Title
Consistent cloud computing storage as the basis for distributed applications

Permalink
https://escholarship.org/uc/item/14p5819g

Author
Anderson, James William

Publication Date
2011

Peer reviewed|Thesis/dissertation
Consistent Cloud Computing Storage as the Basis for Distributed Applications

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Computer Science by James William Anderson

Committee in charge:
Professor Amin Vahdat, Chair
Professor Alin Deutsch
Professor Sujit Dey
Professor George Papen
Professor Stefan Savage

2011
The dissertation of James William Anderson is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

Chair

University of California, San Diego

2011
# Table of Contents

Signature Page ................................................................. iii

Table of Contents ............................................................ iv

List of Figures ............................................................... viii

List of Tables ................................................................. ix

Acknowledgements ............................................................ x

Vita ................................................................................ xii

Abstract of the Dissertation .............................................. xiii

Chapter 1  Introduction .................................................... 1
  1.1 The Need for Cross Data Center Replication ................. 3
  1.2 Replication and Consistency ........................................ 4
  1.3 Application Messaging in Cloud Computing .................. 7
  1.4 Hypothesis ........................................................... 8
  1.5 Overview ............................................................ 10
  1.6 Summary ............................................................. 14

Chapter 2  Related Work ................................................... 15
  2.1 ZANTE and MapStore Related Work ......................... 15
  2.2 CORFU Related Work ............................................... 17
  2.3 XOMB Related Work ............................................... 19
  2.4 Acknowledgment .................................................... 21

Chapter 3  Consistency Costs and Consensus ....................... 22
  3.1 Costs of Strong Consistency .................................... 23
  3.2 PAXOS Overview .................................................... 25
  3.3 PAXOS Implementation ............................................ 27
    3.3.1 Errors and Failures .......................................... 27
    3.3.2 Indexed Consensus ............................................ 27
    3.3.3 Interface ....................................................... 28
    3.3.4 View Establishment ........................................... 30
    3.3.5 View Changes ................................................ 31
  3.4 Storing Service State ............................................. 32
  3.5 Evaluation ........................................................ 34
    3.5.1 Experimental Setup ......................................... 34
    3.5.2 PAXOS Evaluation ............................................ 35
    3.5.3 Network Bandwidth Overhead ............................... 39
5.6.3 Address book ................................................. 88
5.6.4 Limitations .................................................. 89
5.7 Summary ....................................................... 90
5.8 Acknowledgment ............................................. 90

Chapter 6  
Corfu ................................................................. 91
6.1 Motivation ...................................................... 92
6.2 Overview ....................................................... 94
  6.2.1 System Model ........................................... 94
  6.2.2 Corfu Overview ....................................... 95
6.3 Corfu Protocol ................................................. 96
6.4 Design Details ................................................ 101
  6.4.1 Controller ............................................... 101
  6.4.2 Distributed Switches ................................. 102
  6.4.3 Failure Detection ................................... 102
6.5 Analysis ....................................................... 103
  6.5.1 Design Alternatives ................................. 103
  6.5.2 Markov Models ....................................... 104
  6.5.3 Analysis Results ................................... 106
6.6 Evaluation ..................................................... 109
  6.6.1 Object Store ........................................... 109
  6.6.2 Experimental Setup ................................. 109
  6.6.3 Time to Recover ................................... 110
  6.6.4 Model Validation ................................... 112
6.7 State Transfer ............................................... 116
6.8 Summary ....................................................... 117
6.9 Acknowledgment ............................................. 118

Chapter 7  
XOMB ................................................................. 119
7.1 Overview ....................................................... 122
  7.1.1 Load Balancing Switches ......................... 123
  7.1.2 Architecture ......................................... 125
7.2 Design .......................................................... 126
  7.2.1 Pipelines and Modules ......................... 127
  7.2.2 Control Plane .................................... 131
  7.2.3 Design Discussion ................................. 133
7.3 Implementation ............................................... 135
  7.3.1 Pipeline Libraries ................................. 135
  7.3.2 Control Module .................................. 135
  7.3.3 Data Plane ......................................... 137
7.4 Example: Distributed Object Store ...................... 140
7.5 Evaluation ..................................................... 142
  7.5.1 Programmability .................................. 142
LIST OF FIGURES

Figure 1.1: System overview .............................................. 10
Figure 1.2: System architecture ........................................ 13

Figure 3.1: Simple PAXOS messages ................................... 26
Figure 3.2: PAXOS messages with leader ............................. 26
Figure 3.3: PAXOS overview timeline ................................. 35
Figure 3.4: PAXOS CPU usage ........................................... 36

Figure 4.1: ZANTE deployment scenario. ............................ 42
Figure 4.2: ZANTE service architecture .............................. 47
Figure 4.3: ZANTE ALLOC and OPEN example .................... 58
Figure 4.4: ZANTE system architecture .............................. 59

Figure 5.1: MAPSTORE throughput and latency ..................... 76
Figure 5.2: Latency CDFs for state table operations ............... 78
Figure 5.3: MAPSTORE aggregate throughput ...................... 79
Figure 5.4: MAPSTORE throughput during failure .................. 81
Figure 5.5: MAPSTORE file system layout ......................... 86
Figure 5.6: MAPSTORE backup maps ................................. 87
Figure 5.7: MAPSTORE backup layout ............................... 88

Figure 6.1: State loss during recovery ............................. 92
Figure 6.2: CORFU protocol overview ............................... 98
Figure 6.3: Recovery timeline comparison .......................... 103
Figure 6.4: Markov model for NoBackup ............................ 104
Figure 6.5: Markov model for WARBackup .......................... 105
Figure 6.6: CORFU recovery timings ............................... 111
Figure 6.7: CORFU throughput during failure ..................... 113
Figure 6.8: CORFU failure and recovery plot ...................... 114

Figure 7.1: xOMB system architecture .............................. 125
Figure 7.2: xOMB example pipeline ................................. 128
Figure 7.3: xOMB forwarding module ............................... 130
Figure 7.4: xOMB HTTP pipeline ................................. 135
Figure 7.5: Message and buffer state snapshots .................. 138
Figure 7.6: xOMB HTTP throughput ............................... 143
Figure 7.7: xOMB front-end scalability (1KB files) ............. 144
Figure 7.8: xOMB back-end scalability ............................ 144
Figure 7.9: xOMB comparison under extreme load ............... 146
Figure 7.10: xOMB vs F5 attack detector throughput ............ 147
Figure 7.11: xOS vs. Walrus write throughput .................... 147
LIST OF TABLES

Table 1.1: Summary of consistency models ...................... 5
Table 3.1: Summary of important Paxos tables ................. 33
Table 3.2: Protocol overhead ..................................... 39
Table 4.1: Consistency availability requirements ............... 45
Table 4.2: Summary of ZANTE services ........................ 48
Table 5.1: MapStore API ............................................. 62
Table 5.2: Summary of MapStore tables ......................... 72
Table 5.3: Throughput comparison for table configurations .... 77
Table 5.4: MapStore scalability results .......................... 80
Table 5.5: BigTable blogging application results .............. 82
Table 5.6: SQS application results ................................. 84
Table 6.1: Description of failure/repair rates ................... 106
Table 6.2: MTFF and unavailability for NoBackup .............. 107
Table 6.3: MTFF and unavailability for WARBackup ............ 107
Table 6.4: MTFF and unavailability for LocalRecovery .......... 108
Table 7.1: xOMB module code lengths ............................ 143
Table 7.2: NFS latency and throughput .......................... 150
ACKNOWLEDGEMENTS

I would like to acknowledge my colleagues who contributed directly or indirectly to this work. Chip Killian, with whom I worked on the Mace language and toolkit that has been instrumental for developing these systems, as well as the predecessor to MapStore; Alex Rasmussen, who helped build and evaluate MapStore, Zante, and Corfu, and co-author those papers; Ryan Braud and Rishi Kapoor, who helped build xOMB and co-author that paper; Harsha Madhyastha, for co-authoring Corfu; faculty co-authors Keith Marzullo and Hein Meling for their help with Corfu; and my co-author and advisor, Amin Vahdat. I would also like to thank the staff and systems faculty at UCSD who contributed feedback on these projects and helped make sure the required resources were available; Stefan Savage, Geoff Voelker, Alex Snoeren, and Brian Kantor deserve special thanks.

Chapter 2, in part, is currently being prepared for submission for publication of the material. Anderson, James; Rasmussen, Alexander; Braud, Ryan; Kapoor, Rishi; Meling, Hein; Madhyastha, Harsha; Marzullo, Keith; Vahdat, Amin. The dissertation author was the primary investigator and author of this material.

Chapter 3, in part, is currently being prepared for submission for publication of the material. Anderson, James; Rasmussen, Alexander; Vahdat, Amin. The dissertation author was the primary investigator and author of this material.

Chapter 4, in part, is currently being prepared for submission for publication of the material. Anderson, James; Rasmussen, Alexander; Vahdat, Amin. The dissertation author was the primary investigator and author of this material.

Chapter 5, in part, is currently being prepared for submission for publication of the material. Anderson, James; Rasmussen, Alexander; Meling, Hein; Madhyastha, Harsha; Marzullo, Keith; Vahdat, Amin. The dissertation author was the primary investigator and author of this material.

Chapter 6, in part, is currently being prepared for submission for publication of the material. Anderson, James; Rasmussen, Alexander; Meling, Hein; Madhyastha, Harsha; Marzullo, Keith; Vahdat, Amin. The dissertation author was the primary investigator and author of this material.

Chapter 7, in part, is currently being prepared for submission for publication of the material. Anderson, James; Braud, Ryan; Kapoor, Rishi; Vahdat, Amin.
The dissertation author was the primary investigator and author of this material.
VITA

2003 B. S. in Computer Science, B. S. in Literature, Massachusetts Institute of Technology

2004 M. Eng. in Electrical Engineering and Computer Science, Massachusetts Institute of Technology

2011 Ph. D. in Computer Science, University of California, San Diego

PUBLICATIONS


ABSTRACT OF THE DISSERTATION

Consistent Cloud Computing Storage as the Basis for Distributed Applications

by

James William Anderson

Doctor of Philosophy in Computer Science

University of California, San Diego, 2011

Professor Amin Vahdat, Chair

Cloud computing aims to simplify distributed application development by providing location independent access to vast computing resources with services implementing fault tolerance and scalability. As applications and their associated data migrate “to the cloud,” or data center hosted deployments, users increasingly depend on the availability and reliability of these services. Not only does downtime inconvenience users who rely on these services for access to their personal data, but it also results in lost revenue and user confidence. To increase availability, services replicate their systems and data at multiple data centers, typically using some form of eventual consistency, which results in complicated application and system designs and limits the types of applications that can use these services.
This dissertation argues that strongly consistent cloud computing storage and infrastructure services can both significantly simplify and enable new classes of distributed applications by providing a basis for scalability, reliability, and availability. We propose techniques to enable cross-data center replication based on fine-grained scalable replicated state machines that offer either strong or weak consistency. Building upon these techniques, this dissertation describes how the limiting factor for availability in strongly consistent wide-area replicated services often is the time to recover a replica after a failure. We present a solution to this limitation based on a new protocol that maintains strong consistency while greatly reducing recovery time for wide-area services.

Deploying these techniques and protocols for consistent cloud computing requires message routing services capable of performing arbitrary processing based on large amounts of service state. This dissertation presents a new framework for building scalable, programmable, extensible software message routers and middle-boxes using commodity hardware. This framework performs favorably compared to a commercial load-balancing switch while its flexible architecture enables the protocols we propose for implementing wide-area strongly consistent replication.
Chapter 1

Introduction

Web-based distributed applications are becoming increasingly important and pervasive in the lives of many people. Users have come to rely on their email, banking, photos, music, documents, and files being accessible from anywhere and at any time through web services. Social networking and communication services providing text, voice, and video chat are becoming increasingly pervasive. E-commerce sites handle millions of dollars worth of purchases a day. These examples represent the growing class of “cloud computing” applications, which have the defining characteristic of location independence, because these services are delivered from the Internet “cloud.”

Internet-based services and cloud computing applications share certain common requirements of scalability, performance, availability, reliability, security, and efficiency. A service must be able to support potentially hundreds of millions of users and grow by adding more capacity. Users desire good performance, which typically means low latency for a responsive experience and high bandwidth to avoid waiting for content to load. As users entrust the service with important personal data, they expect the service will reliably and faithfully store that data, protecting it from various hardware and software failures. Furthermore, users expect to be able access the service and its associated data at any time. Unavailability for any reason will often result in the service losing revenue and user confidence. Finally, services seek to minimize cost in terms of hardware and operating expenses by working efficiently and using as few resources as possible.
Delivering a service to a global set of users while meeting the requirements of scalability, performance, availability, reliability, security, and efficiency poses significant challenges. The cloud computing model is attractive because it promises access to vast computing resources from anywhere with network connectivity, with the goal of simplifying distributed application development by providing abstractions to make applications highly available, scalable, and deliver good performance.

