Fault-tolerant replication management in large-scale distributed storage systems

Richard Golding
Storage Systems Program, Hewlett-Packard Labs
golding@hpl.hp.com

Elizabeth Borowsky
Computer Science Dept., Boston College
borowsky@cs.bc.edu

Abstract

Failures of all forms happen: from losing single network packets to site-wide disasters. Since businesses rely heavily on their data, it is imperative that failures require minimal time and effort to repair and that the service interruption during the failure or repair period should be as short as possible. To this end, the ideal system should repair itself, relying on humans only when absolutely necessary in the repair process. This paper describes one component of a self-healing storage system: the component that allows for automatic recovery of access to data when the power comes back on after a large-scale outage. Our failure recovery protocol is part of a suite of modular protocols that make up the Palladio distributed storage system. This protocol guarantees that service will be repaired quickly and automatically when enough failures are repaired.

1 Introduction

Many organizations need their computing systems to survive disasters that disable or destroy entire sites, such as power outages, earthquakes, and storms. This leads to providing hosts and storage devices at multiple sites and replicating data to two or more of the sites. Such systems must keep data available as much as possible without compromising its correctness; and when data cannot be available for use, it must be maintained until enough of the system recovers that access can be restored. This is in addition to "ordinary" failure tolerance, which handles the failure of individual hardware components—in our case, storage devices.

Recovering from site failures require different solutions than recovery from single device failures, especially when systems grow large. Site-wide recovery must proceed swiftly and scalably. The primary differences are that many or all components of the system can fail or recover at the same time, and the number of parallel recovery efforts that may be ongoing, for example after a site recovers power, may require that individual recovery steps be frugal in their resource usage to avoid overloading the system. *All* transient state can be lost in such failures, requiring careful attention to ensure that any information needed for recovery is kept on persistent storage.

The Palladio solution for detecting, handling, and recovering from both small- and large-scale failures in a distributed storage system is a synthesis of work done on fault-tolerant distributed systems, particularly consensus for consistency and replication for redundancy and fault tolerance. Our approach was inspired by standard replica control protocols, notably that of El Abaddi and Toueg [10]. Unlike previous work, we implement the storage system using four modular protocols, so that any protocol that meets certain behavioral requirements can be used. In this paper, we focus on the *layout control protocol,* named by analogy with replica control protocols, that can correctly recover a system from site failure.

The Palladio distributed storage system provides virtualized data storage services to applications running on a cluster of hosts, connected using a communication fabric. The system provides the applications with a set of *virtual stores,* which are structured as a logical array of bytes into which applications can write and read data, very much like the high-level interface to a SCSI disk. A write operation is *tri-state atomic*, in that it either completes in its entirety, has no effect, or results in the data range being marked damaged (in the event of rare permanent failures). Reads and writes to a store are serialized. This model is simpler than that of a database—atomicity and serialization are at the granularity of individual I/O operations, rather than multi-operation transactions.

Virtual stores are implemented by allocating space on one or more storage devices. The store's *layout* maps each byte in its address space to an address on one or more devices. The layout can use striping for performance and replication for reliability. Unlike some other distributed storage systems (e.g. Petal [16]), Palladio potentially uses a different layout strategy for each virtual store to obtain near-optimal resource utilization [4, 5].

Palladio allows dynamic changes to a virtual store's layout in order to support fault tolerance and good resource utilization. For example, when a storage device holding one replica fails, the system maintains redundancy by creating another replica using spare space on other devices—akin to using a hot spare disk in a disk array, but spread over an entire system. Similarly, if the throughput needs for a particular store increase, the system may choose to stripe the store's data across more storage devices.

Computing environments that use Palladio are expected to be quite large, storing as much as several petabytes of data on tens of thousands of disk arrays. Protocols that scale well to these sizes must show sublinear growth in execution time, messages, and space as the system size increases. One implication of this constraint is that protocols cannot simply search the network to find other nodes or data. For example, the failure recovery mechanisms cannot search all devices in order to find the replicas that make up a particular store. Likewise, a recovery protocol that involves a broadcast to all nodes can overload the network when a site is recovering from a power failure. This difference in scale has dictated a somewhat different approach for Palladio than has been used in many system: that storage devices take an active role in the recovery of the stores they are part of.

Palladio is implemented as shown in Figure 1. Each host includes a device driver that maps application read and write requests into low-level read and write operations and sends them to the appropriate storage devices. A small set of managers keeps track of the virtual stores in the system, coordinating changes to their layout and handling recovery from failure. Of course, a store's manager can fail, in which case the store's management function is quickly regenerated onto another manager.

The rest of the paper is structured as follows. In Section 2, we outline the behaviors of the protocols used to implement the system. We then present the system assumptions we used, followed by the layout control protocol used in Palladio. We outline proofs for its essential correctness properties, then discuss extensions and related work.

2 Modular protocols

The overall purpose of the Palladio system is to provide robust read and write access to data in virtual stores. To achieve this, it must provide atomic and serialized read and write access, detect and recover from failure, and accommodate layout changes.

Palladio divides the implementation into four protocols, as shown in Figure 1. In this section we discuss the behaviors these protocols must provide each other.

The protocols run between three sets of entities: *hosts,*

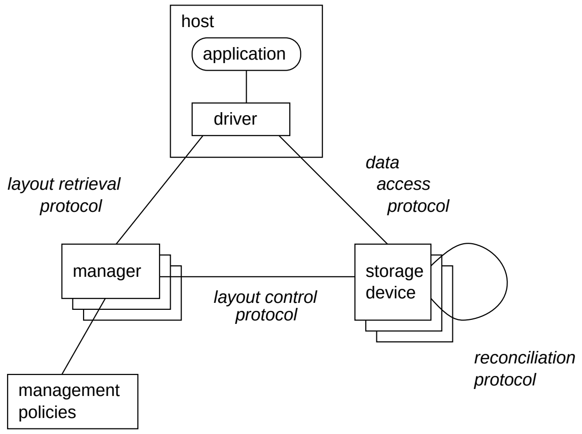

Figure 1. Palladio implementation structure. The manager, storage device, and hosts execute the four protocols shown to provide applications with data storage services.

on which applications run and which include a device driver that translates application I/O requests into per-device operations; *storage devices,* which actually store the data; and *managers,* which maintain the layout information. In addition, there is a *management policy* component that determines what resources should be used to implement each virtual store.

