# Niobe: A Practical Replication Protocol

JOHN MACCORMICK, CHANDU THEKKATH, MARCUS JAGER,
KRISTOF ROOMP, LIDONG ZHOU
Microsoft Research
RYAN PETERSON
Cornell University

The task of consistently and reliably replicating data is fundamental in distributed systems, and numerous existing protocols are able to achieve such replication efficiently. When called on to build a large-scale enterprise storage system with built-in replication, we were therefore surprised to discover that no existing protocols met our requirements. As a result, we designed and deployed a new replication protocol called *Niobe*. Niobe is in the primary-backup family of protocols, and shares many similarities with other protocols in this family. But we believe Niobe is significantly more *practical* for large-scale enterprise storage than previously-published protocols. In particular, Niobe is simple, flexible, has rigorously-proven yet simply-stated consistency guarantees, and exhibits excellent performance. Niobe has been deployed as the backend for a commercial Internet service; its consistency properties have been proved formally from first principles, and further verified using the TLA+ specification language. We describe the protocol itself, the system built to deploy it, and some of our experiences in doing so.

Categories and Subject Descriptors: H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*

General Terms: Reliability, Verification, Algorithms

Additional Key Words and Phrases: Replication, enterprise storage

## 1. INTRODUCTION

This paper describes our experience with designing and deploying a storage backend for large scale web services. The backend was designed to replace an existing backend that had been functioning well for some years, and a key objective was to substantially reduce the system's operations and hardware costs. Although the new system was built with the existing web services in mind, the design principles (especially the replication protocol) are more generally applicable and are not tied to any proprietary service. The overall experience with our design choices in a commercial environment serving customer data has so far been favorable.

The storage backend consists of three inter-related components: a high-performance full-text indexing and query engine, a flat storage subsystem, and a policy module that uses the mechanisms provided by the storage subsystem to implement policies

...

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 0000-0000/20YY/0000-0001 $5.00

specific to the service (e.g., controlling the degree of replication for classes of users). In this paper, we will focus on the details of the storage subsystem. To aid in understanding, we provide sufficient detail about the indexing engine, which is fairly conventional in design, and the policy module, which is specific to our particular deployment.

A key component of our backend is a scalable replicated storage subsystem called *Niobe* and the associated protocols.[1] A primary objective of this paper is to describe the design of Niobe. We have found our design to be sufficiently simple, yet efficient and reliable to be deployed at scale in data centers serving real commercial workloads. In addition, we have proved certain correctness properties of the protocol from first principles, and verified some related properties using TLA+ [Lamport 2002], a formal specification language. This gives us a precise understanding of what guarantees the system can provide to its clients, which in turn enables us to offer meaningful service level agreements (SLAs) to customers.

When we set out to design the storage backend, we were not trying to invent new replication protocols. Indeed, three of the authors had substantial prior experience in designing replication protocols [Lee and Thekkath 1996; Chang et al. 2002; MacCormick et al. 2004] and we believed we could use one of these designs that had proven successful in the past. However, on closer evaluation, we discovered that none of these (or other published protocols we were aware of) provided the features we needed:

—Support for a flexible number of replicas (two or more).

—Strict consistency in the presence of network, disk, and machine failures: reads and writes to a given data item must be atomic and reads must return the last completed write.

—Efficient common case read and write operations without requiring potentially expensive two- or three-phase commit protocols.

—Simplicity (must be easy to implement, deploy, and maintain; and have consistency properties that are simply stated): these requirements were important because our principal goal was deployment at scale by a sizable engineering team, rather than a proof-of-concept implementation by a small group of researchers.

Because of the nature of our environment, we were able to make a few simplifying assumptions:

—Single administrative domain. Since all replicas were expected to reside in data centers owned and operated by a single company, there was little motivation to deal with the complexities of data in multiple administrative domains.

—Non-byzantine failures. While we recognize that byzantine faults do occur in systems, our experience with the existing deployment suggested that this type of failure need not be the primary concern for the first release of the new system.

[1] We hope our system will be as successful at replication as the Niobe of Greek mythology, a mortal woman who had 14 children. It should be noted, however, that Niobe's children were all slain by Apollo and Artemis after she boasted about her fecundity. We will therefore try to avoid exaggerated claims about the Niobe storage system and replication protocol.

The rest of the paper is structured as follows. We first provide an overview of the system in the next section. Subsequent sections describe the details of the Niobe protocol. We then describe our usage experience with deploying this system and offer some conclusions based on our experience.

## 2. SYSTEM DESCRIPTION

This section provides a high-level system overview of Niobe and some of the related components with which it interacts. The system provides facilities to store, index, and retrieve data, while managing the data storage in a scalable, cost-effective way in the face of failures and load imbalances.

External entities interact with the system via *front ends* that act as clerks and provide connection management and caching facilities. For example, the external entity might be an end-user's browser, while the front ends might be application servers interacting with the end-user's data (such as Web pages, blog posts, instant messages or e-mail messages) that happens to be stored in Niobe. From Niobe's perspective, front ends and external entities are outside the system; we will not describe them any further in this paper. In particular, we do not describe the details of how external entities locate an appropriate front end for their data.

Niobe software consists of three major pieces: an indexing and query engine, a replicated storage substrate, and a policy manager as shown in Figure 1. A Niobe installation consists of multiple machines (typically on the order of a few hundreds) that act in concert to index, store, and retrieve data. Each machine runs an instance of an indexing and retrieval engine, a distributed policy manager, and a distributed storage subsystem. In addition to the machines hosting the indexer, policy manager, and the storage system, we have a small number of machines (less than 10) that run a *global state manager* (GSM) module. The GSM module uses a consensus service (e.g., Paxos [Lamport 1998]) to reliably store critical global state used by the storage and the policy manager subsystems.

The indexing machinery indexes received content and retrieves content based on received queries. The indexing engine uses the facilities of the distributed storage system to store its data and indices. The design of the indexing engine, though challenging, follows established ideas from the literature [Barroso et al. 2003; Risvik et al. 2003] and is not the focus of this paper.

The policy manager is distributed. It consists of a central service and a component that runs on each machine, which works in concert with other instances on other machines to implement service specific policies. The primary purpose of the policy manager is to determine when and where a replica should be relocated (i.e., on which physical machine). Relocations can occur as a result of a failure or a load imbalance. Therefore, the policy manager uses sub-components for detecting failures and load imbalances. These are known as the failure detector service and the load monitoring service, respectively. In addition, the policy manager includes a controller service, which makes decisions on, and initiates, replica relocations, based on inputs from the failure detector and load monitoring services.

The distributed storage subsystem works in concert with other instances of itself on other machines to implement replicated, scalable, fault-tolerant storage. The rest of the paper describes the details of the distributed storage system and the

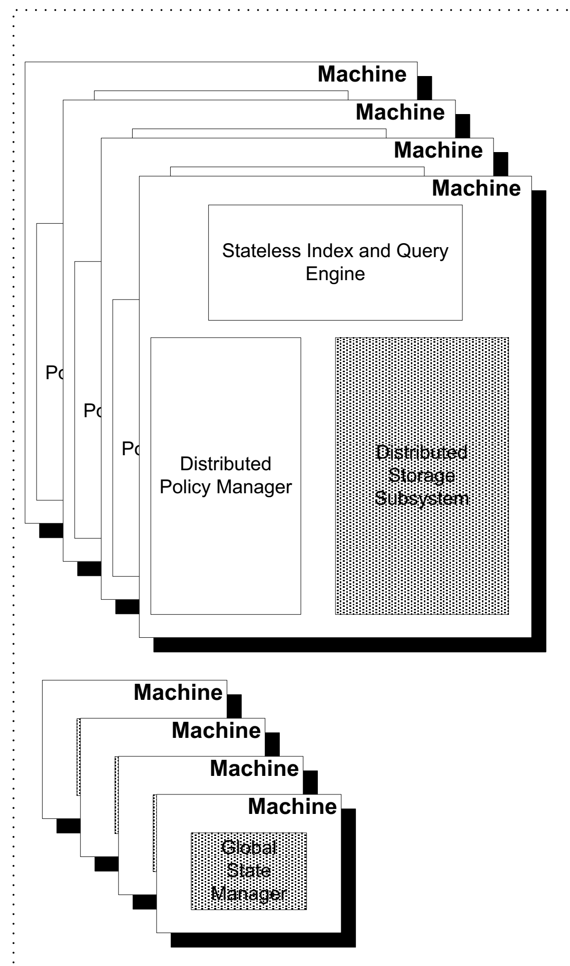


Fig. 1. **Overview of the Niobe System Architecture**

global state manager. For now, we will merely highlight Niobe's basic replication strategy and semantics and defer a fuller treatment until later.

In Niobe, every data item is replicated on multiple machines. The degree of replication is a configurable parameter of the system. At any time, precisely one of these replicas is designated the *primary*. All writes as well as reads are fielded by the primary (in contrast to strategies such as chained declustering [Hsiao and DeWitt 1990]) and uniform loading is achieved by having each machine act as the primary for a roughly equal number of frequently accessed data items.

When a write is sent to a primary, it ensures, before declaring the write successful and returning to the client, that a configurable number of replicas reliably store a copy of the data, and that any replicas which may have failed to store the data are declared dead. Reads of data items sent to the primary replica are immediately satisfied. Read or writes sent to secondary replicas result in an error return to the

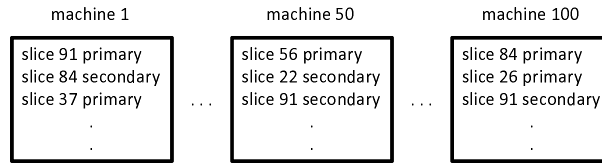


Fig. 2. Layout of data items or "slices"

caller.

The truth about the identity of the primary and which secondaries are up and which are down is maintained in the global state manager (GSM) module. Status changes in the replica set (e.g., takeover of a primary or the reincarnation of a dead replica) are reliably recorded by the GSM before any new requests are accepted.

Normal case reads and writes do not require the services of the GSM. As a result, they are efficient. Failures can cause in-flight writes to a data item not to be applied at some replicas. Niobe uses a recovery mechanism based on a dirty-region log, and the GSM ensures data is consistent before clients are allowed access to the affected data item.

Niobe offers a data model that is very simple, straightforward, and adequate for clients to implement a variety of complex semantics. Niobe provides access to a non-hierarchical collection of data items, much like a disk. It offers single-copy semantics on each data item, but does not offer any ordering guarantees between concurrent writes to different items. If a write to a data item does not succeed, the data item is guaranteed to have either the old value or the new value. In the absence of a successful return from Niobe, a client has no way of knowing if a write succeeded or failed except by successfully reading the value of the data item.

## 3. THE NIOBE PROTOCOL

This section describes the Niobe replication protocol. For simplicity, the description will assume only one "data item" is being replicated, on some small fixed set of $N$ machines (say, $N = 2$ or 3). At any one time, one of these machines is the primary and the others are the secondaries. In practice, an implementation of Niobe involves at least tens, if not hundreds, of machines, with thousands of separate data items being replicated—each machine acts as either the primary or a secondary for many different data items at once, and the assignment of primary and secondary replicas to individual data items is done in any suitable manner. In our deployment, for example, replicas are assigned pseudo-randomly subject to the constraint that the replicas of any one data item must be hosted by machines connected to distinct network switches and power distribution units (PDUs). This ensures all data items remain available after the failure of a switch or PDU. We refer to our data items as "slices" of data; a possible layout is shown in Figure 2.

It would be impractical for every file, web page, blog post, or e-mail message in the system to be a separate data item because we would have to keep global state (to be described in the next section) about each of them in the GSM. Instead, we aggregate data belonging to a fixed large collection of users into a "slice" to keep the amount of data stored in the GSM manageable. Although we have found in practice that a single GSM can support a large number of slices, it would be trivial

to extend the system to multiple GSMs (using a fixed mapping of slices to GSMs) if necessary.

### 3.1 The Niobe state machine

As already noted, the protocol description will restrict attention to a single data item, so we may focus on only the $N$ *replicas* of this data item, denoted $R_1, R_2, \ldots, R_N$. The other entities participating in the protocol are the GSM and the front-end machines. The entities can send messages to one another, which can be arbitrarily delayed, dropped or reordered. The entities are not malicious. They may operate arbitrarily slowly (except for their clocks) and may halt or "reboot" (defined below) at any time. The entities possess clocks whose drift is bounded by a known constant factor. We assume every entity possesses two types of storage: "memory", which is lost on reboot, and "stable storage" which survives reboot. We formally define *reboot* for replicas as follows: when a replica reboots, it executes the `Reset()` algorithm of Section 4.1 using only its stable storage as input. Every entity uses standard techniques[2] to apply mutations to its stable storage atomically.

Formally, the protocol maintains a replicated state machine comprising:

—a vector of *health* bits $H = (h_1, h_2, \ldots, h_N)$, with each bit $h_i$ taking values in $\{\mathsf{Alive}, \mathsf{Dead}\}$

—an *epoch number* $E$, which is a nonnegative integer

—a *primary ID* $P$, taking values in $\{1, 2, \ldots, N\}$

—a *data value* $D$, taking values in some set of possible data values $\mathcal{D}$.

An informal interpretation of this state is as follows. The health bits signify which replicas are currently faulty or out-of-date. The epoch number is incremented every time any change to the health bits occurs (so replicas can use it to check that their view of the other replicas' health is up-to-date). Replica $R_P$ is known as the *primary*, and the other replicas are called *secondaries*. The data value $D$ is the current value of the data being replicated on behalf of the customers.

In contrast to many other implementations of replicated state machines, the entire state is not stored at every entity participating in the protocol. In Niobe, only the GSM stores the health bits, epoch number and primary ID in stable storage. The replicas cache values of these quantities in memory, but need never commit them to stable storage. The replicas store the data value $D$ in stable storage – we'll sometimes refer to this stored value as the replica's *local* data value, to distinguish it from the ethereal $D$ that resides in the replicated state machine. The GSM never stores the data value, even in memory.

The state machine maintains the invariant that the primary is always alive i.e. $h_P = \mathsf{Alive}$. The only permissible changes in the state machine are:

—"kill or reincarnate a secondary": flip one of the non-primary health bits and increment the epoch number i.e. atomically flip $h_r$ for some $r \neq P$, and set $E \leftarrow E + 1$

[2]For example, write-ahead logging, with a local recovery protocol at reboot time.

—“reassign the primary, killing old primary”: change the identity of the primary to some other replica that is alive, and kill the old primary i.e. atomically set $P \leftarrow r$, for some $r \neq P$ such that $h_r = \mathsf{Alive}$, set $h_P = \mathsf{Dead}$, and set $E \leftarrow E + 1$.