Beyond simplifying applications, cloud computing offers other advantages relative to the traditional model of dedicated hosting. Because cloud computing provides services on a usage-based system, or as utility computing, businesses do not need to make large capital investments buying hardware and hosting space, but can rather pay a cloud provider for service as needed. Businesses also no longer need to worry about over-provisioning to meet peak demand and having otherwise underutilized resources. By consolidating resources into the cloud, cloud operators can reduce their hardware costs with economies of scale while also reducing ongoing operational costs such as power, maintenance, backup, and administration by centralizing these tasks and resources.

Cloud computing represents the evolution of the client-server model of distributed computing. The client-server model has remained popular because of its elegance and simplicity: application processing and state is divided among clients and a server, with the benefit being that the server state, application logic, and processing resources can be made available to any client. With Internet-based client-server computing, the clients typically store little state and do minimal processing, whereas the servers store all the application state and execute the majority of the application processing.

Unfortunately, the very simplicity of the basic client-server model also results in its limitations. First, a single server, no matter how fault-tolerant, represents a single-point of failure. Second, a single server, no matter how well provisioned, will not scale beyond a certain number of clients. Distributed applications designed with a peer-to-peer model of communication attempt to avoid these limitations by distributing application logic and state across many servers, and using distributed replication, resource location, and load-balancing protocols to avoid
bottlenecks and single-points of failure. While achieving greater availability and capacity than a single server, these peer-to-peer systems also tend to be much more complicated and difficult to program.

Cloud computing aims to combine the simplicity of the client-server model with the scalability and availability provided by peer-to-peer distributed systems. Of course, the "cloud" itself is an illusion. Cloud applications and services are delivered by millions of servers hosted in data centers spread around the world, and all these servers run big complicated distributed systems. These systems attempt to deliver availability and performance in a scalable fashion by implementing fault tolerance, load balancing, replication, virtualization, resource allocation, and other required functionality. While there are many real-world challenges facing geo-replicated distributed systems and cloud computing services, this work focuses primarily on cross-data center replication.

1.1 The Need for Cross Data Center Replication

Cloud services require cross data center replication to meet target levels of availability and performance. Highly available services aim for five-nines of availability (99.999% uptime), or less than five minutes of downtime per year. This is the gold standard for availability set by the notably well tested public switched telephone network (PSTN). No single data center, no matter how well provisioned, can meet this requirement, because occasionally disasters will take an entire data center offline. Even with multiple redundant copies of the service within a data center, catastrophic events [103], loss of power [77], or loss of connectivity [76] can render the data center (and the service) unavailable. For example, in June 2008, an electrical explosion and fire at a Houston data center operated by The Planet caused the entire facility to be offline for over a day [78]. In July 2009, a fire at Seattle’s Fischer Plaza data center caused several major websites including Redfin and Microsoft’s Bing Travel to become unavailable for several days [42].

Additionally, cloud services want to provide low latency access for a responsive user experience. Because of fundamental speed-of-light delays, a service cannot
provide low latency access to a global set of users from a single location. Rather, the service operates from many data centers strategically located near dense user populations. The service then achieves faster response times by placing a user’s data at a data center site near the user. To maintain a consistent experience for a user, the service must direct all requests from a given user to the correct data center. Users, however, do not stay in one location, and if a user’s data is fixed to a single site, then when that user moves somewhere else, they see longer delays, even though there might be another data center closer to them.

Further, because interactive web applications require multiple round-trip requests between clients and servers, ensuring the best user experience requires minimizing the latency of individual requests. Network latency and probability of encountering network congestion both typically increase with the geographic distance between endpoints. Hence, service providers cannot provide low-latency access to a global client population using a single data center. Additionally, failures in the network beyond the control of any Internet service may render a site unavailable to a subset of the user population and a disaster may render a particular site wholly unavailable for hours or days at a time. Studies have shown Internet path availability averages only two-nines [56].

1.2 Replication and Consistency

Replicated systems must choose a consistency model, which describes how and when replicas exchange updates. Consistency is a spectrum, ranging from weak or eventual consistency [105] to strong or sequential consistency [65]. At a high level, weak consistency offers very few assurances about when replicas see updates—almost anything is possible. In particular, replicas may lose updates or receive duplicate or conflicting updates. Strong consistency guarantees that all replicas see all updates in the same order. Table 1.1 summarizes these differences.

All existing cloud storage services that have published their consistency model [43, 48, 51, 100]—and thus likewise the applications built on them—use some form of eventual consistency. Eventual consistency, however, has a number of
Table 1.1: Summary of consistency models

<table>
<thead>
<tr>
<th>Eventual Consistency</th>
<th>Sequential Consistency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous</td>
<td>Synchronous</td>
</tr>
<tr>
<td>Low latency</td>
<td>Higher latency</td>
</tr>
<tr>
<td>Best effort</td>
<td>Guaranteed semantics</td>
</tr>
<tr>
<td>Updates in any order</td>
<td>All updates in same order</td>
</tr>
<tr>
<td>Complicates applications</td>
<td>Simplifies applications</td>
</tr>
<tr>
<td>May lose updates</td>
<td>-</td>
</tr>
<tr>
<td>Conflicting updates</td>
<td>-</td>
</tr>
<tr>
<td>Inconsistent state</td>
<td>-</td>
</tr>
</tbody>
</table>

unfortunate consequences in the context of cloud computing,

First, eventual consistency makes applications more complicated. This is from the necessary logic to handle inconsistencies as best as possible. Consider Facebook, a leading social networking cloud computing company, as an example. Facebook currently employs two data centers in the United States, a primary on the west coast and a backup on the east coast. The backup acts as a slave for mutable data to the primary. All updates must be directed to the primary and subsequently asynchronously replicated to the backup [100]. To avoid user confusion (e.g., an east coast user performs an update followed immediately by a read that returns stale data), Facebook sets a cookie in HTTP responses following a write to direct all subsequent requests for some (arbitrarily chosen) time period to the primary. Google’s App Engine uses a similar master/slave architecture [26].

Of course, this approach poses a number of limitations: service availability is constrained by the availability of the primary, disaster recovery is not improved, the primary must be provisioned to interactively manage the service’s entire write workload, and adding additional wide-area replicas places further stress on the primary. While there are clearly alternative, perhaps superior, techniques for architecting wide-area replication, the above architecture underscores the difficulty of deploying such functionality even for premier network services.

Second, eventual consistency limits the classes of application that can be
Important Limitations

Due to S3’s “eventual consistency” limitations file creation can and will occasionally fail. Even after a successful create, subsequent reads can fail for an indeterminate time, even after one or more successful reads. Create and read enough files and you will eventually encounter this failure. This is not a flaw in s3fs and it is not something a FUSE wrapper like s3fs can work around. The retries option does not address this issue. Your application must either tolerate or compensate for these failures, for example by retrying creates or reads.

There are applications that users would like to have, such as s3fs, that simply will not work with eventual consistency. Looking forward, applications for finance, health care, and utilities will want to leverage the benefits of cloud computing. All of these applications require strong consistency.

Finally, eventual consistency poses a danger for applications and their associated data. In the current state of cloud computing, we have big distributed systems with few invariants about their state, and they can fail in ways that are difficult to understand. Weak consistency makes applications brittle. Consider the recent February 2009 global Gmail outage [44]. Details are sketchy, but the outage was attributed to a cascading service overload when, during scheduled maintenance, a subset of users were manually switched to a backup copy of e-mail in a European data center. This once again underscores the difficulty of managing wide-area replication for performance and availability.

More recently, in April 2011, Amazon Web Services suffered a major outage effecting its East Coast “availability zone.” This outage significantly impacted many of their customers. In their analysis of the failure [19], AWS commented on the need to make it easier to build applications spanning availability zones, or multiple data centers:
Making it Easier to Take Advantage of Multiple Availability Zones

A related finding from this event is we need to do a better job of making highly-reliable multi-AZ deployments easy to design and operate. Some customers' applications (or critical components of the application like the database) are deployed in only a single Availability Zone, while others have instances spread across Availability Zones but still have critical, single points of failure in a single Availability Zone. In cases like these, operational issues can negatively impact application availability when a robust multi-AZ deployment would allow the application to continue without impact. We will look to provide customers with better tools to create multi-AZ applications that can support the loss of an entire Availability Zone without impacting application availability. We know we need to help customers design their application logic using common design patterns. In this event, some customers were seriously impacted, and yet others had resources that were impacted but saw nearly no impact on their applications.

This event further highlights both the complexity existing applications face when trying to achieve high availability, as well as the growing importance for cloud service providers to offer better systems and abstractions for building reliable applications.

1.3 Application Messaging in Cloud Computing

As described earlier, data centers hosting cloud services run large distributed systems with thousands of servers. While these distributed systems may communicate within the data center in a peer-to-peer fashion, a service exposes a client-server model interface to its users. Cloud services must then create the illusion of their large distributed systems appearing as if they were a small set of servers.

Cloud providers achieve this using some combination of specialized network appliances, hardware switches, and software routing services, depending on the complexity of the particular task. In general, services require three broad classes of functionality. First, they perform static load-balancing across a set of servers.
In the simplest type of load-balancing, any server can handle any request and the goal is to spread the load evenly among a pool of machines. A slightly more complicated usage may direct requests to particular pools of servers based on a statically configured mapping, e.g., requests for /images go to one set of servers and requests for /videos go to another set.

Second, some services need to dynamically route messages to particular servers based on both the contents of the request and dynamic service state. For example, Amazon’s S3 storage service uses a bucket ID to direct a request to the server(s) that store that particular bucket. This is typically a more challenging type of forwarding compared to static load balancing, and often requires services to write customized software forwarding layers to accomplish the task, often used in conjunction with application-layer load-balancing switches.

Finally, some deployments use network appliances to enhance performance and efficiency of their service. Some example functions include protocol inspection and filtering for malicious content, caching and compression for additional performance, connection collapsing for back-end server efficiency, and handling encryption and authentication for security.

Currently, cloud service providers may use multiple specialized products from a variety of vendors. Because these network appliances offer limited extensibility, cloud services often must supplement them with custom software services to achieve the desired functionality. As cloud services offer more features and adapt new protocols, they are increasingly relying on custom software that they can control and modify to meet their requirements.

1.4 Hypothesis

The hypothesis of this dissertation is that strongly consistent cloud computing storage and infrastructure services can both significantly simplify and enable new classes of distributed applications by providing a basis for scalability, reliability, and availability. This dissertation provides a partial solution to the problem of building scalable, high performance, and highly available distributed applications
that advances the state of the art for wide-area replicated systems.

- First, this dissertation argues that certain classes of distributed applications require strong consistency, and many applications would benefit from the simplified semantics it affords. This dissertation argues that consistency needs to be an explicit option exposed by services, allowing application developers to choose the consistency model that best meets their needs, and shows how this can be achieved with an implementation.

- Second, this dissertation argues that the costs of using strong consistency in a wide-area system are acceptable in modern deployments, and further includes a detailed performance study and evaluation of these costs.

- Third, this dissertation argues that fine-grained replicated state machines can be used to provide strongly consistent wide-area services. To support the scale at which cloud services operate, this dissertation introduces a technique for building scalable, efficient, and high performance replicated state machines using dynamic demand binding.

- Fourth, this dissertation argues that with strongly consistent wide-area replicated services, the limiting factor for availability is the time to recover a replica after a failure. This dissertation describes a technique called local recovery that maintains strong consistency while significantly reducing recovery time for wide-area services.

- Fifth, this dissertation argues that the message routing layer cloud services require can be more easily and cost-effectively implemented using scalable, programmable software middleboxes running on commodity servers. This dissertation describes an architecture and implementation for building these programmable and extensible middleboxes.
Figure 1.1: The systems described in this dissertation are deployed at multiple data centers (A – F) connected by wide-area links. Service state is replicated at multiple data centers, and clients interact with storage and compute nodes hosted in these data centers by sending requests over the Internet to one of the replicas. An application-layer front-end switching service forwards requests to the appropriate back-end servers.

1.5 Overview

Previously, we argued that cloud services must be delivered from multiple data centers to meet target levels of availability and performance. In this section, we overview the systems presented in this dissertation, PAXOS, ZANTE, MAPSTORE, CORFU, and xOMB, briefly describing at a high level how they work together to simplify building highly available and high performance distributed applications.

The context for this work is a deployment scenario of multiple data centers, each with thousands of servers, as shown in Figure 1.1. Clients use these
distributed applications by sending requests over the Internet to these servers. For scalability, we separate metadata from the actual application data, and then we replicate each of these at multiple data centers. We use a scalable software message routing service to direct client requests and application protocol messages to the appropriate servers within each data center.

Our approach for building scalable and highly available distributed applications is based on services built using scalable replicated state machines. State machines store application state and modify this state according to application logic as a result of updates submitted by clients. The state machine model is a natural fit for almost any cloud application, as these are designed for a client-server model in which clients submit requests (updates) to the server (the state machine).