The *access protocol* allows hosts to read and write data on a storage device as long as there are no failures or layout changes for the virtual store. It must provide serialized, atomic writes that can span multiple devices. When there has been a failure or layout change, the access protocol must block until the appropriate recovery actions have occurred to ensure that the operations remain atomic and serializable.

The *layout retrieval protocol* allows hosts to obtain the current layout of a virtual store—the mapping from the virtual store's address space onto the devices that store parts of it—from the manager that keeps track of keeps track of the store's layout. The host will typically cache the layout information so that it does not have to access the manager on every operation. The protocol must provide a way for a processor to locate the store's active manager without running into scalability problems.

The *reconciliation protocol* runs between pairs of devices to bring them back to consistency after a failure. The protocol must ensure that writes that are performed using the access protocol are atomic, by either rolling back or committing unfinished writes. Note that the simple semantics provided in a storage system—all writes are overwrites, and atomicity is at the level of single I/O requests—makes this procedure straightforward. Reconciliation can typically be performed repeatedly with no ill effect.

Finally, the *layout control protocol* runs between managers and devices. It maintains consensus about the layout and failure status of the devices, and in doing so coordinates the other three protocols. When a store's layout needs to be changed, the store's manager must initiate a layout update transaction with the storage devices holding parts of the store; when the transaction commits, they must have reached consensus on the new layout. As part of the transaction, the devices block the access protocol until hosts have refreshed their layout information using the layout retrieval protocol. The layout control protocol must also detect when devices have failed and recovered. When a device recovers, the layout control protocol must cause the reconciliation protocol to resolve any differences among replicas, then inform the devices that they can re-enable the access protocol.

We have developed one such set of protocols as part of developing the Palladio system. The details on the access and reconciliation protocols can be found in [1]. The layout retrieval protocol is simple; it uses a tree-based distributed search structure to allow a host to find the manager it should communicate with, after which it simply requests a new copy of the layout. The layout control protocol is the subject of this paper.

3 System assumptions

All *processors*—managers, devices, and hosts—have a bounded difference in processing rate, so that all local processing steps in a protocol take at most a bounded, finite time δ_{step}. Each processor has a clock that is loosely synchronized with real time; that is, there is a bound δ_{clock} on the difference between the local clock and real time.

Each processor is named by a unique identifier, and there is a total precedence ordering on processor identifiers. In addition, storage devices can be *allocated* and *deallocated.* Each device includes an *incarnation number* that is generated at each allocation so that the combination of device identifier and incarnation number is unique over all time.

Processors fail by crashing, and some failures are permanent. A transient failure causes the process to lose all its non-persistent state and to enter a newly-rebooted state.

The *network* reliably delivers messages in a bounded time δ_{msg} in the absence of partitions. The network does not corrupt or replay messages, nor spontaneously generate them. Messages are delivered in FIFO order between pairs of processors. In practice this can be approximated using a transport protocol that masks out occasional packet loss. The network can partition processors into mutually communicating subsets. During a partitioning, any two nodes can communicate if and only if they are in the same partition.

The network provides means of locating all reachable nodes of a particular kind within some bounded time δ_{lookup}. The results of this search may omit some, but not all, of the appropriate nodes and may include only a bounded number of inaccurate nodes. This could be accomplished through a name service implemented as part of the network infrastructure.

We assume a fail-silent model of *failures,* where neither the network nor nodes exhibit performance failures or generate messages outside the protocol. We model all failures locally as a restart of a node to a known state, with all non-persistent data lost; all failures of remote nodes appear indistinguishable from network partitions. There is no bound on the time required to recover from a failure, since some failures will be permanent. Permanent network partitions, in particular, can prevent the layout control protocol from ever recovering. As a result we show that recovery occurs only when recovery makes sufficient resources become available and remain available for at least a bounded *stability period* $\delta_{recovery}$ (Section 5).

4 Layout control protocol

This section presents a summary of the layout control protocol. For full details see [12].

4.1 Epochs and consensus

The layout control protocol tries to maintain agreement between a store's manager and the storage devices that hold the store. There are two things to be agreed upon: the layout of data onto storage devices, and the identity of the store's *active manager*.

While there will usually be many manager nodes on the network, at any given moment at most one manager node will be the *active manager* for a particular store. The identity of the active manager can change over time, either because of failures or to balance load across the system.

The storage devices and manager go through *epochs* of agreement, during which the layout and manager are fixed. They perform a transactional *epoch transition* to move from one layout or manager to another, using a distributed atomic commitment protocol initiated by the manager to ensure consensus. Efficient protocols exist to accomplish this in the face of failures (e.g. [15]), which complete in a bounded time $O(\delta_{msg})$ in the absence of failures. Epoch transitions happen as a result of layout changes, migration of the active manager, or as the final step in recovery.

Each epoch is numbered, and the *epoch number* is increased by one on each epoch transition. The epoch number thus is a shorthand reference for the version of the layout information, and is used as such in to signal when hosts need to refresh their cached copy of the layout.

During an epoch, each non-failed device obtains a time-limited *lease* [3, 14] from its active manager, and periodically asks for the lease to be renewed. As long as the device has a lease, it believes that the manager is functioning. The manager believes that a device is functioning as long as the device has an unexpired lease. When a device does not renew its lease in time, the manager believes the device has failed and maintains this belief until the end of the epoch. Short leases make for fast detection of failure, but impose more messaging overhead than longer leases.

The leases implement a kind of *failure suspector.* The atomic commitment and layout control protocols use this failure suspector to detect when processes have (probably) failed. Consensus is solvable in asynchronous systems using an unreliable failure suspector [7, 8], which will suspect any failed process of having failed but which may suspect some (but not all) correct processes as having failed.

4.2 Interface to other protocols

The correct functioning of the system depends on the data access, data reconciliation, and layout control protocols working together. We discuss a few of the essential properties here.

The data access protocol [1] provides serialized and atomic data read and write operations that span multiple devices. The protocol performs efficiently in the normal case where there are no failures.