—“update the data item”: set $D \leftarrow D'$ for some $D' \in \mathcal{D}$. There may be implementation-specific restrictions on what updates are permissible. For example, our deployment is optimized for appends and deletions. However, the protocol is useful and efficient for arbitrary updates, including atomic “compare-and-swap” mutations in which the new data value depends on the old one.

The GSM module ensures that all changes to the health bits, epoch number and primary ID are serialized, legal, and atomic. We refer to any such change as a GSM *decree*.

### 3.2 The Niobe services

It's convenient to think of each Niobe replica running some subset of four independent processes[3], called the *clerk process*, the *maintenance process*, the *coordinator process* and the *reset process*. The coordinator process is active only on replicas that believe they are the primary; the maintenance process is active only on replicas that believe they are secondaries; the clerk process is active on all replicas. The reset process runs only at startup and during reconfiguration.

We can describe Niobe as comprising 4 distinct client-server services: the *data service*, *clerk service*, *liveness service*, and *health service*. For each service, the client is either a front-end or one of the above four processes, while the server is either the GSM or one of the above four processes. Figure 3 gives a diagrammatic overview of the services and their API functions, and Figure 4 gives further details.

Niobe entities (front-ends, replicas, and the GSM) communicate with each other via *messages*. Each message is either a *request* or *response*, and has a *type* that is one of the eight API functions listed in Figure 4. It is essential that the data service places `Update` and `Read` requests in a single FIFO queue, processing them in the order received. Consecutive `Read` requests may be processed in parallel, but `Update`s must be processed in isolation: any `Read`s received before a given `Update` $u$ must be completed or aborted before $u$ is processed, and any `Read`s received after $u$ may be processed only after $u$ is completed or aborted. This behavior can be implemented using a standard reader-writer lock. (In fact, these serialization requirements can be substantially relaxed: requests that affect disjoint parts of the data item can be reordered, but a detailed discussion of this would take us too far afield here.)

## 4. NIOBE ALGORITHMS

This section gives complete descriptions of all algorithms used in the Niobe protocol. The appendix gives additional formal details, including pseudocode.

[3]Of course, these do not have to be conventional processes; in our deployment, they are separate threads executing in a single address space.

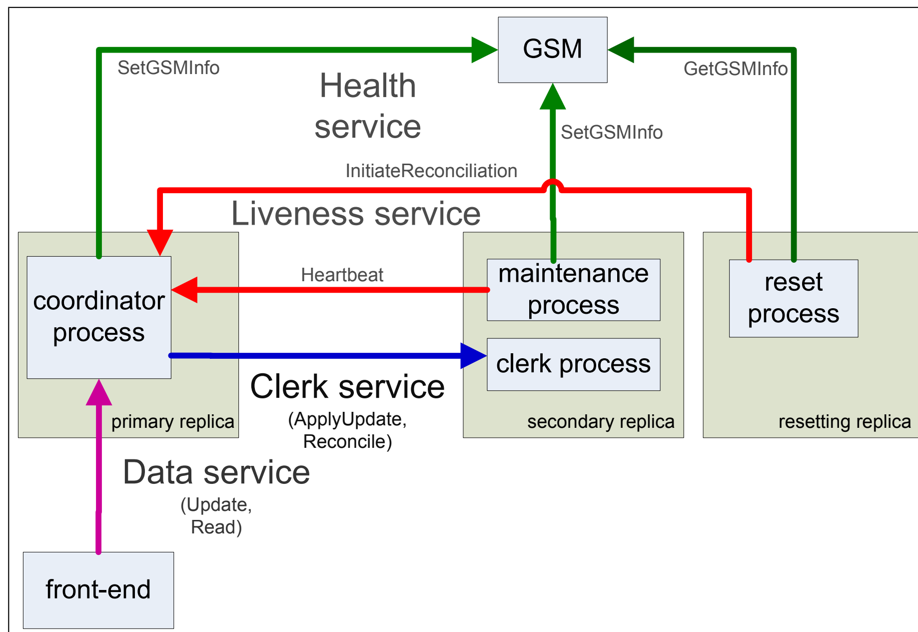


Fig. 3. The four services of the Niobe protocol: data, clerk, liveness and health.

| **Service name and API** | **Clients** | **Servers** | **Primary purpose** |
|---|---|---|---|
| Data service (Update, Read) | Front-ends | Coordinators | Read and write data for front-ends |
| Clerk service (ApplyUpdate, Reconcile) | Coordinators | Clerks | Persist or reconcile local copy of data for coordinator |
| Liveness service (InitiateReconciliation, Heartbeat) | Maintenance | Coordinator | Initiate recovery of a dead secondary; respond to heartbeats |
| Health service (SetGSMInfo, GetGSMInfo) | Maintenance, Coordinators, Reset | GSM | Update and/or disseminate replica health information |

Fig. 4. Descriptions of Niobe services.

## 4.1 Resetting

A Niobe replica begins operations, or restarts after encountering a problem, by executing an algorithm called `Reset`, which uses only the contents of stable storage. Replica $R$ performs a `Reset` by first canceling any pending requests (the requests are simply eliminated – no responses need be sent), and doing local recovery to ensure that any partially-complete updates to its local data value are rolled back.

Then $R$ contacts the GSM to find out the current state of the Niobe state machine. From this point, there are three possibilities: $R$ could be a live primary, a live secondary, or a dead secondary.

If $R$ is a live primary, it first *reconciles* with all secondaries it believes are alive. The objective of reconciliation is to ensure that every live secondary agrees with $R$'s current local data value $D$. This can be achieved by an extremely simple approach, such as sending $D$ to each live secondary, or by using a more sophisticated rsync-like reconciliation protocol that transmits only the differences between $D$ and a secondary's data value. In any case, the outcome is eventually that each live secondary has stably stored $D$ as its local data value. Any secondary that fails to achieve this in a timely fashion gets killed: $R$ passes a GSM decree transitioning the unresponsive secondary into the Dead state. Once reconciliation is complete, $R$ begins its normal operations as a primary: it responds to `Update` and `Read` requests from front-ends, and to heartbeat messages from secondaries.

If $R$ is a live secondary, it listens for `ApplyUpdate` requests from the primary, and also periodically sends heartbeat messages to the primary. Heartbeats are tagged with increasing sequence numbers, and at most one can be outstanding: $R$ never sends a new heartbeat until the primary has acknowledged the previous one (retries with the same sequence number are permitted, however). These heartbeats, which are sent at least every HeartbeatPeriod, serve a dual purpose in the Niobe protocol. First, each heartbeat implicitly grants a *read lease* to the primary, permitting the primary to perform `Read` operations on its local copy of the data without verifying that the secondary $R$ agrees – see the read algorithm described in Section 4.3 for details on this. The second purpose of the heartbeats is to maintain liveness in the face of a failed primary: if the primary becomes unresponsive and fails to respond to a heartbeat, $R$ will attempt to kill it and take over as primary itself. The details of this *takeover* procedure are crucial for maintaining data consistency, and are described below in Section 4.5.

If $R$ is a dead secondary, it requests reconciliation from the current primary. This results in the primary first suspending the processing of update requests, then sending its latest data value to $R$ (with the obvious optimizations of (i) sending only the differences using a variant of the dirty region log, and (ii) delaying the suspension of updates until $R$'s data value is almost perfectly up-to-date). Once $R$ has stably stored the primary's data value, the primary passes a decree with the GSM which reincarnates the dead secondary. The dead secondary $R$ then `Reset`s, and will come back up as a live secondary. Processing of update requests can now resume on the primary; they will now be replicated to the newly reincarnated secondary as well as the other live replicas.

Every message from one replica to another replica or the GSM is stamped with the sender's cached epoch number. If $R$ receives a message whose epoch number is higher than its own, $R$ knows that its GSM information is out of date, and performs a `Reset`. On the other hand, if $R$ or the GSM receives a message whose epoch number is lower than its own, it responds with a special BadEpoch value which will cause the sender to `Reset`.

### 4.2 Updates

Suppose a replica $R$ receives an `Update` request from a front-end, requesting that some mutation `M` is applied to the data value. $R$ first checks that it believes itself to be the primary. If not, $R$ responds with a hint suggesting the correct identity of the primary. Otherwise, $R$ acquires a write lock, guaranteeing that no other `Read` or `Update` requests are being processed, then enters the core of the update algorithm. In parallel, $R$ sends an `ApplyUpdate(M)` request to the clerk service on every replica (including itself) that it believes to be alive. Secondaries should check that they agree with $R$'s epoch number (and `Reset` themselves if not), then apply the mutation to their stable storage and return a success code to $R$.

Any secondary that fails to respond successfully gets killed: $R$ passes a GSM decree transitioning the unresponsive secondary into the Dead state. As with the reconciliation and heartbeat messages, `ApplyUpdate` requests are tagged with $R$'s cached epoch number, and if $R$ receives a BadEpoch response from any of these requests or from a GSM decree, it `Reset`s itself.

If $R$, the primary, cannot get a timely and successful response from its own clerk service, $R$ does not attempt to kill itself via a GSM decree. Rather, it stops responding to heartbeats. This has the effect that one of the live secondaries will notice a missing heartbeat response, and execute the takeover algorithm of Section 4.5. After waiting for a suitable period, $R$ then `Reset`s.

Once every live replica has performed the update successfully (or alternatively, been successfully killed via a GSM decree), $R$ can respond to the front-end. In the normal case, $R$ will return Success, indicating that the mutation has been applied to at least $Q$ replicas, where $Q \geq 1$ is a configuration parameter set by the system administrator, expressing the minimum number of replicas on which a data item must be stored. If there are excessive simultaneous failures and the mutation was applied to fewer than $Q$ replicas, $R$ returns Failure.

Thus, the following invariant is maintained: a completed update operation, whether successful or not, was applied successfully to the stable storage of every replica believed to be alive by the primary at the end of the operation.

### 4.3 Reads

If a replica $R$ receives a `Read` request from a front-end, it first acquires a lock for reading, guaranteeing that no `Update` requests are being processed. (Other `Read` requests may be processed in parallel, however.) Then, $R$ checks that it believes itself to be the primary, and if not, replies with an appropriate hint. Otherwise, it checks that it holds a read lease from every secondary $S$ it believes is alive, as follows. The leasing mechanism employs a parameter GracePeriod, which is the maximum amount of network or other delay that is permitted before we consider a fault has occurred. (Note that the value chosen for GracePeriod affects performance, but not correctness.) We define the *lease period* $L$ by

$$L = \mathsf{HeartbeatPeriod} + \mathsf{GracePeriod}. \tag{1}$$

If the primary $R$ has received a heartbeat from secondary $S$ within the last lease period, its lease from $S$ is still valid. If $R$ has a valid lease from every secondary it believes to be alive, $R$ simply reads its local data value from stable storage and

returns it to the front-end.

On the other hand, suppose the read lease from some live secondary $S$ has expired. Then $R$ attempts to kill $S$, by passing a GSM decree. Obviously, if the decree fails with a BadEpoch response, $R$ `Reset`s itself. Otherwise, $R$ succeeds in killing all secondaries whose leases have expired, and it can return its local data value to the front-end.

Note that the following invariant therefore holds for successful read operations: for every secondary $S$ that the primary believes to be alive at the end of the operation, the primary held a valid read lease from $S$ at some instant during the operation.

### 4.4 Heartbeat processing at the primary

Recall that heartbeat requests are tagged with increasing sequence numbers by the secondary that sends them, so a primary $R$ that receives a heartbeat request from some secondary $S$ first checks this sequence number and rejects the request if it is older than a previously-received request. Otherwise, $R$ checks to see if the time elapsed since the last heartbeat was received is less than the lease period $L$ (Equation 1). If so, the heartbeat is on time and $R$ responds with a success code to $S$. Otherwise, the heartbeat has arrived too late and $R$ attempts to kill $S$ by passing a GSM decree. As usual, $R$ `Reset`s itself if the decree fails with a BadEpoch.

### 4.5 Takeover

As described in Section 4.1, a live secondary $S$ ensures that it sends a heartbeat request to its primary every HeartbeatPeriod. If the primary does not send a timely and successful response, $S$ attempts to kill the existing primary and take over as the new primary. This "takeover" algorithm employs a parameter MaxClockDrift, which expresses the maximum factor by which two clocks in the system can disagree, when measuring an interval of real time. The key step in the takeover algorithm that guarantees correctness is that $S$ waits for the *lease expiry time* $\Delta$ before attempting to kill the primary, where the lease expiry time is given by

$$\Delta = 2 \times L \times \mathsf{MaxClockDrift}. \quad (2)$$

After waiting for time $\Delta$, $S$ submits a GSM decree killing the current primary and appointing itself as the new primary. Whether or not this decree succeeds, $S$ then `Reset`s itself.

The appendix shows rigorously how the value of $\Delta$ in (2) leads to consistent behavior; we summarize the argument here. Recall that there is only ever one heartbeat outstanding between $S$ and the primary, and heartbeats employ sequence numbers to prevent out-of-order delivery. So if $S$ sent its most recent heartbeat $h$ at time $t$, no earlier heartbeat from $S$ is accepted at the primary after $t$. There are two cases: (i) $h$ arrives before $t+L$, and (ii) $h$ arrives after $t+L$. In case (i), we know the primary receives no heartbeat messages in the interval $[t+L, t+2L]$. In case (ii), the primary receives no heartbeat messages in the interval $[t, t+L]$. Thus, by waiting for at least two lease periods (1) without sending a heartbeat request, $S$ guarantees that the primary experiences at least *one* lease period without receiving a heartbeat from $S$. The primary, having experienced a whole lease period without a heartbeat from $S$, will never respond to a `Read` request without first successfully killing $S$ –

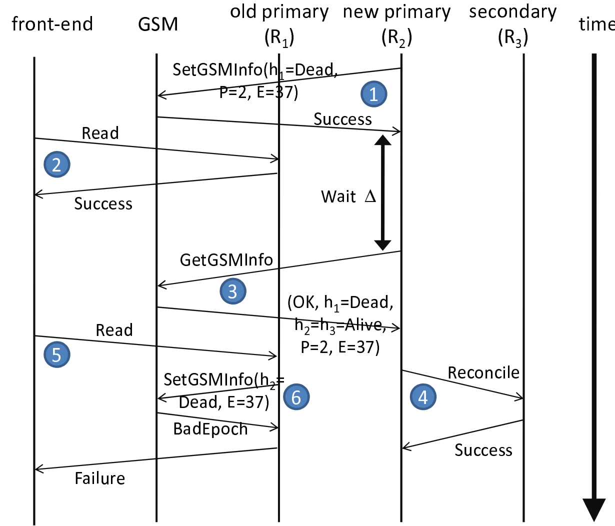


Fig. 5. **A possible takeover sequence.** Communication between $R_1$ (the current primary) and $R_2$ (a secondary) has broken down. (1) $R_2$ notices a missing heartbeat response from the primary and tries to kill it via a GSM decree, which succeeds. (2) Meanwhile, the old primary $R_1$ can continue serving reads from the front-ends until its read leases expire. (3) After waiting the lease expiry time $\Delta$, $R_2$ can `Reset`, which begins by getting the latest GSM information. (4) The new primary $R_2$ must reconcile with any live secondaries (in this case $R_3$) before beginning to serve read and update requests. (5) Meanwhile, the old primary's read lease from $R_2$ has expired, so if the old primary receives a read request, it (6) tries to kill $R_2$ – in this case the attempt fails because $R_1$'s epoch number is out of date.

either during the processing of a heartbeat (Section 4.4) or read (Section 4.3). Of course this discussion assumed that clocks on the primary and secondary advance at the same rate; the extra factor of MaxClockDrift in (2) allows for differing clock rates.