To make state machines fault tolerant, multiple state machines can execute the same application logic. If all these state machines start with the same initial state, and receive the same deterministic updates in the same order, then they will all have the same final state. These are called replicated state machines, with each state machine being a replica. Some of the challenges addressed in this dissertation include building services that support large numbers of replicated state machines and making them efficient.

As an example of the replicated state machine approach to cloud computing, consider a cloud-based email system. Each user could be assigned a unique state machine. The application state would include the mailboxes, messages, and their flags. Mailboxes would initially be empty. The application logic would modify the state in response to updates, such as receiving an email (adding it to the inbox), sending an email (adding it to the sent mailbox), marking a message as read (changing the flags), or moving a message to a different mailbox (deleting it from one mailbox and adding it to another). To make this mail service highly available, the provider may replicate every state machine across several data centers, using a consistency protocol to ensure that each replica sees all updates. As popular cloud-based email services have hundreds of millions of users, this system would need to support a large number of replicated state machines.

We base our consistent replicated state machines on the consensus protocol
Paxos [67], which provides strong consistency. Chapter 3 describes this protocol and our implementation of it in detail, as well as how it can be used to consistently order updates in a consistent replicated log. The consistent replicated log is the basis for replicated state machines, as each replica can apply updates in the order they appear in the log.

Building strongly replicated services that can support very large numbers of replicated objects (such as mail accounts) requires scalable techniques for distributing replicas both across data centers and on servers within data centers, as well as locating these replicas. Additionally, creating and accessing replicas should be fast. Chapter 4 describes ZANTE, an architecture for building scalable and high performance replicated state machines.

To further simplify distributed applications, we build a storage layer upon our ZANTE framework. Our storage system, MapStore, provides a powerful sorted map interface with fine-grained operations. Chapter 5 describes this system and the applications we built with it in detail.

Figure 1.2 shows the layered architecture of our approach. The top layer contains the application logic (MapStore) and the application state (the maps $X,Y,Z$). These are consistently replicated using ZANTE replicated state machines, which use a consensus interface for consistent replicated logs provided by Paxos. Each state machine has its own replicated log and state.

When replicating data across the wide-area, as shown in Figure 1.1, the limiting factor for system availability can become the time it takes to recover a replica after a failure. Replicas with large amounts of state must recover this state by copying it from another replica, which can take several hours over congested or bandwidth constrained wide-area links. If other replicas fail while the previous ones are still recovering, then that replicated object becomes unavailable. In Chapter 6, we present a protocol, Corfu, that overcomes this problem through a technique called local recovery. Corfu works with replicated state machines such as ZANTE by splitting the role of each replica into three components within the same data center: a primary, an asynchronously updated backup, and a switch. Corfu provides a mechanism for safely and consistently failing over to the backup when
Figure 1.2: Layered service architecture for replicated state machines. The figure depicts three replicated objects, $X, Y, Z$, each of which have their own consistent replicated log. The state for these objects and their logs are stored together efficiently on disk.
the primary fails.

Finally, the services described in this dissertation, including ZANTE, MapStore, and Corfu, all rely on software message routing or switching services (see inset for data center $D$ in Figure 1.1) to forward client and protocol messages to the appropriate servers. Existing hardware based load balancing switches and software based reverse proxies do not provide the programmability or scalability required by these services. In Chapter 7, we present xOMB, our framework for building extensible, programmable, and scalable software middleboxes based on commodity hardware.

1.6 Summary

As cloud computing services and applications become more important to individuals, businesses, and other organizations, there is an increasing need to ensure that these applications are reliable and highly available. This dissertation describes techniques for replicating cloud services across multiple data centers to deliver acceptable levels of performance and availability, while maintaining scalability of the application. Furthermore, we will present architectures and services designed to simplify distributed application development and enable new classes of cloud applications.
Chapter 2

Related Work

The systems described in this dissertation build on a large body of related work in distributed systems, ranging from overlay networks, to cluster services, to high performance routers. In this chapter, we briefly describe the most relevant related work.

2.1 ZANTE and MAPSTORE Related Work

Distributed Hash Tables MapStore’s locator service uses consistent hashing techniques long established in the distributed hash table literature [95,102]. While DHTs provide many additional benefits in terms of self-organization and fault tolerance, we assume that MAPSTORE sites are relatively well-provisioned and stable, so many of these features are not required in our environment.

OpenDHT [94] also provides shared storage as a network service. Arbitrary applications take advantage of a DHT “utility”, running across the Internet, to perform put() and get() operations. MAPSTORE provides an API similar to OpenDHT. Relative to OpenDHT, we focus on the appropriate architecture for delivering maps across replicated data centers with a focus on strong consistency.

Data Center Storage. A number of recent efforts have proposed abstractions to support the storage requirements of data center applications. Yahoo’s PNUTS [43] provides replicated storage across data centers but uses eventual consistency for
replication. Amazon’s Dynamo [48] also targets one or more data centers and provides variable degrees of eventual consistency through their use of a “sloppy quorum” that specifies the number of readers and writers that must participate in an update. Neither service targets strongly-consistent applications, but instead target applications that can operate with weaker consistency in exchange for higher availability and lower critical-path latency.

Sinfonia [21] provides strong consistency, but restricts itself to a single data center. It also uses primary-copy replication for replica synchronization instead of Paxos, at the expense of having to rely on synchrony to prevent false failover when recovering from primary replica failure. The Google File System (GFS) [52] distributes and replicates files across multiple nodes in a cluster with a centralized master tracking per-file object location.

**Replicated State Machines.** Our work builds on a large body of protocols to build replicated state machines [98] for fault tolerance. Virtual Synchrony [27] demonstrated, in part, how to establish a total ordering in a distributed environment. Paxos [67] and Timestamped Replication [64] improved the liveness of previous consensus protocols by tolerating some subset of machine failures. We also draw upon systems that use replicated logs to tolerate Byzantine faults [35, 63]. Recent efforts [36, 70] explore some of the practical issues associated with deploying Paxos as a service. We add to this body of work by demonstrating the practicality of deploying replicated state machines using a Paxos service in the wide area, and quantifying in detail the costs involved in a tuned systems implementation.

**Providing Strong Consistency.** Apache ZooKeeper [59] provides serializability through consensus using a custom leader-based atomic broadcast protocol. It also supports *watches* which are analogous to our version-based waiting reads. Boxwood [71] uses Paxos to provide strongly-consistent access to its B-tree data structure, although it uses Paxos primarily for managing membership lists for its various support services. Boxwood uses synchronous replication for fault tolerance and relies on a high-speed LAN to provide its target latencies, making it less appropriate for wide-area consideration. Mencius [73] also implements a Paxos-like
protocol designed for wide area applications, but it uses a rotating leader whereas we have a static leader per map. Our work focuses on the performance costs of a storage system that leverages a range of consistency semantics on top of a replicated state machine. Chubby [30] provides strong consistency through a locking service, and emphasizes reliability and availability at the cost of high performance, whereas MapStore endeavors to provide all three to the extent possible.

2.2 CORFU Related Work

Recovery-Oriented Computing Our work shares the same goals of Recovery-Oriented Computing [32], which aims to increase availability by focusing on decreasing recovery time from inevitable failures, rather than trying to prevent them. While techniques such as microrebooting [33] can be used in conjunction with CORFU, they fundamentally cannot recover from disk failures, which require additional replication for fast recovery.

Previous Paxos Systems A number of recent efforts address the problem of implementing Paxos and making it perform well in real-world usage scenarios. Researchers at Google [36] have provided insight into how a real-world Paxos implementation handles different classes of failures that are not well-specified in the original protocol. SMART [70] performs view changes by retaining the old view until the new view has been established. Like CORFU, SMART attempts to decrease the amount of state transferred during recovery; it does this by letting new replicas use part or all of other co-located replicas’ shared state.

Wide-Area Systems Both previously mentioned systems assume that Paxos is running over a LAN. Mencius [73] delivers high performance for Paxos in a wide-area setting by partitioning the leader role for proposals among replicas in a round-robin fashion. However, the focus of Mencius is on wide-area performance, and it provides the same availability as [67]; however, their approach could be implemented in CORFU.
Steward [24] addresses the problem of providing Byzantine fault tolerance in a wide-area setting consisting of a number of wide-area sites containing several server replicas. Rather than perform a BFT protocol across all replicas in the system, Steward performs BFT at each wide-area site and a Paxos-like protocol across wide-area sites. Steward requires $3f + 1$ replicas at each site to tolerate $f$ failures at any site. More generally, given the large number of RSMs and machines in our target environment, we believe the overhead for variations of hierarchical replication to be prohibitively high. In addition, performing consensus (especially Byzantine agreement) at each site adds latency to the critical path of RSM operations, whereas CORFU has been designed to use asynchronous updates to avoid additional latency.

Indeed, LocalRecovery implements a form of asynchronous hierarchical replication. A more traditional design assigns each data center a set of servers that all act as replicas for the site. Whenever one of these servers receives a client request or cross-site consensus message, it runs consensus locally before responding. This design has the advantage of providing a very high degree of fault tolerance. The drawbacks are the increased latency of running consensus locally on every operation, as well as increased storage and processing costs for the additional replicas. With three wide-area sites, LocalRecovery would require six data copies, plus three switches, which maintain very little state in normal operation. Synchronous hierarchical replication would require nine copies for the same three wide-area sites. We felt that the modest additional complexity of CORFU represented an acceptable trade-off for not imposing additional latency for the critical path of operations and requiring less overall storage and processing. Because of these costs, we do not further consider synchronous hierarchical replication in our analysis or evaluation.

Cluster Services Recently, there has been significant interest in the appropriate system support for managing large-scale clusters. These efforts include MapReduce [47], BigTable [38], Chubby [31], Pig [84], and Dryad [60]. A number of contemporaneous efforts have attempted to ease the development of higher-level network services, such as filesystems or interactive web pages, by developing distributed data structures that scale across multiple nodes in a data center [55, 71].
In all cases, these efforts focus on distribution within a single data center/cluster, while we focus on wide-area replication. Further, we focus on delivering strong consistency while most of these other efforts are typically satisfied with more relaxed consistency.

**IP Takeover** IP address takeover [10] allows a host to assume the IP address of a failed host by broadcasting a gratuitous ARP on the network announcing that its MAC address is now responsible for the failed IP address. Hence, any new packets/TCP connections will be redirected to the new server.

**Group Communication Systems and CORBA** Recovery has been studied extensively in the context of group communication and CORBA. FT CORBA [85] standardize mechanisms such as a generic factory, a centralized controller and a fault monitoring architecture. Several frameworks [74, 75, 81, 90, 93] have been implemented that draw inspiration from the standard. These all use a centralized controller like Corfu, except [74] where the control mechanism is distributed among the replicas. Although both [74, 75] target wide-area replication, none of the above frameworks provide a local recovery mechanism like Corfu.

### 2.3 xOMB Related Work

Most closely related to our work in spirit are Click [79] and RouteBricks [49]. Click provides a modular programming interface for packet processing in extensible routers. The principal difference between Click and xOMB is our focus on extensibility and programmability at the granularity of byte streams rather than individual packets. Our pipelined programming model focuses on efficiently parsing and transforming request/response based communication.

Like xOMB, RouteBricks also focuses on scaling network processing with commodity servers. Their work, like Click, focuses on routing and operates at the granularity of packets. They focus on the more extreme performance requirements of large-scale routers that may require terabits/sec of aggregate communication bandwidth and their in-kernel implementation and VLB-based load balancing de-
livers significant scalability. Middleboxes typically do not require quite the same level of bandwidth performance and while our architecture similarly scales with additional servers, our user-level implementation trades per-server performance for programmability and overall extensibility.

Flowstream [54] also provides an architecture for middleboxes. It employs OpenFlow [12]-controlled hardware switches to separate traffic at flow granularity that is then forwarded to individual servers for further processing. By default these servers would perform network processing at packet granularity. As such, one could view xOMB as the architecture and programming model for extensible transformation of OpenFlow-forwarded byte streams in Flowstream.

There are many commercial hardware/software products for middlebox processing. F5 networks [8] provides popular load balancing switches. Pai et al. [87] performed some of the early academic work in this space. Bivio [4] focuses on deep packet inspection, while Riverbed [14] delivers protocol accelerators among other products. Each product typically focuses on a niche domain and provides a limited extensibility model. In particular, the F5 switch we evaluate uses the Tcl programming language. However, it is not able to support the full generality of our framework, for example with respect to making remote procedure calls or maintaining protocol-specific metadata and state. Certainly, the F5 could not be employed to implement functionality for an entirely different protocol such as the S3 service (Section 7.4). On the other hand, the goal of our work is to provide a unified framework and programming model for a range of traditional stovepipe middlebox functionality.