When there is a failure, or when some device loses its lease, any (multi-device) write operation that includes that device will block. The non-failed devices will then begin queuing subsequent read and write requests. This preserves the state of the data until the failed device is reintegrated into the system. Reintegration includes reconciling the replicas to restore consistency among the the devices, which completes or rolls back any unfinished write operations on reachable devices.

The layout control protocol controls the behavior of the access protocol by manipulating the device's lease: a device may perform read and write operations only while its lease is valid. The layout control protocol also handles reintegrating devices that have returned to service, by running the reconciliation protocol, performing an epoch transition, and only then beginning to issue leases to the device again.

The layout control protocol and access protocol together ensure that the host's cached copy of layout information is consistent with the manager's version. All the protocols use the epoch number as a version number for the layout information, and the host maintains this version number with its cached layout copy. The layout control protocol increases the epoch number at each change in the layout, and ensures that the manager and all non-failed devices agree on that number. On each I/O, the access protocol checks that the host's version number matches the device's version; if not, then the host refreshes its copy using the layout retrieval protocol.

variable	meaning	persistence
e	epoch number	transient
$\langle V, D_V, \Lambda\rangle$	layout	transient
$\{d_{failed}\}$	suspected failed devices	transient

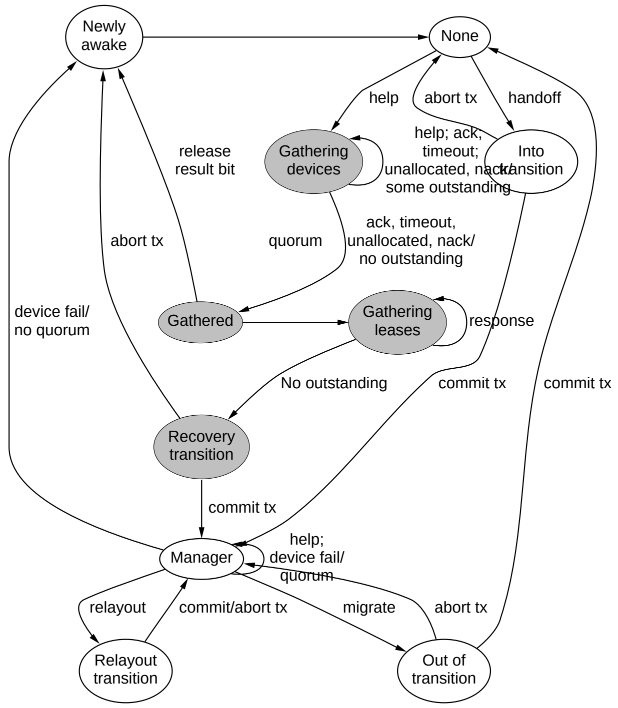

Figure 2. Manager protocol. States in the *recovery sequence* are highlighted. All states transition to the *newly awake* state on failure; these transitions are omitted for clarity.

4.3 Operation during one epoch

During an epoch, there is one active manager and a set of devices that have time-limited *leases* issued by that manager. The leases are used to detect possible failure.

Figures 2 and 3 summarize the protocol's state transitions for both devices and managers. Persistent state is preserved without corruption across a crash failure, while transient data is re-initialized on a crash.

Maintaining leases. Devices must periodically ask the manager to renew their lease (Figure 4). If the manager does not receive a lease renewal request from a device before the lease times out, the manager suspects that the device has failed and places it on a list of suspected failures. The manager will not renew leases for any device on the list, which implies that the device will not be able to serve read or write requests. The device remains on the list of suspects until either an outside management policy decides to permanently

variable	meaning	persistence
e	epoch number	persistent
$\langle V, D_V, \Lambda \rangle$	layout	persistent
d	data	persistent
i_{mgr}	active manager	transient
t_{lease}	lease timeout	transient

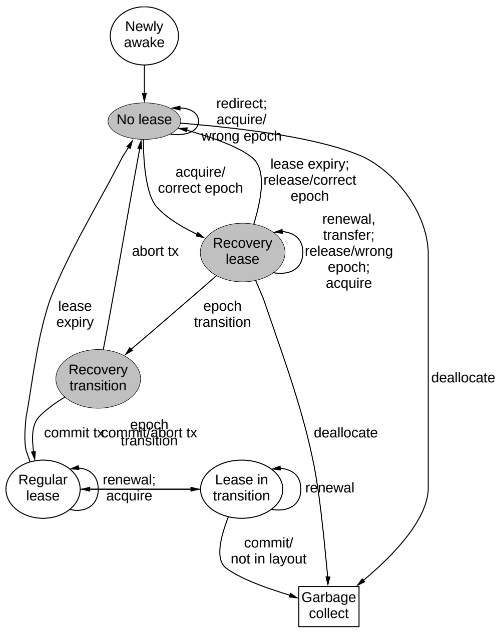

Figure 3. Device protocol. States in the *recovery sequence* are highlighted. All states transition to the *newly awake* state on failure; these transitions are omitted for clarity.

remove the device from the layout, or the device recovers, contacts the manager, and returns to service as part of an epoch transition.

When device suspects manager. If a device requests a renewal but does not get it, the device assumes that the manager has failed and starts the *manager recovery sequence,* discussed below. While the device is without a lease, the data access protocol is blocked to preserve the state of the data until recovery completes.

When manager suspects device. The active manager maintains a list of the devices that it believes have failed during the current epoch. A device that later returns to service will contact the manager for recovery, whereupon the manager will issue the device a *recovery lease* and add the device to a second list of devices that have come back. We discuss this recovery in more detail in Section 4.6.

Whenever the manager suspects a device failure, it must

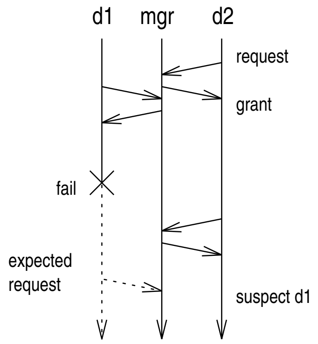

Figure 4. Timeline of communication during an epoch, showing lease renewals and failure suspicion.

determine if it still has both *quorum* and *coverage*. We define quorum as a majority of the devices in the epoch's layout, which is sufficient for our purposes but not strictly necessary: other kinds of quorums will work. This condition ensures that the devices in at most one partition can be active at a time. Note that quorum is only of the devices in the store, not of the entire system. A set of devices *covers* the store if for each byte of the store, at least one device in the set stores that byte, thus ensuring that all of the store's data is available.