Figure 5 shows a possible sequence of events involving the takeover algorithm.

## 5. PERFORMANCE PROPERTIES

This section analyzes the cost of Niobe operations under normal, failure-free conditions. We measure two types of cost, *resource* cost and *latency* cost, and conclude that Niobe's costs are optimal in several respects according to some simple definitions of these costs. Caching effects are ignored for this worst-case analysis.

Latency is measured in terms of *message delays* and *storage delays*. A message delay is the time for a message to be transmitted from one Niobe replica to another. A storage delay is the time for a replica to read or update its data item. We define the latency of an operation to be the number of delays of each type that must occur between the arrival of the front-end's request and the departure of the reply to the front-end.

Resource cost is measured in terms of the number of messages, and the number of stable storage accesses. We define the resource cost of an operation to be the number of messages and storage accesses that must occur between the arrival of

the front-end's request and the departure of the reply to the front-end.

First consider update operations. Under failure-free conditions, there is one message delay for `ApplyUpdate` requests from the primary to the secondaries, one storage delay for the secondaries to apply the update, and one message delay for the secondaries to send an `ApplyUpdate` response to the primary. So the total latency cost of an update is two message delays and one storage delay. The resource cost is easily seen to be $2(N-1)$ messages and $N$ storage accesses, when there are $N$ replicas alive.

Read operations are even easier to analyze, since under failure-free conditions, the primary simply reads its local data value and returns it to the front-end. The latency cost is zero message delays and one storage delay; the resource cost is zero messages and one storage access.

Clearly, the simplistic cost modeling above is not appropriate for all purposes. One obvious improvement would be to acknowledge that messages transmitting data (as opposed to just acknowledgments) are far more costly if the data size is large. In this case one could distinguish between "data messages" and "regular messages". The latency cost of an update would then be one data message delay, one regular message delay and one storage delay; its resource cost would be $N-1$ data messages, $N-1$ regular messages, and $N$ storage accesses. For reads, the costs would be unchanged.

We claim the above costs are optimal for reads, and optimal in terms of storage delays and accesses for updates: this claim follows trivially from the fact that the cost is exactly 1 storage delay or access in each case, and any other protocol must also access the stable storage before it can return or update the data value. On the other hand, Niobe is not optimal in terms of resource cost. For example, Chain Replication [van Renesse and Schneider 2004] achieves resource cost of $N-1$ messages and $N$ storage accesses. However, the storage accesses are serialized in Chain Replication, resulting in a latency of $N$ storage delays.

## 6. CONSISTENCY GUARANTEES

The software engineers who maintain a replicated storage service, and those who write client software for it, are unlikely to be experts on the theory of distributed algorithms. But it is precisely these engineers who must understand the properties of the system so that they can use it correctly. Therefore, this section attempts to give accurate but comprehensible guarantees on Niobe's consistency properties, without relying on the subtle distinctions between such notions as linearizability [Herlihy and Wing 1990], sequential consistency [Lamport 1979], or one-copy serializability [Papadimitriou 1979]. Section 6.1 explains a simple analogy between Niobe's properties and those of a hard drive. Section 6.2 gives mathematically rigorous statements of Niobe's properties, using language that we hope offers useful insights to systems researchers and practioners. Section 6.3 compares with the established notions of linearizability and strict linearizability. Section 6.4 describes our efforts to verify the correctness of Niobe using the TLA+ specification language. A complete description of the formal consistency guarantees is given in the appendix.

### 6.1 "Just like a disk" consistency

The simplest way of describing Niobe's consistency properties is that the system behaves "just like a disk": requests appear to be processed in the real-time order in which they are received by the system. In particular, a read request reflects the effects of all update requests received earlier. This guarantee applies even to update requests that either (i) returned Failure, or (ii) did not return a value (i.e. are still pending). In both cases, one may assume the failed or pending update request either had the requested effect *in its place in the real-time ordering*, or had no effect at all. The only way to determine which of these possibilities actually occurred is to issue a read request after the failed or pending update. Alternatively, one can eliminate the uncertainty by issuing another update request that, if successful, will overwrite the uncertain existing data value. Note that this behavior is indeed "just like a disk": when a disk write returns a disk error, the only way to be sure about the disk contents is to read it or overwrite it.

### 6.2 Formal Mathematical Guarantees

An *operation* $o$ is a request-response pair of messages $(o_{\mathrm{req}}, o_{\mathrm{resp}})$, resulting from a front-end's `Update` or `Read` request.[4] Each of these messages arrives at, or leaves, some Niobe replica machine at a particular instant in time. Let $\tau(o_{\mathrm{req}})$ be the wall-clock time at which $o_{\mathrm{req}}$ is received by its destination, and $\tau(o_{\mathrm{resp}})$ be the time at which $o_{\mathrm{resp}}$ is sent by its source.

`Update` and `Read` responses include a special Success code if they completed successfully. Successful `Read` responses also include the data value that was read, of course. For a completed `Update` operation that requested a mutation `M`, we define the *post-value* of the update to be the local data of the primary that coordinated the update, after the mutation `M` was applied.

Equipped with these definitions, we can now state the first important consistency property of Niobe, termed *causality*: read requests only return values that were previously written, and were not subsequently overwritten. Figure 6 depicts the statement of the theorem graphically.

THEOREM 6.1. *(Causality property) "Every value read was written, but not overwritten." Suppose the system's history includes a read operation* $r = (r_{req}, r_{resp})$ *that returns* (Success, $D$). *Then there exists an update operation* $\tilde{u}$ *that "wrote"* $D$*, and there does not exist an update operation* $u'$ *that "overwrote"* $D$*. Formally, there exists* $\tilde{u} = (\tilde{u}_{req}, \tilde{u}_{resp})$ *such that*

—*the post-value of* $\tilde{u}$ *is* $D$

—*the write arrived before the read:*

$$\tau(\tilde{u}_{req}) < \tau(r_{req}) \tag{3}$$

—*there does not exist an update operation* $u' = (u'_{req}, u'_{resp})$ *such that*

—$u'$ *returns* Success

—*the post-value of* $u'$ *is* $D'$ *for some* $D' \neq D$

[4] For any request that does not receive a response in a given finite history of the system, we can insert an artificial timeout response at the end of the history, so it is safe to assume that all operations do indeed consist of a pair of messages.

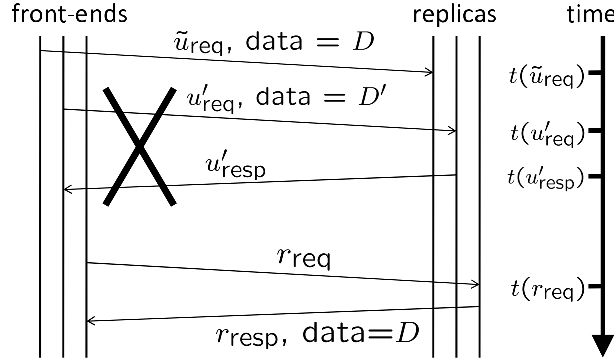


Fig. 6. Theorem 6.1 says that if a request $r$ successfully reads a data value $D$, then some update $\tilde{u}$ must have previously written $D$, and no intervening update $u'$ overwrote $D$ – even if all such requests originate from and arrive at distinct front ends and distinct replicas.

—*$u'$ is received after $\tilde{u}$ and completed before $r$ is received:*

$$\tau(\tilde{u}_{req}) < \tau(u'_{req}) \tag{4}$$

*and*

$$\tau(u'_{resp}) < \tau(r_{req}). \tag{5}$$

The next theorem states a second consistency property of Niobe: written values are *persistent*. Figure 7 depicts this graphically.

THEOREM 6.2. *(Persistence property) "Written values are persistent: two reads return the same value if there are no intervening or pending writes." Let $r = (r_{req}, r_{resp})$ and $r' = (r'_{req}, r'_{resp})$ be two read operations that return* ($\mathsf{Success}, D$) *and* ($\mathsf{Success}, D'$) *respectively. Suppose that $r$ completes before $r'$ begins:*

$$\tau(r_{resp}) < \tau(r'_{req}). \tag{6}$$

*Suppose further that there are no intervening writes: there does not exist an update operation $u = (u_{req}, u_{resp})$ with*

$$\tau(r_{req}) < \tau(u_{req}) < \tau(r'_{resp}). \tag{7}$$

*Finally, suppose there are no pending writes when $r$ begins: there does not exist an update operation[5] $u = (u_{req}, u_{resp})$ with*

$$\tau(u_{req}) < \tau(r_{req}) < \tau(u_{resp}). \tag{8}$$

*Then $D = D'$.*

Notice that the "persistent write" property of Theorem 6.2 does not hold when there are pending updates. In principle, a pending update could be applied at any future time, thus destroying any reasonable notion of persistence. Niobe has some much stronger guarantees that eliminate this possibility, which are stated below. Details and rigorous proofs are given in the appendix.

[5]Recall that for any actual, finite history of the system we can append $\mathsf{Failure}$ responses for any unmatched request.

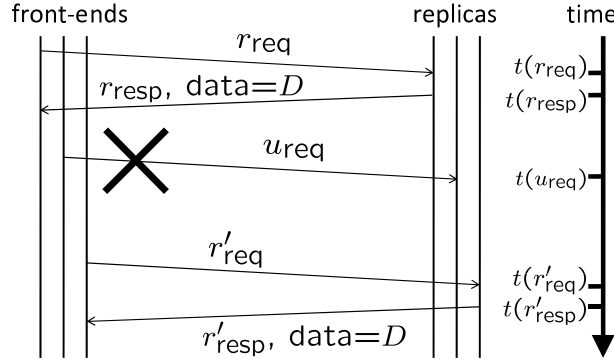


Fig. 7. Theorem 6.2 says that two read requests $r$ and $r'$ read the same value $D$ provided there are no intervening updates $u$ – even if all such requests originate from and arrive at distinct front ends and distinct replicas.

The first guarantee tells us that any *successful* read or update operation permanently cancels any currently pending updates:

THEOREM 6.3. *(Operation Cancellation Effect) The value returned by any read operation received after the completion of a successful read or update operation $o$ is not affected by pending updates $u$ received before $o$: we may assume some response $u_{resp}$ was sent before $o_{req}$ was received.*

The second guarantee tells us that if a resetting primary successfully completes its reconciliation phase, currently pending updates are again permanently canceled:

THEOREM 6.4. *(Reconciliation Cancellation Effect) The value returned by any read operation received after the completion of a primary's reconciliation phase is not affected by pending updates $u$ received before the primary started the `Reset` algorithm: we may assume some response $u_{resp}$ was sent before `Reset` started.*

The proofs of Theorems 6.1-6.4 are given in the appendix, but it is easy to gain intuition for why they hold. The theorems hold trivially during periods when the primary's identity is undisputed, since the primary serializes all reads and updates. The GSM acts like a single, centralized process, ensuring that exactly one secondary takes over as a new primary in any given epoch. Our lease mechanism (Section 4.5) ensures the old primary will cease replying to all requests before the new one starts processing new requests.

Note the guarantees given here all hold in the presence of network partitions. Replicas in the same partition as a majority of GSM servers will continue to function, and any others will be killed off. Progress will cease altogether if no partition contains a GSM majority.

Finally, note that as with any storage system that makes no assumptions about the external network, the guarantees are given in terms of messages entering and leaving the system (in this case, Niobe replica machines). Clients and front-ends must implement reliable FIFO channels between themselves and Niobe machines in order to push guarantees out to the clients. In practice, using TCP or standard sequence number techniques would achieve this.

### 6.3 Comparison with linearizability

*Linearizability* [Herlihy and Wing 1990] is a commonly-accepted notion for consistency in replication protocols. However, as others have pointed out [Aguilera and Frølund 2003], systems that support linearizability can, in principle, allow behavior that may be counter-intuitive to users. Aguilera and Frølund [2003] give a powerful example of this. Consider an Automatic Teller Machine (ATM) that supports linearizability. It might dispense money after checking a customer's account for sufficient funds, but encounter a fault before debiting the account. The account is eventually debited after the fault is repaired, but it could be that the user does not have sufficient funds at that time because of other successful withdrawals in the interim. The ATM's behavior is linearizable, but probably does not accord with the account holder's intuition. A standard solution of this problem is to use database-style transactions, but this is too heavy-weight for a storage service such as Niobe. Instead, Niobe's guarantees resemble *strict linearizability* [Aguilera and Frølund 2003], a stronger form of linearizability that includes the notion of operations aborting before a customer could observe counter-intuitive effects. Theorems 6.3 and 6.4 are the machinery that upgrades Niobe from regular linearizability to a stricter form.

### 6.4 Verification with TLA+

Experience has shown that hand-crafted, conventional mathematical proofs about the properties of distributed algorithms almost always contain errors. Model-checking is a powerful tool that can be used in these situations to increase one's confidence about the properties of a distributed algorithm. Therefore, we specified Niobe in the formal specification language TLA+ [Lamport 2002], and checked some of its properties using the model-checker TLC.

The specification consists of about 400 lines of TLA+, and results in a model with approximately 250 million distinct states, which takes about 2 days to check on a single-CPU machine. As with most specifications, it differs in some respects from the ideal protocol. In our case, the specification does not permit out-of-order message delivery and does not model the reincarnation of secondaries whose data value is out-of-date. The specification does not model real time, so the read lease mechanism is not incorporated – as will be seen below, this has some interesting consequences. In order to restrict the state space to a reasonable size, only $N = 3$ replicas and two distinct data values are permitted.