Reverse proxies such as [3, 11, 18, 20] aim to provide load balancing over a set of web servers. While this is similar in spirit to part of the functionality that xOMB supports, our architecture is much more general and can support arbitrary protocols. Most reverse proxies can only handle a static set of servers, protocols, and have fixed processing options, unlike xOMB which can handle dynamic membership and arbitrary processing. Nginx [11] is fairly extensible, although modules written for it must be compiled into the executable and not loaded dynamically like in xOMB. In addition, nginx does not support general connection collapsing
of client requests, which can severely impact performance with large numbers of clients. Finally, RPCs for making dynamic routing decisions cannot be done with their callback model, although they do support passing arbitrary state between callbacks like xOMB.

Allman performed an early performance study of middleboxes [22]. He found that middleboxes are a mixed bag for performance, increasing or reducing performance under different circumstances. The study also found that middleboxes can reduce end-to-end availability, though typically availability remained at an acceptable 99.9%. One goal of xOMB is to develop a framework to increase the performance and availability of middleboxes.

One challenge with middlebox deployment is ensuring that flows are forwarded through an appropriate set of middleboxes based on higher-level policy. Dilip et al. [61] introduced an architecture to ensure such forwarding. OpenFlow provides a general mechanism to intercept flows and forward them through an appropriate set of middleboxes on the way to the destination. Ethane/SANE [34] is one instance of such an approach for enterprise network security and authentication.

DOA [109] is a delegation oriented architecture for more explicitly integrating middleboxes into the Internet architecture. One goal of DOA is to address the transparency issues introduced by non-extensible hardware middleboxes on evolving network flows. xOMB would ideally make it easier for middleboxes to adapt to changing traffic characteristics. Similar to DOA, I3 [101] explicitly introduces indirection in data forwarding, this time at the overlay level using a DHT.

2.4 Acknowledgment

Chapter 2, in part, is currently being prepared for submission for publication of the material. Anderson, James; Rasmussen, Alexander; Braud, Ryan; Kapoor, Rishi; Meling, Hein; Madhyastha, Harsha; Marzullo, Keith; Vahdat, Amin. The dissertation author was the primary investigator and author of this material.
Chapter 3

Consistency Costs and Consensus

Applications and infrastructure with a global audience increasingly rely on stateful services hosted in data centers. These services must be replicated at multiple geographic locations to deliver target levels of availability and performance to all clients. Many of these services require the semantics of strong consistency; however, variations of weak consistency remain the state of the art for wide-area replication. In this chapter, we examine the fundamental costs of strong consistency for various protocols and storage configurations and present a detailed wide-area performance breakdown and analysis.

Recognizing the need for wide-area replication, services must determine how their data should be replicated. Replication of services that do not share updates across their user populations is relatively straightforward. One approach is to simply place full replicas of the service at multiple data centers. Of course, underlying service state (e.g., the inverted index for an Internet search application) must be periodically synchronized, but this operation does not lie in the critical path for individual user operations.

Replication of services that accept globally-visible updates from users is more difficult since user operations can be dramatically affected by the consistency guarantees that the replication system provides. Two extremes of the consistency spectrum are weak and strong consistency. By weak consistency, we refer to any protocol that uses asynchronous replication to make a best effort of eventually delivering all updates to all replicas. By strong consistency, we refer to any protocol
that provides at least sequential consistency (a total ordering of reads and writes) and thereby necessitates synchronous updates. Today’s multi-data center storage services all provide some form of weakly-consistent replication [43, 48, 51].

Despite being the de facto standard, weak consistency has considerable drawbacks. Weakly-consistent protocols may lose updates, admit conflicting updates, or apply updates in different orders at different replicas. Previous systems vary in their methods for dealing with these shortcomings. None of these solutions are entirely satisfactory for those systems that require strong consistency.

We believe that for network services that require or benefit from strong consistency, the disadvantages of weak consistency are sufficiently severe that a re-examination of the fundamental costs of strong consistency is warranted. In the next chapter, we re-examine the conventional wisdom that strongly-consistent replication is infeasible in a wide-area setting. We then describe PAXOS, a consensus protocol commonly used as the basis for strongly consistent replicated systems. Finally, we present a detailed experimental analysis of the fundamental costs of strongly-consistent replication.

### 3.1 Costs of Strong Consistency

Conventional wisdom in replicated service design says that strong consistency should be avoided for two primary reasons. First, it is held that strongly-consistent protocols are too expensive in terms of latency and are therefore unable to provide acceptable application performance.

Second, the CAP principle [53] states that strong consistency, high availability, and partition tolerance cannot be achieved simultaneously. Service designers have traditionally opted for high availability and partition tolerance at the expense of strong consistency because this trade-off further increases availability by allowing a partitioned replica to continue serving clients.

It is important to note that the total amount of communication for strongly- and weakly-consistent protocols will by necessity be roughly equivalent. A weakly-consistent protocol may have a slight advantage if it can defer transferring some
updates to perform batch writes in a particular time window, but fundamentally both strongly- and weakly-consistent protocols must copy the same number of bytes to other replicas.

Therefore, the actual performance cost of strong consistency comes primarily from the additional synchronous latency necessary to run the required consistency protocol. State-of-the-art protocols can be engineered to impose only a single round trip delay between two sites (assuming a replication factor of three). Furthermore, these protocols pipeline requests to obtain high throughput.

In addition, web applications must already tolerate higher latency because clients already make requests across the Internet, possibly involving many different services and servers. Empirical evidence suggests that the average latency of a request to Amazon’s S3 can range from 300 to 700ms depending on the location of the client making the request [86]. Assuming a minimum RTT of 30ms across North America, the key question becomes whether services that require strong consistency can tolerate this additional latency and still meet their SLAs.

In order to determine whether it is practical or indeed possible to meet application SLAs and still provide strong consistency, a comprehensive understanding of the fundamental costs associated with strongly-consistent replication is required. In this chapter we endeavor to provide such an understanding through a detailed analysis of the performance costs of a service proving strongly consistent replication.

We also re-examine the cost of strong consistency from the perspective of the CAP principle. If we assume strong consistency, then the CAP principle states that we face a trade-off between high availability and partition tolerance. While this trade-off is inherent, we believe that recent trends in web service and cloud computing architectures mitigate this concern. Traditionally, the client updating a data item would be co-located with one of the replicas and that replica would be responsible for updating a set of other replicas. With web services, clients and replicas are disjoint. Hence, a partition preventing client access to one replica need not prevent the update from proceeding as long as the client can reach an alternate replica. Further, with modern consensus protocols [67], updates need
only be applied to a quorum of replicas. Hence, we have moved from a model where any partition among replicas would reduce availability to a model where a partition among a quorum of replicas is required to reduce availability.

### 3.2 PAXOS Overview

Strong consistency among a set of replicas is achieved by using a *consensus* algorithm or protocol. One of the most popular consensus algorithms, and the one used in this work, is PAXOS [67]. The PAXOS algorithm allows a set of nodes to propose values, and provides the guarantee that at most one value will ever be chosen. Further, if a value is chosen, then all correct nodes will agree on the same chosen value. With $2f + 1$ replicas, PAXOS can maintain liveness with up to $f$ crash-failures.

The simple PAXOS algorithm runs in three phases, shown in Figure 3.1. First, a proposer $A$ collects a quorum of PROMISE messages in response to a PREPARE. Second, the proposer sends ACCEPT messages with either a value returned in a PROMISE response or with an unconstrained value otherwise. A value is *chosen* when it has been *accepted* by a quorum of replicas. Finally, every node that receives an ACCEPT message broadcasts LEARN messages to all other nodes. When a node has received a quorum of learn messages, it knows the value has been chosen.

As an optimization, PAXOS also supports a leader to avoid the first phase after the leader has been chosen, illustrated in Figure 3.2. The leader $A$ can immediately propose a value by sending ACCEPTLEADER messages. The other nodes respond with ACCEPTED messages. Once the leader has a quorum of ACCEPTED messages, it broadcasts a CHOSEN message to notify the other nodes of the chosen value.
Figure 3.1: Simple PAXOS messages

Figure 3.2: PAXOS messages with leader
3.3  PAXOS Implementation

While the Paxos algorithm itself can be expressed very concisely, we encoun-
tered a significant gap between the formal algorithm and an actual implementa-
tion. This gap is largely due to a lack of specification for both interfaces and function-
ality. In this section, we describe in detail our full-featured Paxos implementation
in the hope of explaining what is required for a functional Paxos deployment.

3.3.1 Errors and Failures

We consider a node to have failed when it has left the system and will
never return, e.g. due to its disk crashing. Nodes that are unreachable for a
period of time, e.g. due to a network error, are not considered to be failed, merely
temporarily unavailable. Our PAXOS implementation running with $2f + 1$ replicas
will operate normally as long as $f$ or fewer nodes have failed or are disconnected,
and none of those $f$ nodes are the leader. If fewer than $f$ nodes are unavailable,
but one of those nodes is the leader, then our implementation reverts to running
three-round PAXOS for every proposal, either until the designated leader becomes
available again or a new view is chosen with a live designated leader. If a non-
leader replica attempts to submit a proposal to the leader and receives a network
error, then it immediately proposes it using three-round PAXOS, without waiting
for the proposal to time-out. This optimization allows our PAXOS implementation
to maintain steady throughput with an unreachable or failed leader.

3.3.2 Indexed Consensus

Because a single instance of the PAXOS algorithm only allows one value to
ever be chosen, an application that wants consensus for multiple operations must
have a way of running separate instances of the PAXOS algorithm. Our approach
is to treat a series of logically related operations as a log. We assign each operation
a consecutive log index. By running one instance of PAXOS for every log position,
all nodes will agree on both the operation for any given index, as well as the total
ordering of all operations.
If there is only a single writer, then Indexed Consensus can deliver high throughput by pipelining multiple writes at multiple log positions. However, if writes originate near simultaneously from multiple sources, performance may suffer because each update may have to be proposed multiple times before succeeding.

We also provide another interface, called Serial Consensus, in which proposers do not explicitly specify an index, but rather allow the service to assign the next available index. This is accomplished by having the leader assign indices for all nodes. When a leader assigns log positions, all nodes forward operations to the leader, which then assigns the operations consecutive log indexes and proposes them. In the absence of failures, there is never contention for a log index and all proposals are chosen immediately. This ensures steady throughput, even when multiple writers are issuing simultaneous operations. Our implementation supports an interface in which proposals are forwarded to the leader to ensure high throughput without contention.

We allow the application to determine whether Indexed Consensus or Serial Consensus is more appropriate for its requirements. In the future, it may be possible to automatically switch between different versions depending on observed access patterns.

### 3.3.3 Interface

Our PAXOS service request interface provides two methods for proposing values. The first method, `propose`, takes as arguments the `log_id`, the desired index—which may either be an absolute position or the special values `NEXT_INDEX`, meaning the next available index, or `NEW_INDEX`, meaning the next index for which this node has not already issued a proposal—and a buffer containing the proposed value. The second method, `proposeMembers`, takes the same arguments as `propose`, but additionally takes node ids for a leader and set of cohorts that will form the view for that PAXOS log starting from the specified index. Unlike other PAXOS implementations [70], which require the leader to make progress and thus have the internal PAXOS protocol elect new leaders when the old leader has failed, our implementation requires the service using PAXOS explicitly name the leader,
and that PAXOS guarantees not to change the leader unless requested. However, we have designed our service such that it may continue to make progress even if the designated leader is unavailable.

Our approach has the benefit of allowing higher layers to optimize the way they issue requests, e.g. by directing clients to the leader or placing the leader on a lightly loaded host, to increase performance, which would not be possible if the leader were managed by PAXOS. The interface has a third method, markApplied, which tells PAXOS that the application has committed all proposals up to and including the specified index into stable storage. PAXOS may free the state associated with a proposal once all replicas have applied that index.

In response to every proposal request, the PAXOS service guarantees that it will eventually make an upcall with one of three results: 1) that the proposed value/view was chosen, 2) that another proposed value/view was chosen for the requested index or a view change specifying different replicas occurred before the requested index, thus invalidating the proposal, or 3) the proposed value/view failed due to too many network errors attempting to contact other replicas. This interface allows the service using PAXOS complete control over how to respond to the outcome of any given proposal. If a proposal is not chosen due to another proposal being chosen or a view change, the application may either re-propose the same value/view at a new index, or decide it is no longer relevant and discard the proposal. If a proposal ever fails due to network errors, then application may decide to initiate a view change to use different replicas, or it may set a timer and try its proposal again later.

A new PAXOS log begins at index 0 with a view proposal, which we describe in detail below. Once the log has been established, our PAXOS service imposes no limitations on which indexes the application may propose, other than that a replica must propose an index greater than its applied index. This allows an application to have multiple outstanding proposals for both the same index and different indexes. Unlike some other PAXOS implementations [70], which allow only a single outstanding proposal or a fixed $\alpha$ number of proposals, our implementation allows an arbitrary number of outstanding proposals, meaning that an application
may use deep pipelining of requests to increase throughput. Allowing proposals to be made for arbitrary future indices introduces the possibility of safety violations when view changes occur, but our implementation prevents this from happening, as described below.