If the active manager does not have coverage and quorum then it exits, thus causing the devices to initiate recovery when their leases expire.

4.4 Epoch transition

Epoch transitions move the store to the next period of consensus on the layout and active manager, either as part of a change in layout or a failure recovery.

An epoch transition consists of four steps (Figure 5). First, the manager initiates the transaction by sending a message to all the devices believed either not to have failed in the current epoch, or to have failed and recovered. Next, it initiates data reconciliation among any devices that are suspected to have failed and recovered, or that have been added to the layout. Once reconciliation is complete, the manager initiates an atomic commitment protocol. If the commitment protocol produces an "abort" result, each of the participants rolls back any layout changes they have made. (The reconciliations do not need to be rolled back.) If the commitment protocol commits, the manager and devices begin the epoch with fresh leases, which are established as a side-effect of committing and not through explicit communication. Finally, devices that have been removed from the layout "garbage collect" themselves.

Dealing with removed devices requires special attention. In the absence of failures, when a device is removed from

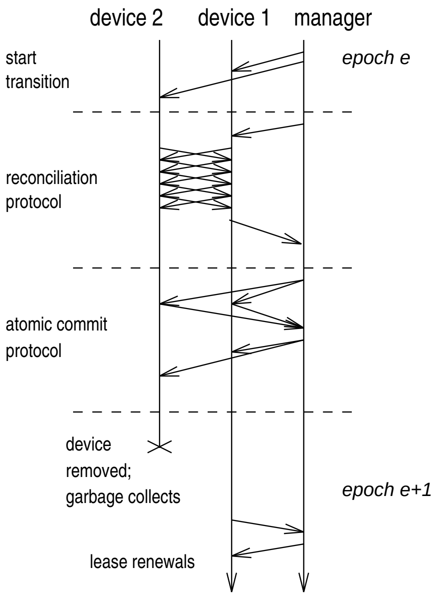

Figure 5. Timeline of communication during an epoch transition, showing data reconciliation, commitment, and garbage collection.

the layout it will participate in the transition from the last epoch where it is part of the store to the first epoch where it is not. The device detects this and garbage collects its data after commitment. However, if the device being removed is suspected of failure or has been partitioned away from other nodes, it will not be part of the epoch transition. When the failure is repaired the device will attempt recovery, as described in the next section, and in the process will discover that the rest of the system has moved on to a later epoch that does not include it and garbage collect its state.

4.5 The recovery sequence

When a device loses its lease, it suspects that the manager has failed and initiates the *recovery sequence,* which (if successful) results in the system returning to normal conditions. Figure 6 illustrates this process. A device can lose its lease for several reasons: because it has failed and recovered; because the manager has failed; or because either the manager or device falsely suspect the other of failure.

Recovery proceeds as a sequence of phases.

Initiation. The device uses the network's locating service (Section 3) to find a manager that it can communicate with, and asks that manager to become a *recovering manager* for the store, telling it the current layout and epoch number (step 1 of Figure 6). The device only asks one manager at a time, and if no manager responds the device will repeat this step until it gets a response.

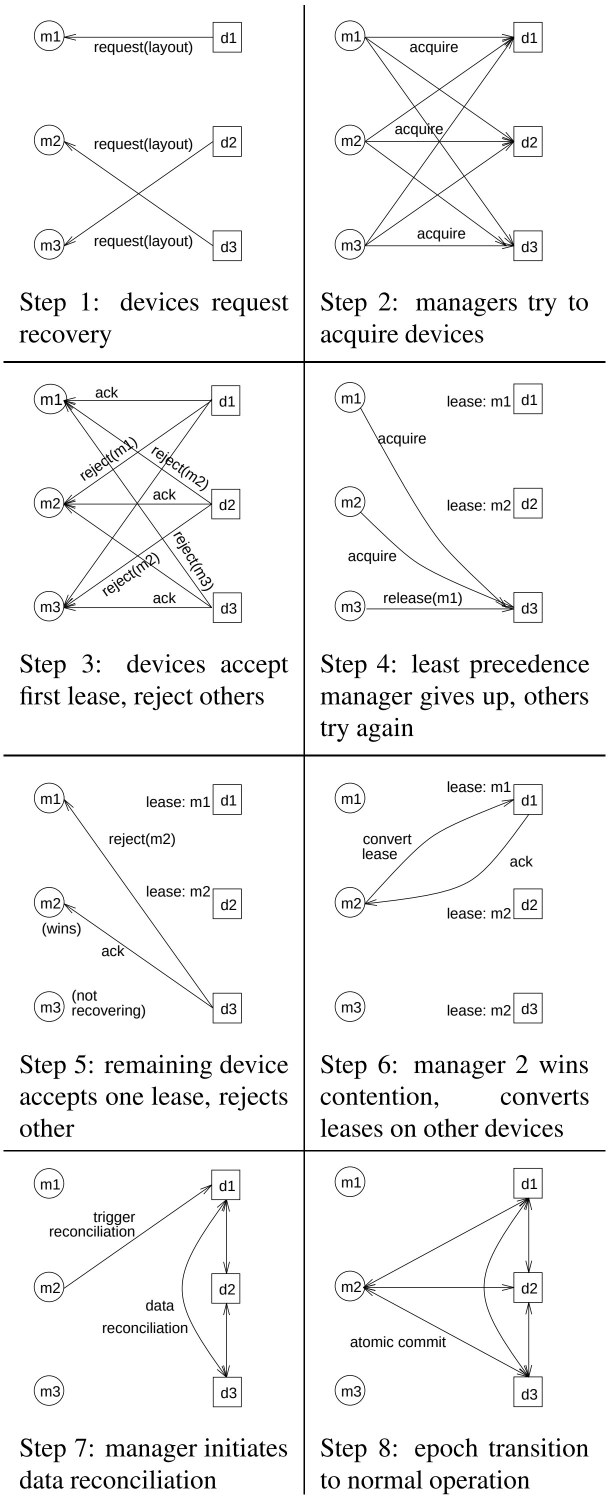

Figure 6. An example of the recovery sequence in a partition with three devices and three managers. All three devices request recovery, and manager m2 wins the contention.

