving the Reliability of Internet Paths with One-hop Source Routing," in *OSDI*, 2004.

[2] F. B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Comput. Surv.*, 1990.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-Value Store," in *SOSP*, 2007.

[4] J. Baker, C. Bond, J. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *CIDR*, 2011.

[5] B. Calder *et al.*, "Windows azure storage: a highly available cloud storage service with strong consistency," in *SOSP*, 2011.

[6] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally-distributed database," in *OSDI*, 2012.

[7] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998. [Online]. Available: http://doi.acm.org/10.1145/279227.279229

[8] ——, "Paxos Made Simple," *ACM SIGACT News*, vol. 32, no. 4, 2001.

[9] D. Taft, "Amazon's Head Start in the Cloud Pays Off," http://www.eweek.com/c/a/Cloud-Computing/Amazons-Head-Start-in-the-Cloud-Pays-Off-584083/, 2009.

[10] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker, "Scalability and accuracy in a large-scale network emulator," in *OSDI*, 2002.

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

[12] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *OSDI*, 1999.

[13] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell, "The SMART way to migrate replicated stateful services," in *EuroSys*, 2006.

[14] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *PODC*, 2007.

[15] K. Church, A. Greenberg, and J. Hamilton, "On delivering embarrassingly distributed cloud services," in *HotNets*, 2008.

[16] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage, "Steward: Scaling byzantine fault-tolerant replication to wide area networks," *IEEE Trans. Dependable Sec. Comput.*, vol. 7, no. 1, pp. 80–93, 2010.

[17] Y. Amir, B. A. Coan, J. Kirsch, and J. Lane, "Customizable fault tolerance for wide-area replication," in *SRDS*, 2007.

[18] M. J. Csorba, H. Meling, and P. E. Heegaard, "A bio-inspired method for distributed deployment of services," *New Generation Computing*, vol. 29, no. 2, pp. 185–222, Jan 2011.

[19] G. Jung, K. R. Joshi, M. A. Hiltunen, R. D. Schlichting, and C. Pu, "Performance and availability aware regeneration for cloud based multitier applications," in *DSN*, 2010.

[20] L. Lamport, D. Malkhi, and L. Zhou, "Reconfiguring a state machine," *SIGACT News*, vol. 41, pp. 63–73, March 2010.

[21] J. W. Anderson, "Consistent cloud computing storage as the basis for distributed applications," Ph.D. dissertation, UCSD, 2011.

[22] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative Byzantine Fault Tolerance," in *SOSP*, 2007.

[23] K. C. W. So and E. G. Sirer, "Latency and bandwidth-minimizing failure detectors," in *EuroSys*, 2007.

[24] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat, "xOMB: Extensible Open Middleboxes with Commodity Servers," in *ANCS*, 2012.

[25] B. E. Helvik, *Dependable Computing Systems and Communication Networks – Design and Evaluation*. Tapir academic publisher, 2009.

[26] ——, "Dependability analysis with reliability block diagrams," library.wolfram.com/infocenter/MathSource/7371, 2009.

[27] J. Dean, "Designs, lessons and advice from building large distributed systems," in *LADIS*, 2009.

[28] C. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. Vahdat, "Mace: Language Support for Building Distributed Systems," in *PLDI*, 2007.

[29] K. P. Birman, "The Process Group Approach to Reliable Distributed Computing," *CACM*, vol. 36, no. 12, 1993.
[30] J. Kirsch and Y. Amir, "Paxos for System Builders," in *LADIS*, 2008.
[31] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: Building Efficient Replicated State Machine for WANs," in *OSDI*, 2008.
[32] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *SOSP*, 2013.
[33] E. A. Brewer, "Towards Robust Distributed Systems (Invited Talk)," in *PODC*, 2000.
[34] N. Raghavan and R. Vitenberg, "Balancing communication load of state transfer in replicated systems," in *SRDS*, 2011.
[35] G. Candea, A. B. Brown, A. Fox, and D. Patterson, "Recovery-oriented computing: Building multitier dependability," *Computer*, vol. 37, pp. 60–67, 2004.
[36] OMG, "Fault Tolerant CORBA Specification," OMG Document ptc/00-04-04, Apr. 2000.
[37] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith, "Eternal – a Component-Based Framework for Transparent Fault-Tolerant CORBA," *Softw., Pract. Exper.*, vol. 32, no. 8, pp. 771–788, 2002.
[38] Y. Ren, D. E. Bakken, T. Courtney, M. Cukier, D. A. Karr, P. Rubel, C. Sabnis, W. H. Sanders, R. E. Schantz, and M. Seri, "AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects," *IEEE Trans. Comput.*, vol. 52, no. 1, pp. 31–50, Jan. 2003.
[39] H. Meling, A. Montresor, B. E. Helvik, and O. Babaoglu, "Jgroup/ARM: a distributed object group platform with autonomous replication management," *Softw., Pract. Exper.*, vol. 38, no. 9, pp. 885–923, July 2008.
[40] H. Meling and J. L. Gilje, "A Distributed Approach to Autonomous Fault Treatment in Spread," in *EDCC*, 2008.

James W. Anderson completed his Ph.D. in the Systems and Networking Group at UC San Diego in 2011. He now works on research and development for a small technology firm in NYC.

Hein Meling is Professor at the Department of Electrical Engineering and Computer Science at the University of Stavanger, Norway, where he runs a small research group developing systems and protocols to improve the robustness of network services. He was co-principle investigator on the IS-home and Tidal News projects funded by the Research Council of Norway, focusing on fault tolerance and recovery in distributed event-based systems and geo-replicated cloud computing infrastructures.

Meling's research interests span dependable and secure computer systems, including distributed systems and data center networks. He received a Ph.D. in 2006 from the Norwegian University of Science and Technology. He spent his sabbatical in 2010/11 at University of California, San Diego, working on Paxos-based protocols for robust and secure network services.

Alexander Rasmussen is a Senior Software Engineer at Trifacta, Inc. Prior to joining Trifacta, Dr. Rasmussen earned his Ph.D. at University of California San Diego, where his research focused on efficient large-scale data processing.

Amin Vahdat is a Fellow and Technical Lead for Networking at Google. He has contributed to Google's data center, wide area, edge/CDN, and cloud networking infrastructure, with a particular focus on driving vertical integration across large-scale compute, networking, and storage. He is an Adjunct Faculty member in the Department of Computer Science and Engineering at the University of California San Diego. He was a Professor in the Computer Science and Engineering Department from 2003-2013.

Vahdat's research focuses broadly on computer systems, including distributed systems, networks, and operating systems. He received a Ph.D. in Computer Science from UC Berkeley under the supervision of Thomas Anderson after spending the last year and a half as a Research Associate at the University of Washington. Vahdat is an ACM Fellow and a past recipient of the the NSF CAREER award, the Alfred P. Sloan Fellowship, and the Duke University David and Janet Vaughn Teaching Award.

Keith Marzullo is the Director of the Federal Networking and Information Technology Research and Development (NITRD) National Coordination Office (NCO). He performed this research as a professor at the University of California, San Diegos Computer Science and Engineering Department, where he served as the Department Chair from 2006-2010. Dr. Marzullo received his Ph.D. in Electrical Engineering from Stanford University, where he developed the Xerox Research Internet Clock Synchronization protocol, one of the first practical fault-tolerant protocols for keeping widely-distributed clocks synchronized with each other.