Although we allow an application to propose a value at an arbitrary index (higher than its applied index), our PAXOS service guarantees that it will notify chosen proposal results in consecutive order. Thus if the last notified proposal is at index \( n \), then if a node learns that a proposal has been chosen for index \( n + k \), it will not notify the application of the proposal until it has learned and notified proposals \( n+1, n+2...n+k-1 \). Unlike some other PAXOS implementations, which need to use Null proposals to fill in gaps between proposals after a new leader is elected, our implementation leaves ensuring progress up to the application. Thus, an application may always propose a new value at the next un-chosen index, which may be a “Null” value for the application, or it may be another proposal value. Either way, we simplify application logic by guaranteeing that every index will be filled by an application proposed value, and that all chosen values will be notified sequentially.

### 3.3.4 View Establishment

We call the creation of a new PAXOS log view establishment. An application using our PAXOS services creates a new log by issuing a view proposal for index 0. Existing PAXOS deployments use either a static configuration or rely on an operator to verify that the creation of the replica group succeeded, and if it did not, then to manually adjust the replicas and retry. Because we designed our PAXOS service for dynamic services that will create many PAXOS logs on-demand, we require a view establishment protocol that meets PAXOS safety properties while avoiding any manual configuration.

To ensure that our protocol satisfies the PAXOS safety property, we ensure that view establishment achieves unanimous agreement among the PAXOS replicas. To understand why, consider the following scenario that assumes only a quorum of replicas are needed to establish a view. The metadata servers agree on data server
replicas $ABC$ as view $v_{abc}$. $A$ and $B$ accept the view proposal, so $v_{abc}$ is established but $C$ has not learned the proposal yet. Then, the failure detector declares $A$ dead and that it should be replaced with $D$ in view $v_{bcd}$. It is now possible for replicas $C$ and $D$ to establish $v_{bcd}$ as $BCD$, while replica $B$ has an established view of $ABC$. Thus, quorum agreement will not satisfy safety for view establishment.

With unanimous agreement required for view establishment, then either $ABC$ have all accepted the view $v_{abc}$, in which case $v_{bcd}$ will not be accepted for index 0, but will get re-proposed as a view change at a subsequent index, or $A$ fails before accepting $v_{abc}$, ensuring that $v_{abc}$ can never be chosen, and thus $v_{bcd}$ can establish the view for index 0.

However, the unanimous agreement requirement may cause Paxos to not achieve liveness. Consider again the scenario described above. If replica $B$ accepts $v_{abc}$ before $A$ fails but $C$ does not accept the proposal, then the traditional Paxos algorithm would prevent any view from ever being established, because when the proposer attempts to get a promise from $B$ for $v_{bcd}$, $B$ would respond with its already accepted proposal for view $v_{abc}$, which the proposer would then be obligated to propose instead. But because $A$ failed, that view can never be chosen. We solve this problem by giving our Paxos service access to a strong failure detector. If the failure detector declares a node to have failed, then that node may never return again. We ensure this by having nodes contact the failure detector when they initialize, before responding to any messages, and if the failure detector thought a node has failed, then it terminates. We then modify the acceptance logic for view establishment in Paxos to check if any of the nodes have failed for the views of any promises containing acceptances, and if so, allow the proposing node to ignore those acceptances. This allows the proposing node to propose a view that can actually be chosen, while not violating safety.

### 3.3.5 View Changes

At any point after a view has been established and the Paxos log is created, an application using the Paxos service may change the replicas in the log or designate a new leader by issuing a view proposal request. We require that the
new view have an overlapping quorum of replicas with the old view. With view changes that are not establishing a PAXOS log, our PAXOS implementation does not require unanimous agreement, but does require that a quorum of the nodes in the intersection of the old view and new view have accepted the view proposal, a necessary and sufficient condition to maintain safety.

When a view change adds a replica to a PAXOS log, that node is only responsible for proposals chosen after the view change index. Because our PAXOS implementation allows arbitrary pipelining of outstanding proposals, we must use special handling for proposals chosen out-of-order to ensure safety. When a proposal is chosen for an index greater than the last chosen index, we place the proposal in a pending data structure, which is stored persistently along with the rest of the PAXOS state and log. When a node learns the proposals for all indexes less than the pending index, if none of the proposals are view changes, then the node commits the pending proposal into the log, treating it as a normal chosen proposal. However, when choosing a view change, if the view change changes the replica set (as opposed to just changing the leader), then all out-of-order proposals must be erased. Further, all PAXOS state (promises, accepts, and learns) for all indices greater than the view change must also be updated to reflect the new view, meaning that all state with a view less than the old view is erased. If we did not take these actions, then a proposal that was chosen in the old view (say, by A and B) could conflict with a different proposal chosen in the new view (by C and D), thus violating safety.

3.4 Storing Service State

How replica data is stored—whether in memory or on disk either synchronously or asynchronously—influences the performance, availability, capacity, scalability, and cost of a replicated service. In this section, we describe the primary tables a strongly-consistent replicated service must maintain and the possibilities for backing stores. Although we recognize that solid-state drives and other forms of emerging persistent memory offer even more choices in this design space, we
limit our discussion to conventional hard disks and memory.

We designed the Paxos services to transparently use either in-memory C++ STL maps or file-backed Berkeley DBs to store their respective service state. The important tables used in Paxos are summarized in Table 3.1. paxosdb stores Paxos state messages for outstanding proposals, and stores as many entries per object as the number of outstanding proposals for that object. This table will be empty in a quiescent system. logdb stores the state of each object’s metadata and data store logs. chosendb stores the chosen proposals for each object. There is one entry in this table for each outstanding proposal for each object; in a quiescent system, this table will be empty.

To maintain correctness across crashes, all four of the tables must be synchronously written to disk. As we show in §5.4.4, storing all of the tables on disk causes Paxos to bottleneck on disk writes and therefore has a significant impact on overall system throughput. Our evaluation shows that even when enabling a large BerkeleyDB cache, an in-memory write-ahead log, and asynchronous disk writes, using disk-backed storage for the tables still results in greatly decreased throughput relative to using in-memory maps.

For many services, keeping state intact from a machine crash is not necessary. If a machine fails for any reason, another server will be reassigned to store its objects. Thus, we consider other storage configurations besides synchronous disk writes. While storing all service state in memory yields the fastest performance, it would be impractical for most deployment scenarios because it limits the aggregate capacity of all Paxos objects to the aggregate memory capacity of the servers. Thus, for our default storage configuration, we use a configura-

<table>
<thead>
<tr>
<th>Name</th>
<th>Mapping Stored</th>
<th>Backing</th>
</tr>
</thead>
<tbody>
<tr>
<td>valuedb</td>
<td>[object name, key] → value</td>
<td>BDB async</td>
</tr>
<tr>
<td>paxosdb</td>
<td>[log key, index] → Paxos state</td>
<td>STL Map</td>
</tr>
<tr>
<td>logdb</td>
<td>log key → log state</td>
<td>BDB async</td>
</tr>
<tr>
<td>chosendb</td>
<td>[log key, index] → proposals</td>
<td>STL Map</td>
</tr>
</tbody>
</table>
tion that offers good performance while still allowing the aggregate capacity to be bounded by disk size, not memory. Specifically, we store the valuedb and logdb tables in a file-backed BerkeleyDB with asynchronous disk writes and an in-memory write-ahead log because these are the only tables that actually grow as either the number of objects or the number of values per object increases. The other tables—paxosdb, choosebd—are backed by in-memory maps, since PAXOS frees state associated with a request once it has been processed. While this design limits the total number of simultaneous requests that may be in progress to available memory, we show in §5.4 that, in practice, memory capacity will likely never be the bottleneck resource for PAXOS.

3.5 Evaluation

To better characterize the costs of strong consistency, we present a detailed performance analysis of PAXOS's PAXOS subsystem. We then examine microbenchmarks showing how PAXOS performs for various latency, throughput, and scalability tests.

3.5.1 Experimental Setup

We use ModelNet [107] to simulate dedicated wide-area links between a set of data centers by passing all wide-area traffic through a ModelNet core running on a dedicated machine. For most of our experiments, we use a configuration with three data centers of ten machines each connected in a ring by 15Mbps links with latencies of 20ms, 50ms, and 70ms. We use different topologies for the scalability experiments with data centers of five machines each connected in a ring with 15Mbps links with uniform 20ms latencies. Up to 60 clients are multiplexed evenly on four machines. Clients communicate with their local data center over a 15Mbps, 3.5ms link. We have configured ModelNet so that intra-data center traffic uses a physical non-blocking Gigabit Ethernet switch rather than being routed through the ModelNet core. Each machine in the cluster has a 2.13GHz quad-core Intel Xeon processor, 4GB of RAM, and a 1Gbps Ethernet adapter.
3.5.2 PAXOS Evaluation

To better understand the processing costs required by strong consistency, we instrumented our PAXOS implementation to record precise timing information for every method call. To minimize profiling overhead, we record the data in memory and write to a log file at the end of the experiment. The experiment, deployed on our standard three data center topology, has a single client connected to data center A (outgoing latencies of 20ms to B and 50ms to C) send sequential write requests with one-byte values to unique keys. We computed these results by averaging the times from 100 write requests, using our standard storage configuration (§3.4) with asynchronous disk writes.

We first show in Figure 3.3 the average latency measured for a request, broken down by network and processing events. This shows the PAXOS protocol when the client directly communicates with the leader, replica A. The client first submits a request to A (1). A executes Phase I of PAXOS by sending ACCEPTLEADER messages to replicas B and C in (2.1 and 2.2). B receives this message first and responds with the Phase II ACCEPTED message (3.1). The proposal has now been chosen because a quorum of replicas have accepted it. When A receives B’s ACCEPTED message, it commits the proposal to its log and sends Phase III CHOSEN messages to B and C (4a.1 and 4a.2). It then notifies the data store service that the update has been chosen, and A responds with success to the client (4b). Next, replica C receives A’s ACCEPTLEADER, and responds with an ACCEPTED message (3.2). Replicas B and C receive A’s CHOSEN messages, and update their logs.
and data store state accordingly. Finally, A receives C’s ACCEPTED message, but ignores it because the proposal has already been chosen.

Most of the latency of a PAXOS operation running across the wide-area is dominated by wide-area network latency. However, overall throughput is limited by CPU processing in Phases I, II, and III of the PAXOS protocol. We describe in detail the processing involved for each of these three phases, accompanied by the timelines in Figures 3.4a, 3.4b, and 3.4c.

**Figure 3.4**: Breakdown of CPU spent executing the three phases of PAXOS as atomic non-blocking events.

**Phase I: PROPOSE and ACCEPTLEADER**

Consensus begins by proposing a value at an index at the leader, which spends $221\mu s$ in the consensus service. $39\mu s$ is spent validating the request and computing the next available index in the log ($22\mu s$ to retrieve the metadata log record). The service spends $24\mu s$ preparing the proposal, which includes constructing a proposal object, scheduling a timer for the proposal and recording state for the proposal.
The leader then spends 103µs to process the proposal. It first validates that the proposal is in the current view for the log, that a value has not already been chosen for the requested index, that it has not already issued accept messages, made any promises, accepted any proposals, or received any learns for the requested index. The leader records its promise in the PAXOS state, records the proposal in a table, and constructs an ACCEPTLEADER message containing the proposal and a hash fingerprint (7µs) of the proposal. Finally, the service spends 32µs and 21µs to send the ACCEPTLEADER message to the cohorts, which includes the time spent in the kernel writing data to the socket buffer. The leader waits for the first response from one of the cohorts before proceeding.

**Phase II: Accepted**

Each cohort spends 137µs processing the AcceptLeader message. First, the service spends 18µs deserializing the message and dispatching it to the correct event handler. Next, the node must retrieve its copy of the replicated log metadata (26µs) and check that the proposal is in the current view and that it does not already have a chosen proposal for that index. The service then spends 5µs validating that the hash fingerprint matches the proposal payload. The service then retrieves the PAXOS state for that index and verifies that it has no outstanding promises, acceptances, or learns. Finally, the node stores the proposal as a promise and an acceptance in the PAXOS state (7µs) and sends a response to the leader with an ACCEPTED message containing the proposal fingerprint, which takes 45µs.

**Phase III: Chosen**

When the leader receives the first response, it can complete that round of consensus, spending 232µs as follows. The leader spends 70µs to check and update its state, including validating the response and computing that it has a quorum of acceptances (23µs to get the log metadata from the table). The service spends 24µs to update the applied index for the cohort, which allows efficient replicated log garbage collection. The leader then updates its own PAXOS state to conclude that it has a quorum of Accepted responses (28µs) and then then spends 34µs and
20\mu s to send Chosen messages to the cohorts. Finally 43\mu s is spent committing the proposal to the replicated log and freeing all the Paxos and proposer state associated with the index, including 11\mu s to write the log to the table. As part of the commit process, the node makes a callback to the data store service informing it that a request has been stored in the log at the given index.