It was impractical to store a history of all requests in the model, so we could not directly verify Theorems 6.1 and 6.2, since both make statements about the entire history of requests. Nevertheless, the model-checker was able to verify some useful and encouraging invariants. For example, it was proved that the replicas remain in sync:

INVARIANT 6.5. *If there are no updates being processed, the stable storage of every* Alive *replica possesses the same local data value.*

We were also able to prove that updates appear persistent to front-ends which perform their reads at the "true" primary (i.e. the replica which is currently primary according to the GSM):

Invariant 6.6. *If there are no updates being processed, a read at the true primary returns the post-value of the most recent successful update.*

The fact that the read lease mechanism is not modeled means that more general "persistence" invariants do not hold in the specified system. For example, it is not true that all reads return the post-value of the most recent successful update. Consider the following counterexample, which can result in what we call a "stale read". Replica $R_1$ is the primary and $R_2$ a secondary. The value $D$ is written on $R_1$ and $R_2$ because of an update request to $R_1$. Then communication between $R_1$ and $R_2$ breaks down; $R_2$ notices a missing heartbeat, kills $R_1$ and becomes the true primary. The value $D'$ is then written on $R_2$ because of an update request to $R_2$ (it need not replicate $D'$ to $R_1$, as $R_1$ is believed dead). Then a read request arrives at $R_1$, which happily returns the stale value $D$. Of course, this cannot happen in the real Niobe protocol: $R_2$ must wait for a lease to expire before taking over as primary, in which case $R_1$ will notice the expiry of the lease before returning a stale read value.

Despite the fact that the specification could not model leases, it gave us some important guidance on this issue of stale reads, by encouraging us to carefully scrutinize the leasing mechanism and the mathematical arguments for its correctness.

## 7. EXPERIENCE AND RESULTS

Niobe is currently deployed, and serving real customer data. In this section we provide details showing Niobe is efficient and effective in practice. The deployment uses typical server-class rack-mounted machines, connected by a combination of 100 Mb and Gigabit ethernet in a data center. Each machine possesses multiple processors, a modest amount of memory, and several terabytes of attached storage in one or two disk arrays.

Two different hardware configurations are both attractive for our environment. In one, the disk arrays employ RAID-5; in the other, they are configured in JBOD ("just a bunch of disks") mode. Specifically, RAID-5 with two replicas of each data item (2xRAID-5), and JBOD with three replicas of each data item (3xJBOD), both met our requirements in terms of mean time to data loss based on manufacturers' specficiations. 2xRAID-5 is cheaper since it consumes less storage, but its performance can be significantly worse for small writes and during the rebuild of a failed drive. The first set of Niobe users opted for 2xRAID-5, but 3xJBOD deployments are also likely in the future. This is one reason for the requirement that Niobe should be flexible, offering excellent performance and guarantees for $N = 2, 3$ or more.

Because the vast majority of operations can proceed in parallel without any centralized bottleneck, we expected that Niobe exhibit near-perfect scalability. Figure 8 demonstrates this unsurprising, but reassuring, result on our test cluster. The experiment uses a simulated workload whose distribution of data sizes, updates, and reads is identical to one of our major customers. The average data size of read and write requests is 30 kB, but with huge variability from 500 bytes to about 50 MB. This workload was run against Niobe clusters of 4, 8, and 16 back-end machines respectively, using the 2xRAID-5 configuration described above. In particular, there are $N = 2$ replicas for each data item, so each machine acts as the primary for

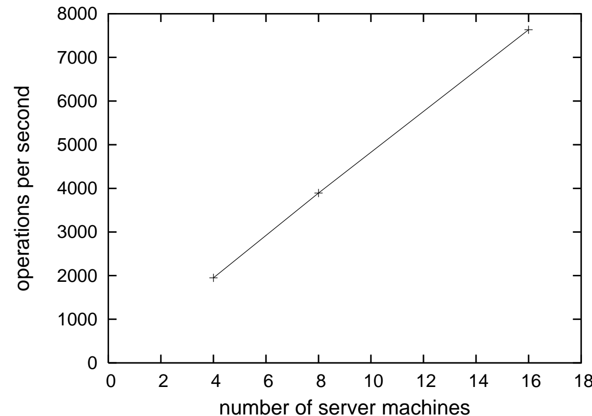


Fig. 8. Throughput scalability of Niobe

approximately half of the data it stores. In each case, sufficient front-end machines are used to saturate the back-ends (it turns out that over 30 front-end machines are required when there are 16 back-ends), and each run takes between 1 and 4 hours to complete. Figure 8 shows the average number of operations per second achieved in each run. It is clear that the throughput scalability is very close to linear.

Figure 9 shows the latencies for read and write operations broken down according to the data size of the operation. The workload and 2xRAID-5 configuration are the same as in the scalability experiment, but now run on a minimal cluster of 2 back-ends. Figure 9 graphs the median latency for operations in a given size range, with error bars representing the 5th and 95th percentiles respectively. There is a painfully obvious performance cliff for writes below 4 kB in size: writes smaller than this take around 200 ms to complete, whereas larger writes can take as little as 40 ms. This is not the well-known "small write" performance issue with RAID-5, as our RAID stripe size is 128 kB. Rather, this cliff may be due to additional read-modify-writes being performed for write requests below the filesystem block size of 4 kB. For both reads and writes, the fixed cost of accessing the disk dominates for data sizes less than about 50 kB, but the cost eventually becomes linear in the data size (at around 100 kB for reads and 2 MB for writes). Large writes are significantly faster than large reads. One possible reason for this is the use of NVRAM cache in the RAID controllers: the reads in this workload are random and derive little benefit from the cache, whereas large writes can safely return as soon as the data is stored in NVRAM.

Figure 10 shows the performance of the same two-machine system over time, as it undergoes a machine failure and subsequent recovery. At minute 16, one of the machines is brought down, and all affected slices (i.e. all slices in this two-machine system) must fail over, either killing off an unresponsive secondary or killing off an unresponsive primary and promoting the secondary to primary. For technical reasons that are irrelevant to the present discussion, the deployment currently enforces a 2.5-minute delay before failover, so single-replica (or "degraded") operations begin in minute 19, with throughput quickly climbing to about 10% less than before

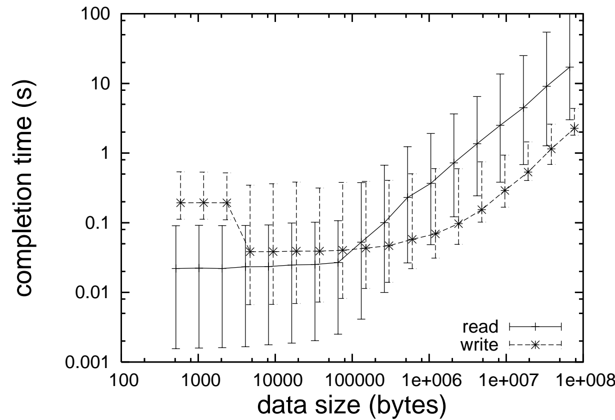


Fig. 9. Niobe latencies

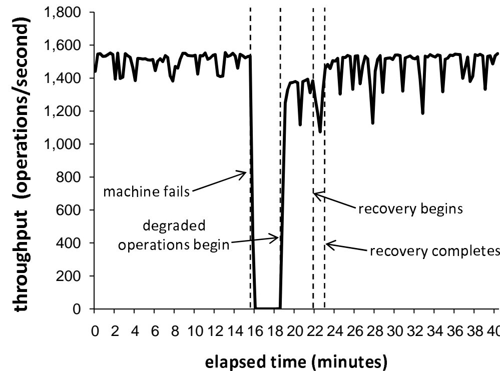


Fig. 10. Throughput during failure and recovery.

the failure. We expect some drop in performance during degraded operation, since the surviving machine is now acting as primary for all of its slices, and primaries do more work than secondaries. (Primaries serve read and write requests, whereas secondaries are impacted only by writes.) At minute 22, the failed machine is brought back online. Normal operations continue during recovery, approximately 20% slower. The system returns to the original level of throughput at minute 23, when recovery completes.

## 8. COMPARISON WITH EXISTING PROTOCOLS

Many of the techniques used in Niobe are adapted from previous work. For example, Niobe uses the “primary-backup” or “primary copy” paradigm for fault-tolerance (in which a primary receives an update from a customer, sends it to one or more secondaries, and then replies to the customer). Primary-backup techniques have been studied since at least the 1970s [Alsberg and Day 1976], are employed in numerous commercial products, and many variants have been proposed and analyzed (e.g. [Budhiraja et al. 1993]). The use of the GSM for ensuring the existence of

exactly one legal primary at all times is strongly reminiscent of the "witness" used by HARP [Liskov et al. 1991] the "auxiliary processor(s)" in Cheap Paxos [Lamport and Massa 2004], the "global state module" in Petal [Lee and Thekkath 1996], the "master" in Chain Replication [van Renesse and Schneider 2004], and has similarities to the Chubby lock service as used in several of Google's services [Burrows 2006]. GSM epochs are closely related to the "viewstamps" in Viewstamped Replication [Oki and Liskov 1988], and the "sessions" in Dynamic Voting [Dolev et al. 1997]. (Dynamic Voting is particularly relevant because the replication scheme in Federated Arrays of Bricks (FAB) [Saito et al. 2004]—a relatively recent system with goals similar to Niobe—is based on it.) Further, HARP and Chain Replication employ a read lease scheme similar to Niobe's, permitting the primary to perform most reads without contacting any secondary. Finally, our technique of using the GSM to kill and reincarnate secondaries, while using all live replicas (but not the GSM) for performing updates, is mathematically equivalent to the "wheel" quorum system studied extensively in the quorum system literature (e.g. [Hassin and Peleg 2006]).

So what is new and interesting about Niobe? In a nutshell, we believe Niobe is more *practical* for large-scale data storage than previously-published replication protocols that we are aware of. Our definition of "practical" comprises: simplicity, flexibility, consistency and performance. We address each of these in turn.

**Simplicity:** Because Niobe is deployed commercially, we know that software engineers and data center operators can implement, maintain, and write client software for a Niobe-based system without any specialized knowledge of distributed systems algorithms. This contrasts with many published replication protocols which may appear simple, but in fact require one or more building blocks (for example, distributed leadership election, reconciliation, or reconfiguration algorithms) that are not completely specified, and in themselves require considerable distributed systems experience for correct implementation or maintenance. Niobe avoids this by using a centralized GSM to store and update the system configuration. Since the GSM need not itself be extremely efficient or scalable (it is only accessed when a component fails), we regard this as a previously-solved problem. One can use either a commercial database, or a distributed consensus algorithm such as Paxos running on a *fixed* set of nodes (dynamic quorums are not required). Chain Replication [van Renesse and Schneider 2004] also uses a GSM-like entity, and is arguably even simpler than Niobe: the chained replica structure results in simple, elegant reconfiguration algorithms with a clear proof of strong consistency. However, the significant extra latency of the chain structure for large writes precludes the use of this design for our application.

**Flexibility:** Reliability is important, but so is continuous service. Storage system administrators should have the flexibility to determine what is most important. Niobe delivers this flexibility, since the administrator can set both $N$, the number of replicas on which data is stored during failure-free operation, and $Q$, the minimum number of replicas on which data may be stored, even in the face of failures. This covers a particularly important special case which has perhaps been given insufficient attention in the literature: in some cases, cost is of such critical importance that $N = 2$ is the best choice. But in this case, many previously-published

protocols will prevent any updates during a failure, since with only one remaining replica, it is not possible to update a majority of the replicas. (It appears that viewstamped replication and FAB both suffer from this problem, for example.) In contrast, if the system administrators value continuous service highly enough that the risk of single-replica updates is worth it, they can set $Q = 1$, and Niobe can happily proceed to perform updates on the one remaining replica during a failure.

**Consistency:** Niobe has simply-stated consistency properties (Theorems 6.1 and 6.2) which have been proved correct, and some additional properties have been verified using formal model-checking tools. In the notoriously buggy world of distributed algorithms, such correctness arguments constitute an important feature of a protocol. FAB, Chain Replication, and Viewstamped Replication are examples of other protocols that also provide clear statements and proofs of their consistency guarantees, so Niobe is by no means unique in this respect.

**Performance:** As argued in Section 5, the cost of Niobe operations in the absence of failures is optimal in certain important respects. This optimality is clearly irrelevant in many cases: if the data size dominates other messages, then extra rounds of messages and/or piggybacking techniques result in indistinguishable performance for non-optimal protocols. For example, it was recently shown that voting-based reads, which require accessing a majority of replicas rather than just the primary, can be efficient in practice despite the extra messages required [Saito et al. 2004]. However, there are situations in which the optimality of Niobe makes an important difference, especially when the data items being stored are small. The performance of protocols which use two-phase commit for updates, such as viewstamped replication and HARP, will suffer in such cases; the same applies to protocols which contact more than one replica for reads, such as FAB. Chain Replication, as already mentioned, suffers a significant latency penalty for large writes, since each replica in the chain must complete its write before forwarding the request to the next replica.

It should be emphasized that some of the simplicity and flexibility of Niobe is due to the deliberately limited read/update interface offered to the customers. Many other replication protocols are more powerful: for example, viewstamped replication and all commercial replicated databases permit arbitrary transactions on multiple data items. On the other hand, designers of replicated file systems have sometimes opted for weaker consistency than Niobe (e.g. the Google file system [Ghemawat et al. 2003], Bayou [Petersen et al. 1997], Coda [Kistler and Satyanarayanan 1992]), which is not appropriate for our customers. The previous work most similar to Niobe is probably Chain Replication, which showed that a storage service can provide high throughput and availability without sacrificing strong consistency. Niobe shows that one can in addition achieve low latency, without sacrificing on throughput, availability, or consistency; and Niobe further offers empirical proof that the protocol is practical at scale.

As already stated, we believe the primary contribution of Niobe lies in tastefully combining previously-published techniques to produce an extremely practical protocol. However, another possible source of novelty is the first detailed description of a decentralized read lease mechanism with multiple replicas. The Google Accounts service [Perl and Seltzer 2006] appears to use a similar approach, but does not spec-

ify the duration of the leases required for correct operation, nor provide a proof of this correctness. Another difference between Niobe and the Google Accounts service is that Google Accounts relies on communication with a majority of replicas, rather than the more flexible possibility of killing all unresponsive secondaries. The basic technique of decentralized read leases is perhaps also implicit in HARP, but the HARP paper describes the details of read leases only in the case of a single secondary. And this case is degenerate: because there is only a single outstanding lease, it is equivalent to the well-known centralized lease scheme of Gray and Cheriton [1989]. In contrast, Niobe uses the fact that every live secondary grants its own lease to the primary, which performs a read only if it succeeds in killing any live secondary whose lease has expired. There is no need for a central lease server or lock server.