When a manager node receives the request to begin recovery, it first checks to see whether it can contact an active manager for the store, using the distributed search structure provided by the layout retrieval protocol. If so, it forwards the request on to the active manager.

Note that several devices can lose their lease at about the same time if a manager has truly failed, resulting in several managers concurrently becoming recovering managers.

Contention. Once a manager accepts a recovery request, it enters into a sequence of contention rounds with any other managers that may also have become recovering managers for that store. Contention establishes which recovering manager will become the active manager. The rounds end when one manager obtains quorum and coverage and completes recovery, or when all recovering managers determine that obtaining quorum and coverage is impossible.

Each round consists of the managers trying to acquire as many of the devices it has not yet acquired as possible by issuing recovery leases to them (steps 2–5 of Figure 6). The recovering managers use these leases to detect probable device failure during recovery, and as a way of keeping the devices from initiating further failure recovery attempts for as long as the recovery manager is functioning. A device acknowledges the first lease it obtains, and rejects later ones with a message indicating the identity of the manager whose lease it did accept.

If one of the managers acquires coverage and quorum of the devices in the layout, it wins and completes recovery by moving to the next step. If a manager has lost the contention—and at least one will—it cancels the recovery leases it has issued and drops out of contention. If no manager has won, then all the managers wait a short while, then repeat the contention step, trying to acquire the devices that have been released by managers that have dropped out (e.g. managers 1 and 2 in step 4).

A recovering manager drops out of contention if it either acquires no devices, or if it determines that it is the lowest precedence manager among those contending according to the global precedence relation on manager ids (Section 3). The manager knows the ids of the other contending managers because a device sends the id of the manager that has already acquired it when the device rejects an acquisition.

Completion. The winning manager contacts all reachable devices that it has not acquired, and replaces their recovery lease with one it issues (step 6). Any devices that do not respond this acquisition are suspected of having failed, and become the initial list of suspected failures for the new epoch. The manager then initiates an epoch transition (steps 7 and 8), including data reconciliation, to return the store to normal service.

Failure. If a recovering manager fails during recovery, the devices' recovery leases will time out and the devices will again try to initiate recovery. If devices fail, they will fail to acknowledge lease renewals and be placed on the list of failed devices.

4.6 Reintegrating failed devices

Once a manager suspects that a device has failed, it will continue to do so through the rest of the epoch. If the device was falsely suspected, or if the device has failed and returns to service, the layout control protocol will effect its recovery during the next epoch transition.

Device recovery begins with the device suspecting the manager. The device will then initiate the recovery sequence, as discussed in the previous section, by finding some manager node and sending it a message requesting recovery. However, if there is still a reachable active manager, the request will be forwarded to it and the device will be reintegrated by that manager at the next epoch transition.

The device performs the same sequence of transitions when recovering from a small-scale failure that leaves the system with an active manager as it would when recovering from a large-scale failure that requires regenerating an active manager.

Once the active manager receives a recovery request from the device, it places the device on a list of devices believed to have recovered and issues the device a recovery lease. If the device fails again, the recovery lease will time out and the manager will take the device off its list of recovered devices. At the next epoch transition, any recovered devices are reconciled with their replicas and are issued leases in the new epoch.

The availability of the virtual store depends, in part, on how long devices must wait between their recovery and the next epoch transition. Epoch transitions are triggered by management policies outside the layout control protocol (except during large-scale failure recovery) and so availability will be highest if those policies respond quickly to news of device recovery.

5 Correctness

In this section we argue the correctness of a few key properties of this protocol.

5.1 Liveness of recovery

We first show that the recovery sequence (Section 4.5) completes in bounded time. We begin by formally defining some of the key concepts in recovery.

Definition 1 *A* virtual store V *is a finite set of bytes.*

Moreover, virtual stores are distinct. Namely for two virtual stores $V, V', V \cap V' = \emptyset$.

Definition 2 *The* layout, $l(V)$, *of a virtual store V is a tuple* $\langle V, D_V, \Lambda\rangle$, *where* $D_V \subseteq D$ *and* $\forall b \in V, \Lambda : b \rightarrow D_b \subseteq D_V$.

In words, the layout of a virtual store represents the mapping of each byte of the virtual store to a set of devices. Each device of the set stores the replicas of that byte of the virtual store. In practice, a store would be broken into byte ranges (or sets of byte ranges in the case of a striped layout) that would each be replicated across devices. However, the given representation is equivalent and simplifies the notation somewhat.

Definition 3 *Given the layout of a virtual store* $l(V) = \langle V, D_V, \Lambda\rangle$, *a set of devices* $D' \subseteq D_V$ covers *the set of bytes* $cov(V, D') = \{b | b \in V \wedge D' \cap \Lambda(b) \neq \emptyset\}$.

In words, the set of devices D' covers the subset of the virtual store for which it stores at least one replica.

Definition 4 *A set of devices D' is said to have* coverage *of a virtual store V if* $cov(V, D') = V$.

Definition 5 *A set of devices D' is said to be* quorum *of a virtual store V with layout* $\langle V, D, \Lambda\rangle$, *if* $D' \subseteq D$ *and* $|D'| > |D|/2$.

The manager recovery sequence consists of one or more devices initiating recovery concurrently, followed by managers performing one or more rounds of the contention phase, with one manager performing the completion once.

Note that in the discussion that follows, we use the term "component" to mean a connected component of the system network in the graph-theoretic sense.

Lemma 1 *If a device without a lease is in a component of the system that contains at least one manager node, some manager in that component will enter the contention phase within bounded time.*

Proof: Upon initiating recovery, the device obtains a finite list of possible managers from the network in time δ_{mgr}, then attempts to contact each manager sequentially. Each contact attempt requires at most $2 \cdot (\delta_{step} + \delta_{msg} + c)$ time. By assumption, at least one manager among the list will respond. ∎

Likewise, the completion phase is bounded, consisting of a fixed sequence of activities for each device followed by an atomic commit.