Analysis

The total average time the consensus service spends processing a request is 535\mu s. The total measured time spent in servers along the critical path processing the request is 750\mu s. The difference between these values is the time spent in the data store service, which must update its own state and execute the logic to perform the requested operation. In addition, the software switches in each data center spent a total of 312\mu s processing the request along the critical path (which includes six separate events), and the client spent 130\mu s sending the request and handling the response. The total average time spent processing the request among all these components is 1192\mu s.

We see that when the leader sends two messages sequentially, such as the AcceptLeader or Chosen messages, that the first send uniformly takes longer than the second. The first send incurs a cost of serializing the message into a byte stream, but the second send uses the previously serialized copy.

Although our Paxos implementation already incorporates many optimizations and has been highly tuned, we briefly consider possible optimizations to further reduce this approximately 1ms of processing time. One opportunity to reduce the processing cost is to use a faster storage mechanism for the persistent Paxos logdb and chosenb tables. BerkeleyDB exhibits much more overhead than STL in-memory maps, even with asynchronous writes and in-memory logging.

We note a possible optimization of eliminating the Phase III messages when using both a leader and only three replicas. The AcceptLeader includes an implicit Accepted, so each of replicas B and C could conclude that the proposal has been chosen when they receive the AcceptLeader. This optimization reduces the latency for non-leader replicas by two message delays. Even with this opti-
Table 3.2: Per-replica overhead for different write sizes (rounded to nearest kilobyte for readability)

<table>
<thead>
<tr>
<th>Write Size</th>
<th>Client</th>
<th></th>
<th></th>
<th>Leader</th>
<th></th>
<th></th>
<th>Cohort</th>
<th></th>
<th></th>
<th>Per-Replica</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Written</td>
<td>Read</td>
<td>Written</td>
<td>Read</td>
<td>Written</td>
<td>Read</td>
<td>Overhead</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1B</td>
<td>2528KB</td>
<td>712KB</td>
<td>11487KB</td>
<td>4680KB</td>
<td>1076KB</td>
<td>5383KB</td>
<td>6450KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1kB</td>
<td>12284KB</td>
<td>712KB</td>
<td>30998KB</td>
<td>14436KB</td>
<td>1076KB</td>
<td>15139KB</td>
<td>6450KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10kB</td>
<td>100175KB</td>
<td>712KB</td>
<td>206780KB</td>
<td>102327KB</td>
<td>1076KB</td>
<td>103030KB</td>
<td>6450KB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As other work [73] has noted, the leader requires more processing than the other replicas. We could implement a rotating-leader algorithm to improve performance if we expect to receive update requests at all replicas. Whether this would actually improve performance is highly application specific. Using our fixed leader implementation, we see that the leader spends 592μs processing a request. In a purely single threaded application, we would expect the CPU of the leader to be the bottleneck for small requests, and that the server should be able to handle 1689 updates per second. Indeed, as we show in §5.4.3, PAXOS can achieve a peak throughput of 1925 updates per second, which we attain because of parallelism in the transport service (the quad-core machines showed an average CPU utilization of 120-160%). While using a fully multi-threaded PAXOS implementation would not reduce the latency or increase the throughput for a given log, it would allow for a potential increase in aggregate throughput across multiple logs.

### 3.5.3 Network Bandwidth Overhead

We next measure the network requirements for strong consistency in terms of bytes transferred. We have a client submit 10,000 WRITE operations of varying size and measure the bytes sent and received by the leader and other replicas. Table 3.2 shows the data transferred and overhead per replica. Subtracting the client payload, the leader sends 551 bytes per request per replica, and each replica sends 110 bytes per request. Thus, the total per replica overhead of PAXOS is 651
bytes per request, which is independent of the client request payload size.

3.6 Summary

In this chapter, we have argued that in a cloud computing setting, the costs of strong consistency are acceptable for many applications. The additional network overhead can be reduced to one wide-area round trip time; with appropriately chosen data center sites (e.g., RTT on a dedicated circuit for sites on the same continent could be on the order of 50-100ms), it then becomes possible to fit at least one strongly consistent write in the SLA budget (e.g., 100-500ms) of many interactive web services. When factoring out network latency, a strongly consistent write applied to three data centers imposes approximately 1ms of latency, with less than 600 $\mu$s on the critical path determining overall service throughput. It is our hope that our experience with PAXOS can both inform construction of wide-area storage services and encourage certain network applications to leverage strong consistency, trading quantified increase in latency for application simplicity and improved semantics. The following chapters will describe services and systems built using our PAXOS implementation.

3.7 Acknowledgment

Chapter 3, in part, is currently being prepared for submission for publication of the material. Anderson, James; Rasmussen, Alexander; Vahdat, Amin. The dissertation author was the primary investigator and author of this material.
Chapter 4

ZANTE

In Chapter 3, we described the Paxos consensus algorithm that can be used to build strongly consistent services based on a consistent replicated log. However, replicating such services across multiple wide-area sites is a challenging task, requiring both overcoming challenges of naming and scalability as well as careful integration over a range of state-of-the-art techniques. Transparently and consistently replicating wide-area services is currently not possible.

Traditionally, when consistency protocols run within a single site, the member replicas are named directly (i.e., by IP address). With multiple data centers using thousands of machines to host replicas, naming member replicas directly severely limits scalability by requiring replicas in one data center to know the specific replica servers in the remote data centers. Furthermore, the services traditionally using strong consistency tend to be deployed on a small scale, so the initial replica membership (and often changes to the replica set) is manually configured. Cloud-scale services, however, may have many thousands of replica groups, and require that new replica groups can be created automatically and consistently.

In this chapter, we present ZANTE, a system for building strongly consistent applications and services based on scalable replicated state machines designed for fine-grained wide-area cross-data center replication. The underlying storage is replicated at multiple data centers for read locality, high availability, and disaster recovery. ZANTE overcomes the challenges of replica naming and automatic replica group creation by decoupling consistent metadata from the replica data
and through *dynamic delayed replica binding*, a new technique that abstracts individual replicas outside of replica sites. For applications with less strict consistency requirements, ZANTE also provides weak consistency, which can be chosen on a per-object basis. By layering weak consistency on top of the baseline strong consistency service, ZANTE simplifies the reasoning regarding the semantics of conflicting updates.

### 4.1 Overview

The goal of this work is to define a wide-area replicated storage abstraction general to the requirements of a range of applications. Services should ideally be written as if storage were confined to a single data center, but the service should be able to transparently reap the locality and fault tolerance benefits of wide-area replication.

We begin with a sample ZANTE deployment scenario of six data centers
shown in Figure 4.1. These data centers may be deployed as a “private cloud” for one company’s infrastructure or as a “public cloud” hosting third party applications [23]. Data centers are interconnected with dedicated, high-bandwidth leased circuits, often allocated in increments of 10 Gb/s [40]. Top tier service providers can have multiple such circuits between individual data centers. While we depict a ring of sites for simplicity, the interconnecting topology can be arbitrary. Internally, each data center may contain thousands of machines. A switching service in each of each data center directs individual messages to back-end machines.

ZANTE stores every map object and its associated metadata at multiple data centers. Different deployments may use different number of replicas to meet performance, availability, and cost requirements; we assume three replicas throughout this paper. ZANTE stores metadata and data in separate services for scalability and performance, so metadata and data will typically be stored on different replica sets as shown in Figure 4.1. Clients may communicate with any replica for metadata or data requests; ZANTE directs clients to the data center that will provide them with the best performance.

4.1.1 Consistency

ZANTE is designed for a cloud service model, where clients submit read or write requests to only a single replica. ZANTE uses a consistency protocol to update the other replicas. ZANTE offers four consistency models—eventual (weak), sequential (writes only), timeline [43], and sequential read/write consistency. Clients select a consistency model on a per-object basis when they create an object.

ZANTE offers different consistency models to allow clients to trade-off consistency for performance and availability, depending on their requirements. Every map object, regardless of consistency, uses its own underlying asynchronous consistent replicated log, which provides a total order for submitted updates using consensus. ZANTE replicas use the replicated log to update a persistent state snapshot as replicated state machines (RSM). When the replicated log has chosen an ordering for an update, each replica applies the update to its state snapshot.
Every update increases a monotonic version number associated with each object.

Although every map object has its own replicated log, the object consistency determines when Zante responds to different requests. For sequential consistency, a replica responds to reads immediately from its state snapshot. The replica submits updates to the replicated log and responds to the client only when consensus completes and the update has been applied. Timeline consistency is the same as sequential consistency, except that the client provides a version number for reads, and the replica will not respond until its state snapshot at least equals the requested version.

Zante layers both sequential R/W and weak consistency on top of the baseline implementation for sequential consistency. For sequential R/W consistency, rather than responding immediately to reads, Zante submits the read to the replicated log and responds with the latest value only when the replicated log returns. Thus, if two different replicas process simultaneous updates, and then each receive a read request, both reads will see both updates. In contrast, with sequential consistency, each read would be guaranteed to see the update submitted to its replica, but not necessarily the other update.

For weak consistency, a replica that receives an update first inserts the update into a cache and then immediately returns a successful response. The replica then asynchronously submits the update to the replicated log. Subsequent reads go to the cache first to return the latest updates. Once the consensus operation completes, the replica service applies the update to its state snapshot and removes the cache entry, meaning that all subsequent reads will view the now-consistent version of the data.

An important invariant of our approach is that all replicas have the same persistent state snapshot, even under weak consistency. Further, we do not need an explicit conflict resolution mechanism as conflicts are resolved by the serialization order assigned by the replicated log, e.g., “last” write wins. Of course, application-specific mechanisms may be required to handle a cache read that returns a different result than what would have been returned had the system waited for consensus to complete.
### Table 4.1: Summary of availability and performance requirements. *Quorum* specifies that the operation must reach a majority of the replicas before it succeeds. The latency specifies whether the client waits for only its local replica or wide-area consensus.

<table>
<thead>
<tr>
<th>Consistency</th>
<th>Replicas</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read</td>
<td>Update</td>
</tr>
<tr>
<td>Eventual</td>
<td>Any</td>
<td>Any</td>
</tr>
<tr>
<td>Sequential (W)</td>
<td>Any</td>
<td>Quorum</td>
</tr>
<tr>
<td>Timeline</td>
<td>Varies</td>
<td>Quorum</td>
</tr>
<tr>
<td>Sequential (R/W)</td>
<td>Quorum</td>
<td>Quorum</td>
</tr>
</tbody>
</table>

Table 4.1 summarizes the connectivity requirements and latency trade-offs for the various consistency models. In the rest of the paper, when we refer to “sequential” consistency, we mean sequential writes-only, unless otherwise specified.

#### 4.2 Replication Challenges

Protocols providing strong consistency have traditionally been deployed in a single site, in which all the nodes communicate over the same local area network [30, 37, 72]. The replicas for these systems run on a relatively small number of machines, they can be configured manually, and the replicas can be named with their unique network addresses. When a replica fails and loses its state, the replica group membership can be reconfigured either manually or automatically [70] with a new replacement replica. If the replicas use an automated failure detector, they can monitor their peers directly.

In contrast, deployments for cloud-scale services may span many data centers and operate tens-of-thousands of machines. At this scale, naming replicas by their network address becomes impractical for several reasons. For instance, different data centers may use same the non-routable IP addresses for private net-
works in different data centers. Doing so would cause ambiguity in the naming of replicas, so routers could not correctly determine the correct destination data center for consensus traffic.

Consider the problem of selecting members for a new replica group composed of $N$ machines from $N$ data centers. The service would like to assign a machine with a certain amount of free disk capacity at each site. One possibility would be to require the site coordinating the replica group creation have either global knowledge about all available machines at every data center site, including specific data about their resource utilization. Establishing this global database and maintaining it with current data would require considerable bandwidth and overhead, especially if the database is itself replicated at every site so that availability for creating new replicas is not constrained to any particular site(s). An alternative possibility is to have each site locally maintain such a machine database and have the coordinating site query each remote site for a specific machine. This approach is more scalable, but contacting remote replicas adds potentially large additional latency for creating a new replica group.

The problem of choosing replicas becomes yet more complicated when replica groups must be created dynamic, in response to a client request. In particular, the system must guarantee that if two separate clients attempt to create an object with the same name at the same time, and send their requests to different replica sites, that the object is only created once. Preventing this could be achieved with some form of centralized control, which would potentially add delay to requests and may not be scalable to large numbers of simultaneous operations. To solve this in a more distributed fashion requires that there be both some a quorum overlap between the replicas chosen for the two different objects and a protocol capable of recognizing the create conflict.