## 9. CONCLUSION

Niobe is a protocol and system for building large-scale replicated storage services in practice. It follows the primary-backup paradigm, and separates reconfiguration activity into a separate, highly-reliable Global State Manager module. Niobe has simply-stated, rigorously-proven strong consistency properties which are of practical use to its clients. Reads and writes have optimal storage latency in theory, and excellent performance in practice. Niobe is currently deployed as the backend of a commercial Internet service. We hope it can serve as a useful model for building storage services in the future.

## APPENDIX

## A. CLIENT-SIDE ALGORITHM DESCRIPTIONS

The next four subsections describe the API functions of the four Niobe services from the point of view of their clients. A following section (Appendix B) describes the algorithms used by the servers to implement the API functions. The services and API functions described here are summarized in Figures 3 and 4.

The coordinator process employs a local, in-memory reader-writer lock (`ReadWriteLock`), which will permit multiple parallel reads when held for reading, but guarantees that when held for writing, only a single write and no reads are in-flight. Recall that the coordinator maintains a FIFO queue of `Read` and `Update` requests, and it is essential that `ReadWriteLock` is acquired on behalf of these requests in this FIFO order. Finally, the coordinator also maintains maintains a local binary variable in memory: `Fitness`, whose value can be either Good or Bad.

### A.1 Data service

A coordinator process exports 2 data service API functions to the customers: `Update` and `Read`.

*`Update(Mutation M)`.* The customer calls this function to update the data value according to some mutation `M`. The mutation can be any operation that can be implemented atomically by a replica on its stable storage. Some obvious possibilities are: (i) overwriting the data value, (ii) appending to the data value, or (iii) performing a compare-and-swap on the data value. A Niobe implementation could permit one or more of these possibilities.

`Update`s can be sent to any replica, but they will fail unless sent to a replica that believes it is the primary. Customers can cache their own belief about the primary's identity, and update it using the hint supplied whenever their belief is wrong (see below).

The `Update` operation can return one of the 4 values Success, Failure, Invalid, or (NotPrimary, hint $= r$). The interpretation of these return values is as follows:

—Success: the update operation was successfully applied to the Niobe state machine. Formal guarantees will be given later, but informally, this means that the update reached stable storage on all Alive replicas, and that the number of such replicas is at least the minimum update replication factor $Q$.

—Failure: the update operation may or may not have been applied to the Niobe state machine, but consistency is nevertheless maintained. This "consistency" is most easily explained as being "just like a disk" (see Section 6.1); a formal definition and proofs are given in Appendix D.

—Invalid: the update operation was not permissible, and therefore wasn't applied. For example, perhaps a compare-and-swap operation returned `false` from its "compare" phase.

—(NotPrimary, hint $= r$): the replica to which the update request was sent does not believe it is the primary, and suggests trying replica $r$ instead.

Customers can treat a timeout the same as a Failure, except that they might also consider retrying at a different replica.

*Read().* The customer calls this function to read the current data value $D$. As with `Update`, the call should be sent to the primary. The possible return values are (Success, Dval), Failure, or (NotPrimary, hint $= r$). Informally, Success means that Dval is equal to the latest value of $D$ that was written; a formal definition is given Appendix D. Failure gives the client no information; it can only retry, perhaps at a different replica. NotPrimary has the same interpretation as for `Update`.

### A.2 Clerk service

The clerk process on each replica exports 2 clerk service API functions to the coordinator processes: `ApplyUpdate`, and `Reconcile`. A coordinator process will only issue these requests if it resides on a replica that believes it's the primary.

*ApplyUpdate(Epoch e, Mutation M).* A coordinator process on a primary replica sends this request to the clerk process on any replica (let's call it the "target replica") in order to apply the mutation `M` to the target replica's local data value. The possible return values are Success, Failure, and BadEpoch. Success means that the target replica has applied to the requested mutation and stored its result in stable storage. Failure means that the mutation may or may not have been applied (but recall that all replicas use standard techniques to make their mutations atomic, so the target replica's local data value is guaranteed to either be the old value or the correct new value). BadEpoch means that the target replica's cached epoch number is greater than the primary's cached epoch number `e`—so the primary should perform a `Reset()`.

*Reconcile(Epoch e, Data d).* A coordinator process on a primary replica sends this request to the clerk process on a secondary replica (let's again call it the "target replica"), in order to overwrite the target replica's local data value with the given data `d`. Note that this operation is an overwrite, not a general mutation: the intention is that primary sends its own local data value as the parameter `d`, so on successful completion, the secondary will be "reconciled" with the primary (and can be brought back to life, if it was Dead). The possible return values are Success, Failure, and BadEpoch; the interpretations of these values are very similar to those in the `ApplyUpdate` function above.

Note that in a practical implementation, the data value might be very large, so for efficiency, the reconciliation process might be achieved via an rsync-like protocol that transmits only the differences between primary and secondary local data values. But the final effect of a more sophisticated reconciliation protocol would be the same as the simple overwrite described above.

### A.3 Liveness service

The coordinator process exports two liveness service API functions: `InitiateReconciliation` (exported to the reset process), and `Heartbeat` (exported to the maintenance process).

*InitiateReconciliation(Epoch e).* The reset process on a replica that believes itself dead sends this request to the primary when it's ready to begin reconciliation. The possible return values are Success, Failure, and BadEpoch, which have very similar interpretations to the two functions above.

*Heartbeat(Epoch e, Counter c).* The maintenance process on a replica that believes itself to be a secondary sends this request to the primary periodically. More precisely, heartbeat messages are sent whenever necessary to ensure that the primary and each secondary never allow more than HeartbeatPeriod to elapse without exchanging a heartbeat (or alternatively, taking some remedial action). The possible return values are Success, Failure, and BadEpoch. Success means that the primary's `Fitness` variable equals Good—so the primary is not aware of any problems with its own operations. Failure means the primary's `Fitness` equals Bad—so the primary has encountered problems and is hoping that another replica will take over as primary. BadEpoch has the same interpretation as above—the requesting replica should `Reset()` itself. A timeout is treated like a Failure.

### A.4 Health service

The GSM exports two health service API functions to the other processes on the replicas: `GetGSMInfo`, and `SetGSMInfo`.

*GetGSMInfo().* The maintenance process on a replica sends this request to the GSM to update the replica's cached copy of the epoch number, health bits and primary ID. The return values are either (Success, `epoch` $= e$, `health` $= [h_1, h_2, \ldots, h_n]$, `primaryID` $= p$) or Failure. Both are self-explanatory.

*SetGSMInfo(newEpoch, newHealth, newPrimaryID).* The maintenance process on a replica (let's call it the "source replica") sends this request to achieve one of the permitted transitions in the Niobe state machine by altering the GSM's epoch num-

ber, health bits, and primary ID. Specifically, two of the three types of transitions listed in Section 3.1 can be achieved this way: (i) kill or reincarnate a secondary, (ii) reassign the primary. Naturally, the proposed new values must conform to the rules given in Section 3.1. In other words, `newEpoch` must be exactly 1 more than the source replica's cached epoch number, and exactly one of `newHealth` and `newPrimaryID` can differ from the source's cached copy: if the primary ID differs, it must be set to the ID of a live replica, whereas if the health bits differ they must do so by flipping only a single non-primary bit.

The possible return values of `SetGSMInfo` are Success, Failure, and BadEpoch. Success means that the proposed `newEpoch` was indeed one more than the GSM's epoch number, so the proposal was deemed legitimate and the proposed new values have been recorded on the GSM's stable storage.[6] Failure gives the source replica no information; it should retry. BadEpoch means the proposed value of `newEpoch` was illegitimate; the source replica should perform a `Reset()`.

## B. NIOBE'S SERVER-SIDE ALGORITHMS

The previous section described the four Niobe services and their APIs, as summarized in Figures 3 and 4. This section completes the description of Niobe by detailing the algorithms used to perform each API function in each service. We'll need some more notation for this. Denote the cached epoch number, health bits and primary ID on replica $R_k$ by $E_k$, $H_k$, and $P_k$ respectively. The local data value on $R_k$ is $D_k$. The individual health bits cached on $R_k$ are $H_k = (h_{k,1}, h_{k,2}, \dots, h_{k,n})$.

The following four subroutines will be useful for describing the algorithms. Each description assumes replica $R_k$ is executing the subroutine.

*`CheckSelfIsPrimary()`.* If $P_k \neq k$, return (NotPrimary, hint $= P_k$), otherwise return Success.

*`SynchronizeEpoch(Epoch e)`.* While $E_k \neq \mathsf{e}$,

—If $E_k > \mathsf{e}$, return BadEpoch. If $E_k < \mathsf{e}$, execute the `Reset` algorithm of Appendix C.

Once $E_k = \mathsf{e}$, return Success.

*`CheckPrimaryLease()`.* Recall from Section 4.5 that the role of primary requires a lease-like mechanism to prevent customers reading stale values. Informally, every secondary periodically grants a lease to the current primary, which gives that primary permission to read and return data values to customers. If the lease issued by a secondary has expired, then the primary must kill off that secondary before it can return a data value to a customer.

The leasing mechanism is implemented as follows. The coordinator process on replica $R_k$ should keep, in memory, a record of the (local) time LastHeartbeat$[j]$ at

[6] Note that for debugging and sanity purposes, the GSM would probably check the legitimacy of the new health bits and primary ID, but this is not necessary for correctness of the Niobe protocol: we assume that replicas are neither malicious nor Byzantine, so they follow the protocol. In particular, if a replica proposes a legitimate new epoch number, its other proposed values are also guaranteed to constitute a legitimate transition.

which it received the most recent communication (a `Heartbeat`, or any other request) from each secondary $R_j, j \neq k$. It also employs two constants: GracePeriod, and HeartbeatPeriod. HeartbeatPeriod is the maximum time a correctly-functioning secondary will permit to elapse without communicating with the primary—as described below, this is achieved by a secondary sending `Heartbeat` requests to the primary whenever necessary. GracePeriod is the maximum amount of network or other delay (such as scheduling delay on the secondary) that is permitted before we consider a fault has occurred.[7]

Here, then, is the `CheckPrimaryLease()` algorithm:

(1) "Kill unresponsive secondaries": Let the current time be $t$. For each $j \neq k$, if $t - \mathsf{LastHeartbeat}[j]$ is larger than HeartbeatPeriod + GracePeriod, then:
   (a) send the GSM a `SetGSMInfo` request killing secondary $R_j$.
   (b) `Reset` if this request returns BadEpoch. If the request fails after sufficient retries, return Failure.

(2) Return Success.

*KillSecondary(Integer j)*

(1) Send a `SetGSMInfo` request to the GSM, setting $h_{k,j}$ = Dead. Perform a `Reset()` on BadEpoch. Retry a suitable number of times on Failure, and then perform a `Reset()`.

(2) If the `SetGSMInfo` request succeeds, set $E_k$ and $h_{k,j}$ to their new values and return Success.

### B.1 Algorithms for the data service

Data service requests are received by a coordinator process on some replica, say $R_k$.

*Update(Mutation M)*

(1) `CheckSelfIsPrimary()`. If this does not succeed, return NotPrimary, together with the appropriate hint, to the customer.

(2) "Fail if we aren't writable": Acquire $R_k$'s local `ReadWriteLock` for writing. Return Failure if this can't be accomplished in a reasonable period.

(3) "Bail if the mutation's invalid": If the mutation `M` cannot validly be applied to $D_k$, return Invalid.

(4) "Give up if our GSM state is stale": If, at any time during the remainder of the algorithm, we receive a BadEpoch response from any `ApplyUpdate` or `SetGSMInfo` request: (i) return Failure, then (ii) perform a `Reset()`.

(5) "Send the mutation to all alive replicas": For each $j$ such that $h_{k,j}$ = Alive, send an `ApplyUpdate`($E_k$, `M`) request to replica $R_j$.

[7] Correctness is not affected by the value of GracePeriod, but liveness and performance are both affected. An excessively short GracePeriod will cause secondaries to be killed off unnecessarily often, reducing the average number of replicas on which data is stored and causing extra GSM traffic and resets. An excessively long GracePeriod will lead to a long delay before the system becomes responsive after a failure.

(6) "Kill off any faulty secondaries": For each $j \neq k$ with $h_{k,j} =$ Alive such that—after a suitable number of retries—the `ApplyUpdate` request to $R_j$ returned Failure (or, equivalently, timed out), execute the `KillSecondary(j)` subroutine.
(7) "Commit suicide if we are faulty": If the `ApplyUpdate` request to $R_k$ (which was sent to the primary's own clerk process) returns Failure or times out after suitable retries, return Failure, and set the local `Fitness` variable to Bad (this has the effect that some secondary will try to take over as primary after the next time it sends $R_k$ a heartbeat).
(8) "Return if all alive replicas committed the mutation": Once it is true that a Success response from every currently alive replica (i.e. every $R_j$ such that $h_{k,j} =$ Alive) has been received, return Success if the number of such replicas was no less than than the minimum update replication factor $Q$, and Failure otherwise. In any case, release `ReadWriteLock`.

*Read()*

(1) Acquire `ReadWriteLock` for reading.
(2) `CheckSelfIsPrimary()`. If this does not succeed, return NotPrimary, together with the appropriate hint, to the customer (after releasing `ReadWriteLock`).
(3) `CheckPrimaryLease()`. If this does not succeed, return Failure to the customer (after releasing `ReadWriteLock`).
(4) "Read the local data value": Read the local data value $D_k$, and release `ReadWriteLock`.
(5) "Return the local data value": Return (Success, $D_k$), if the value of $D_k$ was accessible. If it wasn't (for example, suppose $D_k$ had to be read from a disk and the disk returned a read error after suitable retries), set the local `Fitness` to Bad, and return Failure.

### B.2 Algorithms for the clerk service

Clerk service requests are received by a clerk process on some replica, say $R_k$. This clerk executes requests as follows.

*ApplyUpdate(Epoch e, Mutation M)*

(1) `SynchronizeEpoch(e)`.
(2) Apply mutation `M` to the current local data value $D_k$, resulting in a new local data value recorded in stable storage. Return Success if this was accomplished, otherwise return Failure (this could be necessary if, for example, a disk or other error was encountered). Recall that the clerk uses standard techniques to ensure that the mutation is applied atomically, so that even when it returns Failure, the stably-stored value of $D_k$ is either the old or new value and not some corrupted value.

*Reconcile(Epoch e, Data d)*

(1) `SynchronizeEpoch(e)`
(2) Atomically overwrite, in stable storage, the local data value $D_k$ with `d`. If this is accomplished successfully, return Success, otherwise return Failure.

### B.3 Algorithms for the liveness service

Liveness service requests are sent from the maintenance process on some replica, say $R_j$, to a coordinator process on some replica, say $R_k$. Each coordinator process maintains some local state in memory, for every $j \neq k$: LastHeartbeat$[j]$, which is the (local) time at which the most recent heartbeat from $R_j$ was received in the current epoch; and HeartbeatCounter$[j]$, which is the counter value sent with that most recent heartbeat.