Lemma 2 *A single round of the contention phase of the recovery protocol terminates in bounded time.*

Proof: A round of the contention phase is defined by a single manager's attempt to acquire leases on the devices (D_V) of the virtual store. This entails at most $|D_V|$ round-trip message times or time-out periods. Since $\delta_{timeout} > 2\delta_{msg} + \delta_{step}$, a single round takes time at most $|D_V|\delta_{timeout}$. ∎

Definition 6 *A time period t is* stable *if no failures occur in the time period, and it is longer than the bounded recovery time.*

Definition 7 *The state of the system is said to be* recoverable *with respect to virtual store V if there exists a component of the system that contains a manager and has coverage and quorum or V.*

Lemma 3 *If the system is in a recoverable state at the beginning of a stable period, it remains in a recoverable state throughout the stable period.*

Proof: By way of contradiction, assume the state is no longer recoverable at the end of a stable period. Thus, the component that had a manager, coverage and quorum initially is now missing at least one of these properties. Thus, some element of that component must be partitioned away from the component. This contradicts the assumption that the period was stable. ∎

Lemma 4 *If a virtual store has an active manager at any point during a stable period, then it has an active manager at the end of the stable period.*

Proof: By way of contradiction, assume there is no longer an active manager for the store at the end of a stable period. In order for this to happen either the manager failed, or devices that were constituting either coverage or quorum were partitioned away from the component, causing the manager to exit. Any of these cases require that some type of failure occurs, contradicting the assumption that the period was stable. ∎

Definition 8 *The set of* potential managers *of a virtual store with layout* $\langle V, D_V, \Lambda\rangle$ *is comprised of the managers that respond with acquire messages in response to a request for recovery from a device of the store.*

Lemma 5 *In a stable recoverable system, the set of managers involved in the recovery protocol for store V with layout* $\langle V, D_V, \Lambda\rangle$ *is at most* $|D_V|$.

Proof: At the end of the stable period the set of devices (of the virtual store layout) in the component with a manager, coverage and quorum is of size at most $|D_V|$. Since each of these devices solicits management serially, and no failures occur during the stable period, there is at most one manager acting on behalf of each device. Thus there are at most $|D_V|$ initial managers, let this set be denoted M. Since only managers with leases will be solicited for help in recovery in later contention rounds, managers in M are the only managers that will acquire leases throughout the recovery protocol. Since $|M| \leq |D_V|$ the lemma holds. ■

Lemma 6 *In a recoverable stable system, the recovery protocol terminates with a single manager in bounded time.*

Proof: For virtual store V with layout $\langle V, D_V, \Lambda\rangle$, let D_c denote the subset of D_V in the system component with a manager, coverage, and quorum at the end of the stable period. By Lemma 5 there are at most D_c managers in the component competing in the recovery protocol, each of which can be credited to an initial request from some device of D_c.

After each round of contention, each manager either wins, exits, or begins another round of acquires. After one round by each manager, either a manager wins or a manager exits (possibly both will occur). If a manager exits it either gained no leases or received nacks only from devices with higher precedence managers. This exit event can be credited to the device on whose behalf the manager joined contention, and (by Lemma 2 occurs within the bounded time it takes for the exiting manager to complete a round. At most $|D_c| - 1$ exit event can take place, each one credited to a different device, and each one taking bounded time after the last event occurs. Thus, in bounded time either a manager has won recovery or there is only a single manager left in the contention protocol. In the latter case, since we assume the component was recoverable, the manager must have coverage and quorum, and thus wins the protocol.

Since the period is stable, no failures occur during recovery. Due to the property that a device has only a single manager, and because at most one manager can hold a quorum of the leases, the lemma holds. ■

Theorem 1 *If the system is recoverable with respect to virtual store V with layout $\langle V, D_V, \Lambda\rangle$, then at the end of the stable period V has a single active manager.*

Proof: By Lemma 1 the sequence takes a finite time to start. By Lemma 6 the recovery protocol terminates with a single active manager. By Lemma 4 once there is a single active manager there will be one until a failure occurs, i.e., at the end of the stable period. ■

5.2 Safety

Safety constitutes ensuring one-copy serializability [10] of data, which means that replicas appear to be consistent with each other and that there is proper serialization of reads and writes. Fundamental to this property is the requirement that there be at most one actively managed set of devices storing replicas of the virtual store (lest independent sets diverge due to different streams of writes). In addition, we must ensure that the active set of replicas of the virtual store has one-copy serializable layout information, and that correct resource allocation is maintained throughout.

5.2.1 At most one active manager

Definition 9 *For a virtual store V, the set of* active managers *for V (denoted $A(V)$) is defined to be the set of all managers that have issued regular leases (i.e. not recovery leases) on devices in V.*

In the ideal, a virtual store would always have exactly one active manager. Failures, however, mean that sometimes there will be none, either because the manager failed or because it can no longer communicate with a quorum and coverage of devices. Managers have a lag time before they detect device failures, limited by the device lease renewal period. This makes it possible for one manager, which has been active, to be "doomed" to exit because of failures it has not yet detected, while another manager is being regenerated by the failure recovery sequence. This transient overlap may lead to more than one manager being active. In practice we expect these transient periods to be short and infrequent. In Section 5.2.3, we will show that this does not affect data consistency.

Theorem 2 *For any virtual store V, at all times except for the device lease renewal period after a failure, $|A(V)| \leq 1$.*

Proof: By assertion, in the stable state of the system, each virtual store has exactly one active manager.

When a failure or repair occurs there are several possible outcomes. If there is no component that has coverage and quorum of devices, then no manager can be active and any previously active manager will exit within time bounded by the device lease renewal period. There can be at most one component with quorum and coverage, and if one exists, it either already contains an active manager or it does not. In the former case, management will continue; in the latter case management will be regenerated within bounded time. Any other active manager will suspect sufficient device failures within the device lease renewal period, and will exit. Thus, except for brief periods after failure, $|A(V)| \leq 1$. ■

5.2.2 Correct resource allocation

In any practical distributed system, it is important that resources be neither leaked, leading to unbounded resource consumption, nor deallocated too early, leading to data loss. In this section we show that the layout control protocol meets these conditions.