### 4.2.1 Dynamic Delayed Replica Binding

ZANTE overcomes these challenges with a novel technique for strongly consistent replication protocols: dynamic delayed replica binding. Instead of naming replicas by IP or hostnames, replicas are named by the data center site in which
they are located. A distributed software switching service (Chapter 7 describes this service in detail) operating as the front-end for the data centers parses the object name from requests and forwards the message to the correct replica machine within the site. When the switching service encounters unrecognized object names, it coordinates with a server allocation service to dynamically bind the object to a local replica. All subsequent requests for the object will be consistently forwarded to the correct replica. This binding happens on demand in the context of a request—the request gets processed normally once the binding has been dynamically assigned.

In the next section, we describe in detail how the ZANTE component services work together to provide a scalable solution for wide-area strongly consistent replication.

### 4.3 Design Overview

We designed ZANTE as a collection of interconnected services to meet our goals of scalability, high availability with variable per-object consistency, and performance. Table 4.2 provides a brief summary of these services, and Figure 4.2 shows their composition.
Table 4.2: Summary of ZANTE services

<table>
<thead>
<tr>
<th>Service</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>locator</td>
<td>statically maps 160 bit key space to data center sites</td>
</tr>
<tr>
<td>placement</td>
<td>determines which sites to use as replicas</td>
</tr>
<tr>
<td>allocation</td>
<td>assigns objects to servers on a per-data center basis</td>
</tr>
<tr>
<td>switching</td>
<td>direct messages to appropriate servers</td>
</tr>
<tr>
<td>metadata</td>
<td>state machine to store object metadata</td>
</tr>
<tr>
<td>data store</td>
<td>state machine to store object contents</td>
</tr>
<tr>
<td>replicated log</td>
<td>per-object replication for RSM</td>
</tr>
<tr>
<td>failure detection</td>
<td>notifies when to begin recovery</td>
</tr>
</tbody>
</table>

4.4 Scalability

As a service targeting a global set of users, ZANTE must be scalable. Adding more machines, whether to an existing site or through new data centers, should yield an increase in operation throughput and capacity commensurate to the additional number of machines added. Meeting this goal requires an architecture that distributes load both among machines within a data center and among data centers in the network. This distribution must be accomplished without centralized components and with minimal information sharing between data centers. Also, internal data center information should be self-contained and the service should not require knowledge of rapidly-changing global state.

4.4.1 Distributing and Locating Objects

On creation, clients assign objects unique names. To allow ZANTE to scale to large numbers of objects, we must distribute the responsibility for locating the servers that store each object while ensuring that the failure of a small number of global components does not render any data unavailable because it cannot be located. We accomplish this with a hierarchical approach and decoupling the servers
that know information about an object—the metadata service—and the servers that actually store the contents of the object—the data store service. Chapter 5 describes one implementation of a data store service in detail.

The locator service, at the top of the hierarchy, maintains the data center sites storing the metadata for a given object. The service statically divides (not necessarily uniformly) a 160-bit key space among the data center sites [102]. Object names are hashed into a 160-bit identifier, determining the site responsible for the metadata for the corresponding object. To ensure high metadata availability, we replicate all metadata state across the owning site and its two successors in the ring. Metadata is always replicated using strong consistency, even if the object itself is only weakly consistent. Any metadata request may be directed to any of these three replicas. This allows clients or the locator service to locate the metadata using a very small amount of static state, namely the mapping from data centers to their key spaces.

In the second layer of the hierarchy, the metadata service tracks the sites that store the object contents. The metadata service handles object creation and deletion and ensures that only one object with a given name can exist at a time. ZANTE guarantees object uniqueness, even if different data centers receive simultaneous requests to create the same object. Three aspects of the ZANTE design work together to meet this requirement. First, the static mapping of object metadata based on the object name hash to data centers ensures that all metadata requests will go to the same set of data centers. Second, dynamic demand replica binding ensures that all requests for a given object sent to a given data center will be directed to the same replica server within that data center. Third, our PAXOS implementation, described in Chapter 3, is designed to allow dynamic replica group establishment even with simultaneous attempts from different replicas and ensures that exactly one replicated log will be created.

The metadata service responds to client OPEN requests with the data store replica site that will provide the highest expected performance. The metadata service also coordinates and monitors interactions with the data store service, such as creating or deleting state for an object, migrating replicas, or performing
recovery.

The switching and allocation services occupy the bottom of the hierarchy. The allocation service runs locally in each data center and assigns and maintains mappings from object names to specific servers for the replicas in its data centers, with separate mappings for metadata and data store replicas. All incoming traffic—whether from clients or remote data centers—passes through a distributed switching service that forwards messages, addressed by object name, to the appropriate local servers. The switching service only maintains a soft-state forwarding table, populated with entries from the allocation service when it sees unrecognized object names. Chapter 7 describes the switching service in detail.

We expect the data center sites to change infrequently and thus we assume that all data centers have accurate global group membership. Adding a new data center or permanently removing an existing one requires re-balancing some address metadata, but because we decouple metadata from data storage, we do not have to re-balance or move any stored object data unless doing so would provide a desirable increase in performance; we need only create new replicas for all the maps that had a replica stored at a retired data center to maintain the target replication level.

\section*{4.5 Availability and Consistency}

To provide these consistency models, we designed the metadata and data store services as replicated state machines (RSM) with a persistent \textit{state snapshot} that is updated with callbacks from the asynchronous consensus service providing a \textit{replicated log}, discussed in detail in Chapter 3. We focus on the data store service since the metadata service always uses sequential consistency. For sequential consistency, an up-to-date replica responds to reads immediately from its state snapshot, unless the client provides a version number to ensure timeline consistency \cite{43}. The replica submits updates to the consensus service and responds to the client only when consensus completes and the update has been applied.

We layer both sequential R/W and weak consistency on top of our baseline implementation for sequential consistency. For sequential R/W consistency, rather
than responding immediately to reads, we submit the read to the consensus service, and respond with the latest value only when the consensus service returns. Thus, if two different replicas process simultaneous updates, and then each receive a read request, both reads will see both updates. In contrast, with sequential consistency, each read would be guaranteed to see the update submitted to its replica, but not necessarily the other update.

For weak consistency, a replica that receives an update first inserts the update into a cache and then immediately returns a successful response. The replica then asynchronously submits the update to the consensus service. Subsequent reads go to the cache first to return the latest updates. Once the consensus operation completes, the replica service applies the update to its state snapshot and removes the cache entry, meaning that all subsequent reads will view the now-consistent version of the data.

An important invariant of our approach is that all replicas have the same persistent state snapshot, even under weak consistency. Further, we do not need an explicit conflict resolution mechanism as conflicts are resolved by the serialization order assigned by the consensus service, e.g., “last” write wins. Of course, application-specific mechanisms may be required to handle a cache read that returns a different result than what would have been returned had the system waited for consensus to complete.

4.5.1 Replicated Log

Our high-performance consensus service, with a replicated log interface, forms the core of the metadata and data store replicated state machine services. We chose the Paxos [67] consensus protocol as the basis for our implementation, described in more detail in Chapter 3. The metadata and data store RSM services each create a new replicated log, associated with a set of replicas, for every object. The consensus service differentiates logs with a unique log key, based on the object’s name. RSMs submit updates (or Paxos proposals) to be appended to the end of a log. The consensus service internally assigns every proposal a unique log index and runs the Paxos protocol to reach consensus. When the Paxos protocol completes
for a given proposal, we say that the proposal is \textit{chosen}, which means it has been serialized into the log and no replica can ever have another update chosen for that index. The consensus service issues a callback to the RSM when it learns the next consecutive chosen index for a log, at which point the RSM can \textit{apply} the operation at that index to its state snapshot. To allow reclamation of old applied entries from the log, the RSM informs the consensus service when a given index has been applied to persistent state. Because the replicated log is continually truncated, a new replacement replica must copy the state snapshot for the object from an existing replica.

\subsection{4.5.2 Recovery}

As with any data center scale system, we expect hardware and machine failures to be frequent events. We use an active failure detection service to monitor application liveness from all servers within a data center. When it has not received a heartbeat message within a configurable timeout, the failure detector notifies the allocation service. The failure detection service also monitors the liveness of the switching services at remote data centers, so that local placement service can avoid directing clients to temporarily unavailable sites.

When notified of a machine failure, the allocation service begins recovery by compiling a list of the objects stored on the failed node. It then instructs the switching service to remove all references to the recovering addresses from its forwarding cache. Next, for each object, the allocation service determines that server’s role—metadata or data store server—picks a suitable replacement server within the same data center, and sends a message to one of the non-failed metadata replicas to begin recovery. The allocation service periodically retransmits the recovery message until it receives a confirmation for the completion of the reconfiguration. We omit a detailed discussion of individual server recovery due to space, but provide a brief overview.

The appropriate RSM directs the consensus service to remove the failed replica from and add its replacement to the log membership. At this point, the new replica may participate in consensus, but may not yet respond to client requests.
The new replica then requests a state snapshot for the object at an index greater than or equal to \( i \), the index at which the primary was added to the replica group. The snapshot that the new replica receives will be at some index \( k \geq i \). To complete recovery, the replica initializes its state snapshot to \( k \) and then replays any updates in its the replicated log from index \( k \) to the current index.

If the server(s) hosting the allocation service fail and are replaced, the allocation service’s state can be recovered by querying each machine in the data center for the objects hosted at that machine. The locator service contains only static state and recovers by copying this state from another instance. The switching and placement services maintain only soft state, so they require no recovery.

### 4.5.3 Automatic Replica Set Management

Our goal is for ZANTE to support millions or billions of state machines. Since each state machine has its own consistent replicated log, we must be able to create new logs dynamically. Creating a new Paxos log requires establishing the set of replicas (or replica set) to participate in consensus for that log. We call this process replica set establishment. Changing the members of a replica set for a given log involves executing a replica set change. Note that a replica set change implicitly involves a view change. Replica set changes and establishment are performed as proposals in the operation log, and every replica set has an associated replica set index at which it was chosen.

To the best of our knowledge, all existing systems that use consensus require replica set establishment and changes to be initiated manually in response to a failure, especially when operating under our permanent failure model. Some systems [70] perform automated replica set maintenance, but not in reaction to failures. For these systems, the number of RSMs is small and permanent failure infrequent. To support our scalability requirements, we had to design and implement automatic replica set establishment for fast state machine instantiation and automatic replica set maintenance for fast recovery from failures.
4.6 Performance

ZANTE must optimize both the latency and throughput of client operations for all types consistency. To minimize latency, our consensus service uses a leader for coordinating Paxos, which reduces the basic three rounds of messages for every proposal to just two, of which only one is on the critical path [66]. For three replicas, this decreases critical-path latency to a single wide-area round trip to one other replica. Using a standard “fixed-leader” approach, as opposed to a rotating-leader [73], non-leader replicas that receive requests must forward them to the leader to be serialized and wait for a response from the leader before responding to the client, adding two one-way message delays to the latency. Thus, a replica’s proximity to the leader will impact both the latency and throughput its clients see for updates.

We have taken this into consideration in the design of our placement service, whose function is to pick the “best” replicas for a given object or client. The metadata service first consults the placement to obtain the set of data store replicas for a new object. Every time a client opens an object, the metadata service asks the placement service to rank the data replicas for that client, e.g., by network distance to individual replicas and overall data center load. In our implementation, we always attempt to assign the data center closest to the client as one of the data replicas and to designate that replica as the Paxos leader. When picking the replica on an Open, our policy always favors the local data center, even if it is not the leader, which minimizes read latency but may add extra delay for updates. We leave more complex assignment policies, including separate rankings for reads and updates and dynamically reconfiguring data store replica group membership, as future work.

To achieve high throughput, ZANTE supports pipelined requests. By submitting many outstanding requests, clients can attain high throughput despite the latency inherent in wide-area round-trip times for strong consistency. Further, the consensus service sends the payload only once to conserve bandwidth and reduce message delays; subsequent messages employ strong hashes [35].

When offered load surpasses system capacity, peak throughput will decrease
as the system begins to suffer from livelock. To provide good throughput under high load, ZANTE implements an admission and rate control system modeled after the TCP retransmission timeout computation [29, 88]. The rate control system measures the latency for operations as they complete, and uses a statistical significance test [50] to compare these samples against an exponentially-weighted moving average of request latency to determine how to adjust the maximum number of outstanding requests, described in more detail below. As shown in §5.4.3, the rate control system maintains low operation latency and high throughput by throttling client requests past peak utilization.

### 4.6.1 Admission Control

As a client request-driven system, a given data store server may receive any number of simultaneous client requests: a single client may send many pipelined requests, or many clients may send requests for the same or different maps. For a given map hosted on a given set of servers, some bottleneck resource exists that limits the steady-state throughput at which the system can process requests. As our experiments show, this resource will typically be either CPU speed for small requests, or network bandwidth for large requests. Regardless of the particular bottleneck resource, if ZANTE continues to accept and process requests after it has already begun processing its steady-state maximum, the latency for all requests begins to increase dramatically as the requests begin to languish in queues. If the offered load increases too much, the peak throughput will actually begin to decrease slightly as the system begins to suffer from livelock. Therefore, to provide good throughput under high load, we conclude that ZANTE must have an admission and rate control system.