The coordinator executes liveness service requests as follows.

*InitiateReconciliation(Epoch e)*

(1) `SynchronizeEpoch(e)`

(2) For debugging purposes, one could execute `CheckSelfIsPrimary()` here, but it's not necessary for correctness. The client will only send an `InitiateReconciliation` to a replica it believes is the primary. Since the client and server both believe the same thing after `SynchronizeEpoch`, the server must also believe it is the primary, so there is no need to check this. Furthermore, the client requesting reconciliation should be a dead secondary, so $h_{k,j} = h_{j,j} =$ Dead, but there's no need to check for this for the same reason.

(3) For efficiency, one could perform a preliminary reconciliation here before blocking updates during the "real" reconciliation. That prevents a long delay in serving update requests. The preliminary reconciliation is easy to perform: just consecutively issue `Reconcile`$(E_k, D_k)$ requests to $R_j$ until they are observed to complete quickly enough (or some sensible maximum number of attempts is reached).

(4) "Fail if we aren't writable": Acquire $R_k$'s local `ReadWriteLock` for writing. Return Failure if this can't be accomplished in a reasonable period.

(5) "Reconcile with the secondary": Send a `Reconcile`$(E_k, D_k)$ request to $R_j$. If this `Reconcile` returns Failure, return Failure. If it returns BadEpoch, return Failure and perform a `Reset()`. In both cases, release `ReadWriteLock`. Otherwise, `Reconcile` returns Success, so continue.

(6) "Activate the reconciled secondary": Issue `SetGSMInfo` to the GSM, reincarnating the reconciled secondary by setting $h_j =$ Alive. If this results in Failure, retry; if BadEpoch, return Failure and perform a `Reset()`; otherwise the GSM returned Success so do the following:
   (a) Wait for a length of time specified by the parameter TypicalResetDuration, which specifies how long a well-behaved secondary should take to reset itself as an Alive secondary. The reason for this is that if we resume operations too quickly, we may be tempted to kill the resetting secondary (it may fail to send a heartbeat, or fail to respond to an `ApplyUpdate`). While killing such a secondary causes no correctness problems, it does lead to obvious performance problems (the secondary could be trapped in a cycle of continuous death and reincarnation).
   (b) Set $h_{k,j} =$ Alive
   (c) Set LastHeartbeat$[j] =$ current local time $t$.
   (d) Set HeartbeatCounter$[j] = 0$.

(e) Release `ReadWriteLock`.

*`Heartbeat(Epoch e, Counter c)`*

(1) `SynchronizeEpoch(e)`
(2) As above, `CheckSelfIsPrimary()` could be useful for debugging but is not necessary.
(3) If $c <$ HeartbeatCounter$[j]$, abort without sending a response. (The request must be a retry of some heartbeat to which we have already responded, and $R_j$ must have received that response.)
(4) If $c =$ HeartbeatCounter$[j]$, go to step 7. (The request is a retry of some heartbeat to which we have already responded, but $R_j$ might not have received that response.)
(5) Otherwise, we have $c >$ HeartbeatCounter$[j]$. Let the current local time be $t$. If $t -$ LastHeartbeat$[j] >$ HeartbeatPeriod + GracePeriod, send no response and instead execute `KillSecondary`$(j)$. Otherwise, continue.
(6) Set LastHeartbeat$[j] = t$ and HeartbeatCounter$[j] = c$.
(7) If $R_k$'s local `Fitness` is Good, return Success; otherwise return Failure.

### B.4 Algorithms for the health service

The GSM's algorithms for implementing the health service are very obvious. On receiving a `GetGSMInfo` request, the GSM returns its current values for the epoch number, health bits and primary ID. On receiving a `SetGSMInfo` request, the GSM returns BadEpoch if the requester's epoch number is out of date, and otherwise the GSM updates its variables with the submitted values (recording them in stable storage) and returns Success. If it encounters a problem doing so, it returns Failure.

## C. THE RESET ALGORITHM

Apart from the four Niobe services described above, the only activity performed by a Niobe replica is the *reset* algorithm. A reset occurs when a replica

—first begins operation
—restarts after a transient failure
—receives a BadEpoch response
—discovers its epoch number is out of date during `SynchronizeEpoch()`.

The reset algorithm on replica $R_k$ is as follows.

*`Reset()`*

(1) "Cease all activity": kill any currently-running services. (If some requests are currently being processed, we can either wait until they are finished or terminate their processing. In practice, it is probably most graceful to give requests a reasonable time to complete, then terminate any remaining ones.)
(2) "Do local recovery": Perform any local recovery operations to ensure that the stably-stored value $D_k$ is valid. (For example, if the replica was halfway through a long-running update to $D_k$ when it encountered a failure and was rebooted, it now uses standard logging and recovery techniques to revert $D_k$ to its value before the interrupted update.)

(3) "Get GSM information": Send the GSM a `GetGSMInfo` request, and assign the result to $E_k$, $H_k$, $P_k$.

(4) "Launch clerk process": Start the clerk process, which listens for clerk service requests and acts on them.

(5) "Start primary or secondary operations": If we are the primary ($P_k = k$), run the `StartPrimary()` algorithm described below. Otherwise: if we are a live secondary ($P_k \neq k, h_{k,k} = \mathsf{Alive}$) run the `StartAliveSecondary()` algorithm; if we are a dead secondary ($P_k \neq k, h_{k,k} = \mathsf{Dead}$) run the `StartDeadSecondary()` algorithm.

*StartPrimary().* This algorithm consists of the following steps:

(1) Initialize:
   (a) Fitness = Good
   (b) LastHeartbeat[$j$] = current local time $t$ for all $j \neq k$
   (c) HeartbeatCounter[$j$] = 0 for all $j \neq k$

(2) "Reconcile with all alive secondaries": For each $i \neq k$ such that $h_{k,i} = \mathsf{Alive}$, issue a `Reconcile`($E_k, D_k$) request to $R_i$. If any of these requests returns BadEpoch, perform a `Reset()`.

(3) "Kill off any faulty secondaries": For each `Reconcile` request that returns Failure after suitable retries, attempt to kill the offending secondary $R_j$ by executing the `KillSecondary(j)` subroutine (see Appendix B).

(4) "Launch coordinator process": Once it is true that a Success response from every non-local currently alive replica (i.e. every $R_j$ such that $h_{k,j} = \mathsf{Alive}, j \neq k$) has been received, start the coordinator process (which does nothing but listen for and act on data service and liveness service requests).

*StartDeadSecondary().* This algorithm consists of a single step:

(1) "Reconcile with primary and reset": send an `InitiateReconciliation` request to replica $P_k$. If this returns Success, perform a `Reset()`—after resetting we will be Alive and will be able to start normal operations. If the `Initiate-Reconciliation` request returns BadEpoch, or does not return Success after reasonable retries, perform a `Reset()` anyway.

*StartAliveSecondary().* Alive secondaries maintain an in-memory counter called SecondaryHeartbeatCounter.

(1) "Initialize": set SecondaryHeartbeatCounter = 0

(2) "Launch maintenance process": Start the maintenance process, which repeatedly runs the `HeartbeatLoop` described below.

*HeartbeatLoop().* Each secondary must monitor its primary, and attempt to take over if it appears the primary has failed. This is accomplished by repeatedly executing the following algorithm, called `HeartbeatLoop()`. The secondary also employs a parameter MaxClockDrift, which expresses the maximum factor by which two clocks in the system can disagree, when measuring an interval of real time.

The `HeartbeatLoop()` algorithm is as follows:

(1) Sleep for HeartbeatPeriod.

(2) Increment SecondaryHeartbeatCounter.

(3) "Send a heartbeat to the primary". Let the cached epoch be $n$ and SecondaryHeartbeatCounter be $c$. Send a `Heartbeat(n, c)` request to the primary. If the return value is Success, go to step 1. If the request times out, retry a suitable number of times using the same value of $c$. Otherwise the heartbeat request has either failed or timed out too many times, so continue to step 4.

(4) "Wait for the primary's lease to expire". Wait for the period of time $2 \times (\text{HeartbeatPeriod} + \text{GracePeriod}) \times \text{MaxClockDrift}$. After waiting for this period, we are guaranteed the primary will not respond to any further read requests without first killing off any secondaries it considers unresponsive. Hence, we are free to proceed to step 5, in which we will attempt to become the new primary.

(5) "Take over as new primary". Issue a `SetGSMInfo()` request to the GSM, setting $h_{P_k} = \text{Dead}$, $P = k$, and of course incrementing the epoch number. This has the effect of killing the unresponsive primary and assigning replica $R_k$ as the new primary. If the `SetGSMInfo()` returns Success, perform a `Reset()`, which will cause us to read the new GSM state and start acting as a primary. Retry the `SetGSMInfo()` if it fails; `Reset()` if it returns BadEpoch.

This algorithm can be augmented with an obvious optimization: piggyback virtual heartbeats on any other communication between primary and secondary. In this case the secondary maintains a value $V$, which is the local time at which the most recent virtual or real heartbeat was sent. Step 1 then becomes "Sleep for $\text{HeartbeatPeriod} + V - t$", and Step 3 only sends its heartbeat request if $t - V > \text{HeartbeatPeriod}$. (As usual, $t$ here is the current local time.)

## D. PROOFS OF FORMAL CONSISTENCY GUARANTEES

In this section we prove that Niobe possesses useful consistency guarantees.

A read operation is *valid* if it reaches step 4 of the `Read` algorithm of Appendix B.1. Suppose a valid read operation is executing on replica $R$, with cached epoch number $n$. Suppose further that $R$'s local clock has value $t$ when $R$ begins executing step 4 of the `Read` algorithm, and that at this moment $R$'s stable storage contains the data value $D$. We define the *pre-state* of the read operation to be the tuple $(D, R, n, t)$.

Extremely similar definitions apply to update operations. An update operation is *valid* if it reaches step 5 of the `Update` algorithm of Appendix B.1. Suppose a valid update operation is executing on replica $R$, with cached epoch number $n$. Suppose further that $R$'s local clock has value $t$ when $R$ begins executing step 5 of the `Update` algorithm, and that at this moment $R$'s stable storage contains the data value $D$. We define the *pre-state* of the update operation to be the tuple $(D, R, n, t)$.

For any request message $m_{\text{req}}$, let $\tau(m_{\text{req}})$ be the wall-clock time at which $m_{\text{req}}$ arrives at its destination. For any response message $m_{\text{resp}}$, let $\tau(m_{\text{resp}})$ be the wall-clock time at which $m_{\text{resp}}$ leaves its source.

We are ready to prove a crucial property of Niobe: non-overlapping operations can be ordered naturally according to the epochs in which they were executed.

LEMMA D.1. *"If an operation begins after another completes, the earlier operation is executed in an epoch no greater than the later one." Let $o = (o_{req}, o_{resp})$ and $o' = (o'_{req}, o'_{resp})$ be two non-overlapping valid operations, so*

$$\tau(o_{resp}) < \tau(o'_{req}). \tag{9}$$

*Let their pre-states be $(D, R, n, t)$ and $(D', R', n', t')$ respectively. Then $n \leq n'$.*

*Proof.* Suppose $n' < n$ and argue for a contradiction. We first claim that $R \neq R'$: this follows because if $R = R'$, then (9) implies $t < t'$ which contradicts $n' < n$. Hence we know that $R'$ was primary during epoch $n'$, and $R$ was primary during some later epoch $n$. Thus, $R'$ must have been killed by some other replica $R^*$, and this killing must take place before $o_{\text{resp}}$ is sent. (Note that we are sure that $R' \neq R^*$, but it is possible that $R^* = R$.) But when $R^*$ took over as primary, it waited for $R'$'s lease to expire (step 4 of the `HeartbeatLoop` algorithm)—so $R'$ will never again perform a read during epoch $n'$. So operation $o'$ cannot be a `Read` request. But can it be an `Update` request? Well, note that at the time $R^*$ kills $R'$, $R'$ believes $R^*$ is alive. So in order to perform a write after this killing but still during epoch $n'$, $R'$ will need to either complete a successful `ApplyUpdate` on $R^*$ (step 5 of `Update`) or kill $R^*$ (step 6 of `Update`). But $R'$ cannot do either of these things, because its epoch number is out of date. Hence, $o'$ cannot be an `Update` request, and the proof is complete. □

Next we prove another important property of Niobe: all alive replicas stay "in sync" with each other. But we must first formalize the concept of being "in sync". Consider the clerk service on a replica when it is about to apply a mutation to its local data value $D$ during its execution the `ApplyUpdate` algorithm (Appendix B.2). The value of $D$ before the update is applied is called the *pre-value* for this execution of `ApplyUpdate`.

LEMMA D.2. *"Alive replicas are in sync with the primary." In any execution of step 5 of the `Update` algorithm, all executions of `ApplyUpdate` are applied to the same pre-value.*

*Proof.* When it `Reset`s, a primary ensures all secondaries it believes are alive agree with its data value (step 2 of `StartPrimary`, Appendix C) before it performs any updates. So the Lemma certainly holds for the first `Update` request performed by a given primary in a given epoch. Now apply induction on the number of updates performed by a given primary $R$ in a given epoch $n$: assume the previous `Update` was applied to the same pre-value at each secondary that $R$ believes is alive, and try to prove this holds for the next `Update` on the same primary $R$ in the same epoch $n$. Well, if $R$'s belief about the set of alive secondaries does not change before the next update, this is immediate—data values on the secondaries can't have changed, because if some secondary $S$ did accept an update from some machine other than $R$, it would need to have an epoch number other than $n$, and then $R$'s next `ApplyUpdate` to $S$ won't succeed. That takes care of the case when $R$'s belief about the set of alive secondaries does not change between updates. The only other possibility is that $R$ reincarnated one or more secondaries between updates. But

during such a reincarnation, $R$ forces the reincarnated secondary to agree with it, in step 5 of `InitiateReconciliation` (Appendix B.3). And it does so while holding `ReadWriteLock` for writing, so the data values on any other alive secondaries cannot change while this reconciliation takes place. □

That fact that updates are always applied to the same data value on all replicas where they are executed has a very important consequence that will simplify our analysis: rather than having to deal with general mutations, we can assume that every update is in fact a simple overwrite. This is formalized in the following corollary.

COROLLARY D.3. *"We may assume all mutations are overwrites." Suppose a customer sends a valid* `Update(M)` *request to some replica. Let $D$ be the unique value (which is well defined, by the previous Lemma) to which $M$ is applied successfully during step 5 of the* `Update` *algorithm. Let $D'$ be the result of applying $M$ to $D$. Let $O_{D'}$ be a special "overwrite" mutation that replaces any data value with $D'$. Then the history of the system's stable storage is identical to a history in which the* `Update(M)` *request is replaced by* `Update(`$O_{D'}$`)`.