Theorem 3 *No device garbage collects its state without having been explicitly removed from the layout through an epoch transition.*

Proof: Devices garbage collect their state in two cases: when they participate in an epoch transition that removes them from the layout, or when they are notified that they have an outdated epoch number and are no longer in the layout. In the first case is clear that garbage collection is safe.

In the second case, the device receives a "deallocate" message from a manager. This message is only sent from an active manager when it receives a request for recovery from a device with an old epoch number that is not part of the current layout. The device must have been part of the layout in some past epoch, and is not now. Since layouts are only changed through epoch transitions, there must have been some epoch transition where the device was removed. ■

Theorem 4 *Any device that is removed from the layout in an epoch transition that commits eventually garbage collects its state, assuming the device eventually is part of a recoverable component containing an active manager.*

Proof: There are three cases to consider, based on the state of the system when the epoch transition occurs.

First, when the device has not failed, and the manager believes it has not failed, the device will participate in the epoch transition and garbage collect its state at the end.

Second, when the device has failed, whether or not the manager believes it to have failed, the device will eventually recover and initiate recovery. Eventually that recovery sequence will contact the active manager for the store because the device will eventually be in a (recoverable) component containing the active manager, and any recovery attempt in that component will cause the device to forward its help request to the active manager. The active manager will then inform the device that it should garbage-collect its state.

Finally, when the device has not failed, but the manager believes it to have failed, no manager will issue leases to the device. Within a bounded time the device's lease will time out, and the device will begin recovery as in the previous case. ■

We cannot make the stronger claim, that all removed devices are eventually garbage collected, in the presence of permanent failures because the device blocks garbage collection until it receives notice from an active manager. The less-strong claim, that garbage collection occurs if all failures are eventually repaired, does not hold because pathological sequences of overlapping partitionings could occur, each partition being eventually repaired but never allowing the device to contact an active manager.

5.2.3 Data consistency

We now turn to showing that the data is maintained correctly. We show first that the store's layout is kept correct.

Theorem 5 *The layout of each store remains one-copy serializable.*

Proof: By Theorem 3 no device gets garbage collected prematurely, and by the property that epoch and layout changes are made through a transaction, one-copy consistency is maintained from one layout to the next. In addition, the layout changes are performed serially, yielding one-copy serializability. ■

We make the following assumptions about the underlying data access protocol:

- A write to a virtual store can be viewed as an atomic action that either returns success or nothing.
- If a write operation returns success, it has succeeded (i.e. can be viewed to have happened atomically).
- If a write operation returns nothing, either it succeeded, and can be viewed as an atomic operation, or it fails, and no data is written to any part of the store.
- There exists a linearization of the successful writes consistent with every read value returned.

That is, in the absence of failure the data access protocol provides one-copy serializability.

Lemma 7 *[1] Except in the case of unrecoverable data loss due to device failure, the reconciliation protocol will terminate with data of a virtual store consistent and serialized.*

Theorem 6 *The data in managed virtual stores is one-copy serializable.*

Proof: When no failures occur, the underlying data access protocol ensures one-copy serializability of data.

In the case of failure (barring unrecoverable data loss) either the store remains on line and the management is not interrupted, or the management will be recovered eventually. In the first case, the underlying data access protocol will ensure one-copy serializability. In the second case, the reconciliation procedure will be run, and by Lemma 7 the data will be consistent and serialized upon transition back into the managed state.

In the case of unrecoverable data loss, outside intervention is necessary to get the data back into a consistent (and managed) state. ■

6 Extensions

In the presentation so far, we have made some simplifications for clarity.

Single manager. Palladio actually replicates a store's active manager, using two replicas and one witness [17]. Normally, one manager can fail or be partitioned away, and the remaining managers will attempt to regenerate the missing replica. This increases the apparent reliability of the manager, making manager recovery rare.

Whole devices. We have presented devices as the unit of replication, in practice Palladio uses a smaller granule. A device stores multiple *chunks*, similar to disk partitions or array LUNs, each of which acts independently in the layout control protocol.

Reintegrating devices. The way device failure is handled can be optimized. Currently, when a device fails while a store is active, the manager puts the device in its list of failed devices and it stays there until the data reconciliation protocol is triggered as part of an epoch transition. While this is correct, it means that devices can be locked out for a long time. This can be improved as follows: when a device recovers in the active partition, it will request manager recovery. This request will be forwarded to the active manager, which can use the message as an indication that the device has recovered. The manager can bring the device up to date by triggering the data reconciliation protocol. Once complete, the manager can issue the device a lease, which will return the device to normal operation, and remove the device from its list of failed devices.

Synchrony model. We have assumed a synchronous system model to simplify the analysis of correctness. We are investigating whether the system is correct under a more relaxed model, such as timed asynchrony [9]. The only difference between our current assumptions and this model is it permits performance failures in the network, resulting in late messages.

Failure suspectors. The layout control protocol we have presented uses leases as failure suspectors, but leases only work in systems with limited failure and synchrony semantics. The layout control protocol uses the failure suspector to control the access protocol: the device blocks any I/O requests while it suspects manager failure. We are investigating whether we can substitute a stronger failure suspector into the layout control protocol, as part of making the protocol resilient to a wider range of failures, without disturbing the other protocols.

7 Related work

Several other distributed storage systems have been built. The Petal system [16] provides distributed virtual disks that are implemented using a RAID-5 layout over a set of storage servers. The Palladio work differs from Petal in aiming to scale much larger than Petal, and in using more complex layout schemes to implement virtual stores, allowing resource usage to be tailored to a store's performance needs. The Swift system [6] provides striped storage over a set of servers, somewhat like Petal does. The Cheops effort [1], part of the CMU NASD project, focuses on providing file system services, rather than storage services, to hosts; however, it internally uses optimistic concurrency control for building distributed RAID-5 disk arrays. The data access protocol used in Palladio was developed in collaboration with this project. The Global File System (GFS) project [21] also focuses on building a shared file system service, and internally uses a network storage pool mechanism that provides similar semantics to Palladio virtual stores. GFS uses device-implemented locking mechanisms for concurrency control.

None of the projects mentioned are explicitly investigating scale—indeed, GFS explicitly uses search to find the parts of accessible storage pools—and only the Petal project has so far focused on recovery after failure.