*Proof.* This follows immediately from Lemma D.2: since every mutation is applied to an identical value, the results are the same as simply overwriting with the value that would have been produced by the mutation. □

In the remainder of this section, we will make use of this corollary by writing all `Update` operations as `Update(`$O_D$`)` for some $D$. Corollary D.3 guarantees that no generality is lost in doing so—all the remaining proofs apply equally well to general mutations, but they are simplified by viewing all updates as simple overwrites.

Suppose that at some instant $t$ (according to its own local clock), replica $R$ has cached epoch number $n$ and local data value $D$. How could $D$ have come to be written on $R$'s stable storage? Informally speaking, the value $D$ was "caused" by some update request from a customer at some time in the past. We would like to formally define this notion of causation, by tracking back through the history of the system to find the update request which is the "original ancestor" of the observed value $D$. The next few definitions enable us to do that.

We call the tuple $(D, R, n, t)$ an *instantaneous state*. Looking through all the algorithms in Appendices B and C, we see there are precisely two ways in which a local data value can change: execution of `ApplyUpdate`, and execution of `Reconcile`. Specifically, the $D$-value in the instantaneous state $(D, R, n, t)$ was written by precisely one of the following possibilities:

—`ApplyUpdate(`$n'$`, `$O_D$`)` executed on $R$ at some epoch $n'$, beginning at (local clock) time $t'$

—`Reconcile(`$n'$`, `$D$`)` executed on $R$ at some epoch $n'$, beginning at (local clock) time $t'$

In the first case, we say the *immediate ancestor* of $(D, R, n, t)$ is the tuple (`ApplyUpdate`, $D, R, n', t'$), and write this as

$$(D, R, n, t) \longrightarrow (\texttt{ApplyUpdate}, D, R, n', t'). \tag{10}$$

(The "$\longrightarrow$" symbol can be read as "is the child of".) In the second case, we say the *immediate ancestor* of $(D, R, n, t)$ is the tuple $(\texttt{Reconcile}, D, R, n', t')$, and write this as

$$(D, R, n, t) \longrightarrow (\texttt{Reconcile}, D, R, n', t'). \tag{11}$$

Any execution of `ApplyUpdate` is the result of a call from some execution of the `Update` algorithm. So we can define the immediate ancestor of a tuple $(\texttt{Apply-Update}, D, R, n, t)$ to be the tuple $(\texttt{Update}, D, R', n, t')$, where the $\texttt{ApplyUpdate(}O_D\texttt{)}$ request was sent by the `Update` algorithm on replica $R'$, with cached epoch $n$, and sent at time $t'$ (according to the local clock of $R'$). We write this as

$$(\texttt{ApplyUpdate}, D, R, n, t) \longrightarrow (\texttt{Update}, D, R', n, t'). \tag{12}$$

Similarly, any execution of `Reconcile` is the result of a call from some execution of either the `InitiateReconciliation` or `StartPrimary` algorithms. In the former case, we can define the immediate ancestor of a tuple $(\texttt{Reconcile}, D, R, n, t)$ to be the tuple $(\texttt{InitiateReconciliation}, D, R', n, t')$, where the `Reconcile` request was sent by the `InitiateReconciliation` algorithm on replica $R'$, with cached epoch $n'$, and sent at time $t'$ (according to the local clock of $R'$). In the latter case, the ancestor of $(\texttt{Reconcile}, D, R, n, t)$ is defined in a similar fashion as $(\texttt{StartPrimary}, D, R', n, t')$, where here $t'$ is the local time at which `StartPrimary` started execution on $R'$ with cached epoch $n'$. We write these cases as

$$\begin{aligned}(\texttt{Reconcile}, D, R, n, t) &\longrightarrow (\texttt{InitiateReconciliation}, D, R', n, t')\\ (\texttt{Reconcile}, D, R, n, t) &\longrightarrow (\texttt{StartPrimary}, D, R', n, t')\end{aligned} \tag{13}$$

respectively.

We continue defining immediate ancestors in a straightforward fashion. The immediate ancestor of a tuple $(\texttt{InitiateReconciliation}, D, R, n, t)$ or $(\texttt{Start-Primary}, D, R, n, t)$ is the immediate ancestor of the instantaneous state $(D, R, n, t)$, as in Equations (10) or (11).

On the other hand, a tuple $(\texttt{Update}, D, R, n, t)$ has no ancestor: it is the direct result of some customer sending the request $\texttt{Update(}O_D\texttt{)}$ to replica $R$. Since any local data value on any replica originates with some customer request, we can make the following definition. The *original ancestor* of an instantaneous state $(D, R, n, t)$ is the tuple $(\texttt{Update}, D, R', n', t')$ obtained by repeatedly finding immediate ancestors. This is written

$$(D, R, n, t) \rightsquigarrow (\texttt{Update}, D, R', n', t'). \tag{14}$$

The symbol "$\rightsquigarrow$" can be read as "is descended from".

LEMMA D.4. *"The epoch of an original ancestor is no greater than its descendent's epoch." Suppose* $(D, R, n, t) \rightsquigarrow (\texttt{Update}, D, R', n', t')$. *Then* $n' \leq n$.

*Proof.* It's easy to check that each application of the "immediate ancestor" operation does not increase the epoch number. □

The previous Lemma assures us that ancestors occur in earlier epochs, but do they also occur earlier in real time? The answer is yes, as the following Lemma states.

LEMMA D.5. *"The original ancestor precedes its descendent."* *Let* $(D, R, n, t)$ *be an instantaneous state on* $R$*, let* $(\mathtt{Update}, D, R', n', t')$ *be the original ancestor of* $(D, R, n, t)$*, and write* $u' = (u'_{req}, u'_{resp})$ *for the update operation executing at time* $t'$ *on* $R'$*. Let* $\tau_R(t)$ *be the wall clock time when* $R$*'s local clock reads* $t$*. Then*

$$\tau(u'_{req}) < \tau_R(t). \tag{15}$$

*Proof.* This is immediate, since every link in a chain of ancestors is produced by a message which consumes a strictly positive amount of wall clock time, so the elements of the chain are sequentially ordered according to their wall clock time. □

The previous two lemmas tell us, roughly speaking, that ancestors precede their descendents both in epoch number and in real time. The next lemma addresses a different type of ordering: the ordering between a successful read operation on a replica and the successful killing of that replica.

LEMMA D.6. *"A successful kill occurs after the start of a successful read."* *Let* $r = (r_{req}, r_{resp})$ *be a successful read on replica* $R$ *in epoch* $n$*, returning* $(\mathsf{Success}, D)$*. Suppose another replica* $R'$ *successfully kills* $R$ *in some epoch* $n' \geq n$*. Then the* `SetGSMInfo` *that* $R'$ *uses to kill* $R$ *occurs after* $r$ *was received. More formally, denote the* `SetGSMInfo` *operation by* $s = (s_{req}, s_{resp})$*. Then*

$$\tau(r_{req}) < \tau(s_{req}). \tag{16}$$

*Proof.* The proof will make heavy use of the notation $\tau_R(t)$, which is the wall clock time at which $R$'s local clock reads $t$.

The case $n' > n$ is easy due to serialization at the GSM: if $R$ successfully issues a `SetGSMInfo` (say, $s^* = (s^*_{\rm req}, s^*_{\rm resp})$) at epoch $n$, then $\tau(r_{\rm req}) < \tau(s^*_{\rm req}) < \tau(s_{\rm req})$ and we're done; otherwise some other replica kills $R$ at epoch $n$ and we are reduced to the case $n' = n$.

So assume $n' = n$. Write $L = \mathsf{HeartbeatPeriod} + \mathsf{GracePeriod}$. Consider the "most recent" heartbeat $h = (h_{\rm req}, h_{\rm resp})$ sent from $R'$ to $R$, defined formally as follows: $h_{\rm req}$ is the heartbeat request immediately preceding the sending of $s_{\rm req}$ by $R'$. Suppose $h_{\rm req}$ had sequence number $c$, and was sent at wall clock time $T$, measured as $t$ on $R$ and $t'$ on $R'$—so we have $\tau_R(t) = \tau_{R'}(t') = T$. Note that $R'$ never sends a heartbeat request with a given sequence number until it has received acknowledgments for all heartbeats in the current epoch with smaller sequence numbers (see step 3 of `HeartbeatLoop()`, Appendix C). So the only heartbeat requests that can arrive at $R$ from $R'$ in the time interval between $T$ and the sending of $s_{\rm req}$ have sequence number $c$. Let $H$ be the set of all such heartbeat requests arriving at $R$ in this interval of time. (In a typical scenario, there will be at most one element in $H$, but there could be more than one if $R'$ has retried its $c$th heartbeat multiple times.) If $H$ is empty, then $r_{\rm req}$ must arrive before $\tau_R(t + L)$, since otherwise `CheckPrimaryLease` will fail during the execution of $r$. Similarly, if all members of $H$ arrive after $r_{\rm req}$, the same argument applies. So in both these cases we have

$$\tau(r_{\rm req}) < \tau_R(t + L). \tag{17}$$

On the other hand, if some element of $H$ arrives before $r_{\text{req}}$, it must do so before $\tau_R(t+L)$, since otherwise (again) `CheckPrimaryLease` will fail during the execution of $r$[8]. But in this case, $r_{\text{req}}$ must arrive within time $L$ of the first heartbeat in $H$ (same reason again). That is, in this case we have

$$\tau(r_{\text{req}}) < \tau_R(t + 2L). \tag{18}$$

Combining (17) and (18), we see that in all cases we have

$$\tau(r_{\text{req}}) < \tau_R(t + 2L). \tag{19}$$

Now write $M = \mathsf{MaxClockDrift}$, and observe that because of the waiting period specified in step 4 of the `HeartbeatLoop` algorithm (Appendix C), $s_{\text{req}}$ is sent after $\tau_{R'}(t' + 2LM)$:

$$\tau_{R'}(t' + 2LM) < \tau(s_{\text{req}}). \tag{20}$$

But because $\tau_R(t)$ and $\tau_{R'}(t')$ represent the same instant $T$ in wall clock time, and $M$ is the maximum factor by which two machines can disagree about the length of an interval of time, we are guaranteed that $\tau_R(t+2L)$ is earlier than $\tau_{R'}(t'+2LM)$:

$$\tau_R(t + 2L) < \tau_{R'}(t' + 2LM). \tag{21}$$

Combining (19), (20), and (21) yields (16), completing the proof. □

A primary replica is *stable* if it reaches step 4 of `StartPrimary`, and its *stable value* is the local data value at the start of step 4. An epoch is *stable* if its primary ever becomes stable.

Stable epochs are important because they provide a kind of temporal barrier to the system: the unique stable value becomes visible to all subsequent epochs, and any uncertainty about the outcome of pending or failed updates is removed. The next lemma formalizes this notion of a "stable barrier".

LEMMA D.7. *(Stable Barrier Lemma) "A chain of ancestors whose epochs bracket a stable epoch has the same data value as the epoch's stable value." Suppose $n$ is a stable epoch with primary replica $R$ and stable value $D$. Let $(D', R', n', t')$ be an instantaneous state on some* Alive *replica in an epoch no earlier than $n$, so $n' \geq n$. Let $(\texttt{Update}, D', \tilde{R}, \tilde{n}, \tilde{t})$ be the original ancestor of this instantaneous state, and suppose that the ancestor is from an epoch strictly earlier than $n$, so $\tilde{n} < n$. Then $D' = D$.*

*Proof.* Assume $D' \neq D$ and argue for a contradiction. Pick the earliest element in the chain of ancestors with epoch at least $n$, say $a^* = (?, D', R^*, n^*, t^*)$, and let its immediate ancestor be $\tilde{a}^* = (?, D', \tilde{R}^*, \tilde{n}^*, \tilde{t}^*)$. Note that this implies $\tilde{n}^* < n \leq n^*$. The proof splits into two cases, depending on whether $R^*$ is Alive in epoch $n$. If $R^*$ is Alive during epoch $n$, replica $R$ sends it the value $D$ before becoming stable. So the value $D'$ written by operation $\tilde{a}^*$ in epoch $\tilde{n}^*$ was overwritten, contradicting the fact that $D'$ survives until $n^*$.

[8]It's also possible that $R$ will kill $R'$ in step 5 of `Heartbeat` before `CheckPrimaryLease` fails—and this contradicts our assumption that $R'$ kills $R$.

In the other case, $R^*$ is Dead in epoch $n$. But from the statement of the lemma, we know $R^*$ becomes Alive by epoch $n'$. So $R^*$ must be reincarnated in some epoch $N \geq n$. But this would mean $R^*$ receives a `Reconcile` in epoch $N$, contradicting the fact that the immediate ancestor of $a^*$ was $\tilde{a}^*$ in epoch $\tilde{n}^*$. □

It turns out that successful updates provide a similar type of "barrier" to the system, removing any uncertainty about all pending or previously failed updates. This type of "update barrier" is formalized in the next lemma. Recall from Corollary D.3 that any update operation $u$ is equivalent to a customer request of the form `Update(`$O_D$`)` for some $D$. We call $D$ the *value* of $u$.

LEMMA D.8. *(Update Barrier Lemma) "A chain of ancestors bracketing a successful update has the same data value as the successful update." Let* $(D', R', n', t')$ *be an instantaneous state in some stable epoch* $n'$*, and suppose* $(\texttt{Update}, D', \tilde{R}, \tilde{n}, \tilde{t})$ *is its original ancestor. Let* $u = (u_{req}, u_{resp})$ *be a successful update in epoch* $n$*, with value* $D$*. If* $\tilde{n} < n < n'$*, then* $D' = D$*. Moreover, if* $\tilde{n} = n < n'$ *and* $\tau_{\tilde{R}}(\tilde{t}) < \tau(u_{req})$*, then again* $D' = D$*.*

*Proof.* The proof is extremely similar to Lemma D.7. Let the instantaneous state at step 8 of the `Update` algorithm executed on behalf of $u$ be $(D, R, n, t)$. Assume $D' \neq D$ and argue for a contradiction. Pick the earliest element in the chain of ancestors with epoch at least $n$, say $a^* = (?, D', R^*, n^*, t^*)$, and let its immediate ancestor be $\tilde{a}^* = (?, D', \tilde{R}^*, \tilde{n}^*, \tilde{t}^*)$. Note that this implies $\tilde{n}^* < n \leq n^*$. The proof splits into two cases, depending on whether $R^*$ is Alive in epoch $n$. If $R^*$ is Alive during epoch $n$, replica $R$ certainly sends it the value $D$ during execution of $u$ (and may subsequently overwrite $D$ with some later updates in epoch $n$, also)—unless $\tilde{n} = n$ and $\tau(u_{\mathrm{req}}) < \tau_{\tilde{R}}(\tilde{t})$. So the value $D'$ written by operation $\tilde{a}^*$ in epoch $\tilde{n}^*$ was overwritten, contradicting the fact that $D'$ survives until $n^*$.

In the other case, $R^*$ is Dead in epoch $n$. But from the statement of the lemma, we know $R^*$ becomes Alive by epoch $n'$. So $R^*$ must be reincarnated in some epoch $N \geq n$. But this would mean $R^*$ receives a `Reconcile` in epoch $N$, contradicting the fact that the immediate ancestor of $a^*$ was $\tilde{a}^*$ in epoch $\tilde{n}^*$. □

Successful read operations provide the same type of barrier as successful updates and resets:

LEMMA D.9. *(Read Barrier Lemma) "A chain of ancestors bracketing a successful read has the same data value as the successful read." Let* $(D', R', n', t')$ *be an instantaneous state in some stable epoch* $n'$*, and suppose* $(\texttt{Update}, D', \tilde{R}, \tilde{n}, \tilde{t})$ *is its original ancestor. Let* $r = (r_{req}, r_{resp})$ *be a successful read in epoch* $n$*, returning data value* $D$*. If* $\tilde{n} < n < n'$*, then* $D' = D$*. Moreover, if* $\tilde{n} = n < n'$ *and* $\tau_{\tilde{R}}(\tilde{t}) < \tau(r_{req})$*, then again* $D' = D$*.*

*Proof.* The proof is essentially identical to the proof of Lemma D.8, except that we invoke Lemma D.2 to assert that if $R^*$ is Alive during epoch $n$, it must have local data value $D$ during the execution of $r$. □

We are now ready to prove the first important property of Niobe: every value read was written, but not overwritten.

*Proof of Theorem 6.1.* Suppose $r$ is executed on replica $R$, and the instantaneous state of $R$ at the start of step 4 of the `Read` algorithm is $(D, R, n, t)$. Let $(\texttt{Update}, D, \tilde{R}, \tilde{n}, \tilde{t})$ be the original ancestor of $(D, R, n, t)$, and take $\tilde{u} = (\tilde{u}_{\text{req}}, \tilde{u}_{\text{resp}})$ as the update operation being performed on $\tilde{R}$ at the instant $\tilde{t}$. Intuitively, $\tilde{u}$ is the operation that "caused" the value $D$ to be read during operation $r$. We must now show that no intervening updates could have successfully written a different value between the start of $\tilde{u}$ and the start of $r$.

Let us assume that such an "intervening overwrite" $u'$ exists, with the properties given in the theorem, and argue for a contradiction. If there is more than one such possible $u'$, choose the latest one in the latest epoch. Since $u'$ exists, its `Update` algorithm must have executed on some replica $R'$. Let $(D', R', n', t')$ be the instantaneous state of $R'$ as it begins step 8 of this `Update` execution.

To summarize the assumptions and notation so far:

—$r$ executed on $R$ with cached epoch $n$, and returned $(\mathsf{Success}, D)$

—$\tilde{u}$ executed on $\tilde{R}$ with cached epoch $\tilde{n}$,

—$u'$ executed on $R'$ with cached epoch $n'$, and returned $\mathsf{Success}$,

and the operations' messages are ordered according to:

$$\tau(\tilde{u}_{\text{req}}) < \tau(u'_{\text{req}}) < \tau(u'_{\text{resp}}) < \tau(r_{\text{req}}) < \tau(r_{\text{resp}}). \tag{22}$$

By Lemma D.4, $\tilde{n} \le n$. But the case $\tilde{n} = n$ is trivial (since updates in any one epoch are processed in series, in receipt order), so we may assume $\tilde{n} < n$.

By Lemma D.1, $n' \le n$. But $n' = n$ is also trivial, so we may assume $n' < n$.

Hence, we are left with two possible cases: (i) $\tilde{n} \le n' < n$, and (ii) $n' < \tilde{n} < n$.

*Case (i): $\tilde{n} \le n' < n$.* We apply the Update Barrier Lemma (Lemma D.8) to the ancestral chain $(\texttt{Update}, D, \tilde{R}, \tilde{n}, \tilde{t}) \rightsquigarrow (D, R, n, t)$ which brackets the successful update $u'$, obtaining the contradiction $D' = D$.

*Case (ii): $n' < \tilde{n} \le n$.* In this case we know $\tilde{R}$ considered $R'$ to be $\mathsf{Dead}$ when it performed the update $u$ (otherwise $R'$ would have had epoch number $\tilde{n}$ when $u$ was performed). So there exists some replica $R^*$ (possibly equal to $\tilde{R}$) that killed $R'$ at some epoch $n^*$, with $n' < n^* \le \tilde{n}$. But you can only be killed by a replica you consider to be alive. So when it performs $u'$, replica $R'$ believes that $R^*$ is alive. So to complete $u'$, $R'$ must either successfully `ApplyUpdate` on $R^*$, or kill $R^*$. But $R'$ can do neither of these, because its epoch $n'$ is less than $n^*$, so its attempts will be met with a $\mathsf{BadEpoch}$ response. □

We are now ready to prove the second important property of Niobe: written values are persistent.

*Proof of Theorem 6.2.* Suppose $r$ executes on $R$ and $r'$ executes on $R'$. Let the instantaneous state at the start of step 4 of the `Read` algorithm for these two operations be $(D, R, n, t)$ and $(D', R', n', t')$ respectively. Let the original ancestors of these instantaneous states be $(\texttt{Update}, D, \tilde{R}, \tilde{n}, \tilde{t})$ and $(\texttt{Update}, D, \tilde{R}', \tilde{n}', \tilde{t}')$ respectively.

By Lemma D.1, $n \le n'$. But the theorem is trivially true if $n = n'$, so in fact we may assume $n < n'$. By taking the "instantaneous state" of Lemma D.5 to

be step 4 of $r'$, we conclude $\tau(\tilde{u}'_{\mathrm{req}}) < \tau_{R'}(t')$. And since $\tau_{R'}(t') < \tau(r'_{\mathrm{resp}})$, we conclude

$$\tau(\tilde{u}'_{\mathrm{req}}) < \tau(r'_{\mathrm{resp}}). \tag{23}$$

Combining (23) and the no-intervening-updates assumption (7), we have

$$\tau(\tilde{u}'_{\mathrm{req}}) < \tau(r_{\mathrm{req}}). \tag{24}$$

We know from Lemma D.4 that $\tilde{n} \leq n$ and $\tilde{n}' \leq n'$. That leaves us with two cases: either (i) $n < \tilde{n}' \leq n'$, or (ii) $\tilde{n}' \leq n < n'$.

Case (i): $n < \tilde{n}' \leq n'$. $R$ was killed by some other replica, say $R^*$, in epoch $n^* \leq \tilde{n}'$. By Lemma D.6, this employs a message $s^*_{\mathrm{req}}$ after $r_{\mathrm{req}}$, so $r_{\mathrm{req}} \prec s^*_{\mathrm{req}}$. But $\tilde{u}'$ is a valid update, so $\tilde{u}'_{\mathrm{req}}$ arrives after $\tilde{R}'$ has communicated with the GSM in epoch $\tilde{n}'$, using, say, the message $\tilde{s}'_{\mathrm{req}}$. It might be that $s^*_{\mathrm{req}} = \tilde{s}'_{\mathrm{req}}$, but we certainly have $\tau(s^*_{\mathrm{req}}) \leq \tau(\tilde{s}'_{\mathrm{req}})$ since $n^* \leq \tilde{n}'$. So we have

$$\tau(r_{\mathrm{req}}) < \tau(s^*_{\mathrm{req}}) \leq \tau(\tilde{s}'_{\mathrm{req}}) < \tau(\tilde{u}'_{\mathrm{req}}), \tag{25}$$

contradicting (24).

Case (ii): $\tilde{n}' \leq n < n'$. Let $n^*$ be the first stable epoch after $n$, so $n < n^* \leq n'$. Let $R^*$ be the primary in $n^*$, and $D^*$ its stable value. By applying the Stable Barrier Lemma (Lemma D.7) to the ancestral chain $(\texttt{Update}, D', \tilde{R}', \tilde{n}', \tilde{t}') \rightsquigarrow (D', R', n', t')$, we immediately have

$$D^* = D'. \tag{26}$$

But now consider the sequence of primaries (each unstable, except for the first and last) from $R$ in epoch $n$ to $R^*$ in epoch $n^*$. (By the way, $R^*$ and $n^*$ are not the same as in case (i).) By a simple induction, each of the primaries after $R$ in this sequence has local data value $D$ before it executes `Reset`. The initial inductive step is provided by Lemma D.6: $R$ is killed after $r_{\mathrm{req}}$ (when its local data value is $D$), so its killer must also have local data value $D$ on `Reset` (since the killer was Alive during $R$'s `StartPrimary` and during any `Update` that wrote $D$ on $R$). The inductive step from each unstable primary to the following primary is also clear, since no unstable primary can accept an update. Thus $D^* = D$. Combining with (26), we have the contradiction $D = D'$. □

Finally, we provide proofs of Theorems 6.3 and 6.4.

*Proof of Theorem 6.3.* There are two cases depending on whether $o$ is an update or a read. First suppose $o$ is an update, say, $u'$. By assumption, $u'$ was successful. Let $r$ be a read operation received after the completion of $u'$—so $\tau(r_{\mathrm{req}}) > \tau(u'_{\mathrm{resp}})$. The statement of the Theorem is equivalent to claiming that the original ancestor of $r$ cannot be $u$, unless $u$ and $u'$ share the same data value. But this follows immediately from Lemma D.8.

In the other case, $o$ is a read. This case is similar, except that Lemma D.9 is invoked in place of Lemma D.8. □

*Proof of Theorem 6.4.* This is very similar to the two cases in the previous result, except that this time Lemma D.7 is invoked in place of Lemmas D.8 or D.9. □

## REFERENCES


Aguilera, M. and Frølund, S. 2003. Strict linearizability and the power of aborting. Technical report 2003-241, Hewlett-Packard Laboratories.

Alsberg, P. and Day, J. 1976. A principle for resilient sharing of distributed resources. In *Proc. 2nd Int. Conf. Software Engineering*. 627–644.

Barroso, L. A., Dean, J., and Holzle, U. 2003. Web search for a planet: The Google Cluster architecture. *IEEE Micro 23,* 2, 22–28.

Budhiraja, N., Marzullo, K., Schneider, F. B., and Toueg, S. 1993. The primary-backup approach. In *Distributed Systems*. ACM Press/Addison-Wesley Publishing Co.

Burrows, M. 2006. The Chubby lock service for loosely-coupled distributed systems. In *Proc. Symp. Operating System Design and Implementation (OSDI'06)*. 335–50.

Chang, F., Ji, M., Leung, S., MacCormick, J., Perl, S., and Zhang, L. 2002. Myriad: cost-effective disaster tolerance. In *Proc. Conf. File and Storage Technologies (FAST'02)*.

Dolev, D., Keidar, I., and Lotem, E. Y. 1997. Dynamic voting for consistent primary components. In *PODC '97: Proc. 16th ACM Symp. Principles of Distributed Computing*. 63–71.

Ghemawat, S., Gobioff, H., and Leung, S.-T. 2003. The Google file system. In *Proc. ACM Symp. Operating Systems Principles (SOSP)*.

Gray, C. and Cheriton, D. 1989. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP '89: Proc. 12th ACM Symp. on Operating Systems Principles*. 202–210.

Hassin, Y. and Peleg, D. 2006. Average probe complexity in quorum systems. *J. Comput. Syst. Sci. 72,* 4, 592–616.

Herlihy, M. P. and Wing, J. M. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12,* 3, 463–492.

Hsiao, H.-I. and DeWitt, D. 1990. Chained Declustering: A new availability strategy for multiprocessor database machines. In *Proc. 6th Int. Data Engineering Conference*. 456–465.

Kistler, J. J. and Satyanarayanan, M. 1992. Disconnected operation in the Coda file system. *ACM Trans. Comput. Syst. 10,* 1, 3–25.

Lamport, L. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers C-28,* 9 (September), 690–91.

Lamport, L. 1998. The part-time parliament. *ACM Transactions on Computer Systems 16,* 2, 133–169.

Lamport, L. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley.

Lamport, L. and Massa, M. 2004. Cheap paxos. In *Proc. Int. Conf. Dependable Systems and Networks (DSN'04)*. 307–314.

Lee, E. K. and Thekkath, C. A. 1996. Petal: Distributed virtual disks. In *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

Liskov, B., Ghemawat, S., Gruber, R., Johnson, P., and Shrira, L. 1991. Replication in the Harp file system. In *SOSP '91: Proc. 13th ACM Symp. Operating Systems Principles*. ACM Press, 226–238.

MacCormick, J., Murphy, N., Najork, M., Thekkath, C. A., and Zhou, L. 2004. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proc. of Symposium on Operating System Design and Implementation (OSDI 2004)*.

Oki, B. M. and Liskov, B. H. 1988. Viewstamped replication: a new primary copy method to support highly-available distributed systems. In *SOSP '88: Proc. 7th ACM Symp. Operating Systems Principles*. 8–17.

Papadimitriou, C. H. 1979. The serializability of concurrent database updates. *J. ACM 26,* 4, 631–653.

Perl, S. E. and Seltzer, M. 2006. Data management for internet-scale single-sign-on. In *Proc. 3rd USENIX Workshop on Real, Large Distributed Systems (WORLDS'06)*.

PETERSEN, K., SPREITZER, M. J., TERRY, D. B., THEIMER, M. M., AND DEMERS, A. J. 1997. Flexible update propagation for weakly consistent replication. In *SOSP '97: Proc. 16th ACM Symp. Operating Systems Principles*. ACM Press, New York, NY, USA, 288–301.

RISVIK, K. M., AASHEIM, Y., AND LIDAL, M. 2003. Multi-tier architecture for web search engines. In *Proc. 1st Latin American Web Congress (LA-WEB 2003), Empowering Our Web*. IEEE Computer Society, 132–143.

SAITO, Y., FRØLUND, S., VEITCH, A., MERCHANT, A., AND SPENCE, S. 2004. FAB: building distributed enterprise disk arrays from commodity components. *SIGOPS Oper. Syst. Rev. 38,* 5, 48–58.

VAN RENESSE, R. AND SCHNEIDER, F. B. 2004. Chain replication for supporting high throughput and availability. In *Proc. Symp. Operating Systems Design and Implementation (OSDI).*

...