The layout control protocol fills a similar conceptual role in Palladio as do membership mechanisms in group communication systems [20, 22]. The epoch transition provides a way for devices to join and leave the group of devices supporting a virtual store. Moreover, virtual storage systems naturally decompose into subcomponents in a way similar to group communication systems [2, 13]. Storage systems, however, experience very infrequent layout changes compared to the frequency of read and write requests, and membership changes are hidden from (and uninteresting to) the applications reading and writing data. This leads to a different set of design choices in the decomposition and the protocols—notably, the use of a heavyweight epoch transition mechanism in order to make individual I/O requests simple and fast.

In comparison to quorum systems, we use the very simple concept of quorum as a strict majority over each virtual store layout group. This approach is similar to work on use of quorum for replication control [18, 11]. We have not yet investigated if more generalized quorum systems [19] are applicable (or necessary) for our target of data storage systems in the face of site failure and arbitrary partitions.

8 Conclusions

We have designed a replication management system that automatically and correctly recovers from the large-scale failures we expect in distributed data storage systems, such as site-wide power failures and network trunk partitioning.

There are a few key ideas we used in designing the system.

Modular protocol design. We divided the overall problem of replicated data management into subproblems, with a protocol specialized for each and clean interfaces between them. We also made use of atomic commitment and failure suspicion as building blocks, which made design and analysis easier and more likely to be correct.

Active device participation. Devices initiate recovery, and have all the information needed to do so. This ensures that recovery begins quickly and that it does not involve a wide-scale search for the devices that make up the store. This should improve recovery time, and will use fewer communication resources than an approach based on centrally-maintained global state or on passive devices that must be located after failure.

Distributed management function. Organizing management on a store-by-store basis reduces the amount of information any one manager must maintain, reduces the effects of a single node failure to only a few stores, limits the scope of activity when a store's layout changes, and enables quick, as-needed management regeneration.

Coverage and quorum condition. This condition determines whether a store is available for service. The combination requires that all of the data in the store be available, as well as ensuring that at most one partition in the system can proceed. While quorums are a well-known idea, the combination with coverage is important for storage systems.

Acknowledgments

The authors thank Peter Bosch (Universiteit Twente), Randal Burns (UC Santa Cruz), John Wilkes (HP Labs), and the anonymous reviewers for their comments on the paper.

References

[1] K. Amiri, G. A. Gibson, and R. Golding. Scalable concurrency control and recovery for shared storage arrays. Technical Report CMU–CS–99–111, Dept. of Computer Science, Carnegie-Mellon Univ., 1999.

[2] N. T. Bhatti, M. A. Hiltunen, R. D. Schlichting, and W. Chiu. Coyote: a system for constructing fine-grain configurable communication services. *ACM Trans. on Computer Systems*, 16(4):321–66, November 1998.

[3] A. D. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart. The Echo distributed file system. Technical Report 111, Digital Equipment Corp. Systems Research Center, Palo Alto, CA, September 1993.

[4] E. Borowsky, R. Golding, P. Jacobson, A. Merchant, L. Schreier, M. Spasojevic, and J. Wilkes. Capacity planning with phased workloads. In *Proc. of the 1st Workshop on Software and Performance*, October 1998.

[5] E. Borowsky, R. Golding, A. Merchant, E. Shriver, M. Spasojevic, and J. Wilkes. Using attribute-managed storage to achieve QoS. In *Proc. of the 5th Intl. Workshop on Quality of Service*, June 1997.

[6] L.-F. Cabrera and D. D. E. Long. Swift: a storage architecture for large objects. In *Proc. of the 11th IEEE Symp. on Mass Storage Systems*, pages 123–8, October 1991.

[7] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. In *Proc. of the 11th ACM Symp. on Principles of Distributed Computing*, pages 147–58, 1992.

[8] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–67, March 1996.

[9] F. Cristian and C. Fetzer. The timed asynchronous system model. Technical Report CSE97–519, Computer Science Dept., Univ. of California at San Diego, 1997.

[10] A. El Abaddi and S. Toueg. Maintaining availability in partitioned replicated databases. *ACM Trans. on Database Systems*, 14(2):264–90, June 1989.

[11] D. K. Gifford. Weighted voting for replicated data. In *Proc. of the 7th Symp. on Operating Systems Principles*, pages 150–62, December 1979.

[12] R. Golding and E. Borowsky. The Palladio failure recovery protocol. Technical Report HPL–SSP–99–1, Storage Systems Program, Hewlett-Packard Laboratories, March 1999.

[13] R. A. Golding and D. D. E. Long. Using an object-oriented framework to construct wide-area group communication mechanisms. In *Proc. of the Int. Symp. on Applied Computing: Research and Applications in Software Engineering, Databases, and Distributed Systems*, 1993.

[14] C. G. Gray and D. R. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. of the 12th ACM Symp. on Operating Systems Principles*, pages 202–10, December 1989.

[15] R. Guerraoui and A. Schiper. The decentralized non-blocking atomic commitment protocol. In *Proc. of the 7th IEEE Symp. on Parallel and Distributed Processing*, October 1995.

[16] E. K. Lee and C. A. Thekkath. Petal: distributed virtual disks. In *Proc. of the 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, October 1996.

[17] D. D. E. Long and J.-F. Pâris. Voting with regenerable volatile witnesses. In *Proc. of the 7th Int. Conf. on Data Engineering*, pages 112–19, April 1991.

[18] D. D. E. Long and J.-F. Pâris. Voting without version numbers. In *Proc. of the Intl. Conf. on Performance, Computing, and Communications*, pages 139–45, February 1997.

[19] D. Malkhi, M. Reiter, and R. Wright. Probabilistic quorum systems. In *Proc. of the 16th ACM Symp. on Principles of Distributed Computing*, August 1997.

[20] D. Powell. Group communication. *Communications of the ACM*, 39(4):50–3, April 1996.

[21] K. W. Preslan, A. P. Barry, J. E. Brassow, G. M. Erickson, E. Nygaard, C. J. Sabol, S. R. Soltis, D. C. Teigland, and M. T. O'Keefe. A 64-bit, shared disk file system for Linux. In *Proc. of the 16th IEEE Symp. on Mass Storage Systems*, March 1999.

[22] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: a flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.