Our goals for designing the rate control system were fourfold: 1) maintain a request latency near the minimum possible, 2) achieve a throughput close to the maximum possible, 3) allow full utilization of available resources, and 4) dynamically determine the optimal rate. Regarding the last point, we wanted a rate control system was not hard-coded or sensitive to any user-provided parameters specific to a given deployment scenario; specifically, we wanted the same rate control sys-
tem with the same parameters to work equally well when used on a LAN, in a wide-area deployment with uniform latencies, and in a wide-area deployment with non-uniform latencies. Designing an admission control system to meet all these goals proved to be rather challenging, as the goals of high throughput/utilization are at odds with that of low latency.

We modeled our system after the TCP retransmission timeout computation, as in [29, 88]. ZANTE maintains a list of outstanding requests, i.e., requests that have been submitted to the consensus service and are currently in progress. The rate control system dynamically determines and adjusts the optimal maximum number of outstanding requests, and the admission control system stops accepting new requests when this maximum has been reached. The optimal number of outstanding requests is computed based on the exponential weighted moving average of request latency and variance of request latency. We record the target latency, computed by the rate control algorithm, that a request should meet when we submit a new request to the consensus service when that request fills the last available request slot. When that request completes, if it took less time than the target latency, then we increase the number of slots by one. If it exceeded the target, then we decrease the number of slots by two. Furthermore, to prevent the number of slots from slowly growing at an unbounded rate, every time we process a number of requests equal to the maximum number of slots, and the number of slots has not been filled, then we decrease the number of slots by two. We use parameters $\alpha = 1/8$, $\beta = 1/4$, and $K = 1$ to control the decay rates of the EWMAs and the multiple of the variance. Our experiments have shown that our algorithm is not very sensitive to $\alpha$ or $\beta$, but it is fairly sensitive to $K$—smaller $K$ values bias the control towards lower throughput and lower latency, and larger $K$ values bias towards higher throughput and higher latency. When all available slots are full, the admission control system waits for active requests to leave the system and thereby create space for new requests to be admitted. Any new requests received by a server during this time are queued in the TCP transport, which increases the client-perceived latency of those requests. Clients may also implement their own rate control algorithm based on the latency they observe for requests for the same
map—as the latency increases, they can decrease their own rate. If a client always sends as fast as possible, eventually its TCP queue will fill and it will block waiting for available TCP socket buffer space.

4.6.2 Versions

When clients may be redirected to different replicas to sustain availability in the face of failures, they often want a guarantee that a read will not return a stale value. Linear consistency ensures this, but comes at the significant expense of additional wide-area latency and load on the system for all read operations. To avoid this cost, ZANTE employs an alternative (if slightly weaker) approach: version numbers. Every data operation response includes the current version at the replica. Clients may optionally include this version in subsequent requests, thereby informing the service of the latest version they have seen. Reads with greater versions will be deferred until the local state has reached a version consistent with the request. Write operations need not be deferred, as the consensus service will correctly serialize them regardless of the version of the state snapshot when they are issued. Our client library will, if desired, automatically track version numbers to ensure consistent reads with no additional complexity for the client application.

Versions provide another performance benefit as well, a feature we call waiting reads. A waiting read is essentially a callback that the client may register with ZANTE to be notified when an object is updated. The client may set a waiting flag and the last version it has seen for any read request, and ZANTE will only respond to the read when its version exceeds the one provided by the client. The data store service will clear waiting read requests after a timeout to avoid having to track state indefinitely. Waiting reads improve performance by both decreasing the response time for the client when a new value is available and also by decreasing the load on the system that would be caused by clients otherwise having to poll. We use this feature in our queuing service application discussed in § 5.5.2.
4.7 ALLOC and OPEN example

To demonstrate the end-to-end process of client interaction with ZANTE, we walk through an example scenario where a client allocates and opens an object named \(x\), as illustrated in Figure 4.3. The client uses the locator service to hash the object name to the data center, \(A\) in this example, statically responsible for storing the metadata for \(x\). In step 1, the client transmits an ALLOC to the data center, which contacts the local allocation service for the metadata server. In step 2, the metadata service computes its two successor replicas in the ring, \(B\) and \(C\) in this example, and runs consensus to establish a replicated metadata log for \(x\). In step 3, the metadata service acknowledges success of the client ALLOC. Simultaneously in step 4, the metadata server contacts the placement service for the data store replica sites to store \(x\), obtaining, in this case, \(B\), \(D\), and \(F\). The metadata service forwards the request to one of the data store sites, \(F\), which initiates consensus in step 5 to establish a replicated data log for \(x\). In step 6, \(F\) acknowledges success to the metadata server. In step 7, the client attempts to OPEN \(x\); the metadata server retrieves data replicas \((B, D, F)\) from its local state and sends the set to the placement service for the closest data center, \(F\) in this case. In step 8, the
metadata server responds to the client with site $F$. Finally, the client sends a Write request directly to $F$ in step 9.

## 4.8 Architecture

To support very large numbers of replicated state machines, ZANTE efficiently multiplexes RSMs on physical resources. Figure 4.4 shows the ZANTE layered service architecture. Every ZANTE server running the metadata or data store services can support many replicated objects, each sharing the same RSM logic. Figure 4.4 shows three different objects, $X, Y, Z$, and their associated state. The replicated state machines use the consensus API provided by Paxos to maintain their state based on consistent replicated logs, with each object having its own replicated log and Paxos state. To manage this state, we designed both ZANTE and Paxos to store replica data in a few Berkeley DBs. By combining the object ID as part of the DB key, we can efficiently combine replica state in the same
underlying database.

4.9 Summary

In this chapter, we described the challenges for deploying scalable replicated state machines across the wide area, including locating and naming replicas, automatic replica group creation and maintenance, and efficiently managing replica state. We described techniques to overcome these challenges, including decoupled consistent metadata and dynamic replica binding. In Chapter 5, we will look at a storage service we built using ZANTE and a number of sample applications, as well as an evaluation of the systems. In Chapter 7, we will describe the scalable software message routing service used to implement dynamic replica binding.

4.10 Acknowledgment

Chapter 4, in part, is currently being prepared for submission for publication of the material. Anderson, James; Rasmussen, Alexander; Vahdat, Amin. The dissertation author was the primary investigator and author of this material.
Chapter 5

MapStore

In this chapter, we present MapStore, a scalable storage service providing strong consistency designed for fine-grained wide-area cross-data center replication. MapStore exports named sorted maps as a powerful and convenient infrastructure for distributed applications and services. Applications, which may be running at an arbitrary location (from an end-user smartphone to one of the data centers themselves), perform reads and writes to the named maps. The underlying storage is replicated at multiple data centers using Zante for read locality, high availability, and disaster recovery. We evaluate MapStore with high level applications modeling a queuing service [23] and a blogging application built on BigTable [38].

5.1 API

MapStore provides a sorted map as the storage abstraction, which offers an expressive interface for client applications and allows for an efficient implementation. Specifically, MapStore provides the abstraction of a sorted collection of \( \langle \text{key}, \text{value} \rangle \) pairs. Keys are unique and both keys and values are arbitrary byte strings with a fixed maximum length. Maps are sorted lexicographically by key to allow for in-order iteration through keys and lower-bound prefix matching, both of which are used in the applications described in Section 5.4.

Clients create a new MapStore map object by calling Alloc and specifying the desired consistency. Clients must Open a map object to receive a MapDe-
Table 5.1: MapStore API

<table>
<thead>
<tr>
<th>Metadata Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ALLOC</strong> *(objectname, seq</td>
</tr>
<tr>
<td><strong>OPEN</strong> *(objectname, read</td>
</tr>
<tr>
<td><strong>FREE</strong> <em>(objectname)</em></td>
</tr>
<tr>
<td><strong>READMETADATA</strong> <em>(objectname)</em></td>
</tr>
<tr>
<td><strong>SETPERMISSIONS</strong> <em>(objectname, readers, writers, owners)</em></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>READ</strong> <em>(objectaddr, key)</em></td>
</tr>
<tr>
<td><strong>READRANGE</strong> <em>(objectaddr, lowerbound, upperbound, limit)</em></td>
</tr>
<tr>
<td><strong>READLIST</strong> <em>(objectaddr, key1...key_n)</em></td>
</tr>
<tr>
<td><strong>CONTAINSKEY</strong> <em>(objectaddr, key)</em></td>
</tr>
<tr>
<td><strong>FIND</strong> <em>(objectaddr, lowerbound, inclusive)</em></td>
</tr>
<tr>
<td><strong>WRITE</strong> <em>(objectaddr, key, value)</em></td>
</tr>
<tr>
<td><strong>REMOVE</strong> <em>(objectaddr, key)</em></td>
</tr>
<tr>
<td><strong>APPEND</strong> <em>(objectaddr, value)</em></td>
</tr>
<tr>
<td><strong>COMPAREANDSET</strong> <em>(objectaddr, key, compare, value)</em></td>
</tr>
<tr>
<td><strong>MULTIOP</strong> <em>(objectaddr, comparisons, writes, removes)</em></td>
</tr>
</tbody>
</table>

scriptor before reading or writing it. Data operations include several variants for reading the values for one or more keys, searching for the next key greater than a reference key, writing a key/value pair, and removing a key and its value. In addition, MapStore supports an atomic operation (**MULTIOP**) that compares and then writes/removes multiple keys [36]. Finally, MapStore provides an operation to append a value with a key that the service generates to be the lexicographically largest in the object.

5.1.1 Data Model

We considered three different common storage abstractions for MapStore: an infinite byte array [21], a sorted map, and an object store with containers [23].
Our chief concern when choosing a storage abstraction is choosing the right trade-off between ease of use, flexibility and power. The byte array abstraction is the simplest and most flexible abstraction of the three, but constructing more powerful abstractions atop this abstraction requires a great deal of programmer effort and such a low-level abstraction may require many operations to perform one logical operation. In contrast, the object store abstraction has a rich interface that makes it easy to build some classes of applications and can usually perform its operations using a single message. However, other classes of applications are made more difficult to build because the programmer is restricted to the high-level interface that the object store provides.

Ultimately, we thought that the sorted map abstraction provided the best set of trade-offs. A map data structure consists of \( \langle \text{key}, \text{value} \rangle \) pairs. Our abstraction allows both keys and values to be arbitrary byte strings with a fixed maximum length, and requires that keys within the same map are unique. Maps are lexicographically sorted by key to allow for in-order iteration through keys and lower-bound prefix match lookups, both of which are used extensively in the sample applications described in Section 5.4.

To better illustrate many of our design decisions for our abstractions, API, and overall system architecture, we will frame our rationale in relation to three example applications/services: a file system (MFS), a work-queueing service (MQS) modeled after Amazon’s Simple Queue Service [23], and a structured-data storage service (MTABLE) modeled after Google’s BigTable [38]. We feel that these applications are representative of the classes of service that a general-purpose wide-area storage service should support. Furthermore, when replicated at multiple sites, all of these applications would benefit from strong consistency by 1) reducing complexity: no code would be needed to handle conflicts caused by concurrent weakly consistent writes; and 2) correctness: as shown in our evaluation, some MQS tasks may never be processed under weak consistency.
5.2 Abstractions

The storage abstraction exported by our service should be sufficiently low-level and general to naturally support a variety of applications and services, while retaining as much power as possible to support high-performance services. We consider the power of an abstraction to be the amount of developer effort required to make that abstraction accomplish a given task: a weaker abstraction requires more operations than a more powerful one. Meeting our goals requires balancing fundamental design tensions between the generality and power of the abstraction, both of which directly influence the simplicity of the interface versus the simplicity of applications, as well as considering how the abstraction may impact application performance.

Abstractions become more powerful by either narrowing their focus to become more specialized or by increasing their complexity (or both); thus either generality or simplicity is usually sacrificed to obtain the more powerful abstraction. We can illustrate this idea with three familiar of programming languages: C, C++, and Perl. The C language is very general and can be used to build almost any type of system and runs on almost any platform. But it is also the “weakest” of the three languages, typically requiring more lines of code to perform similar tasks. C++ introduces more powerful abstractions in the form of classes and templates, but this power comes with additional complexity. By our definition of power, Perl is also more powerful than C (and C++), in that very short Perl programs can accomplish tasks that would take pages of C code. However, Perl attains this power through sacrificing the generality of C; e.g., Perl is much better at string processing than handling binary data.

We considered three common abstractions for our storage service: an infinite byte array [21], a sorted map, and an object store with containers. The byte array is the lowest level abstraction, has a very simple interface, and is the most flexible—you could build any other abstraction and application on top of it—but it is also the least powerful, in that it requires the most effort to build more powerful abstractions. In contrast, an object store provides a very high-level abstraction with a fairly complex interface: create/delete objects and containers, assign objects
to containers, and read/write byte ranges. This powerful interface makes it easy
to build some types of applications; for instance, building a file system would be
relatively simple with this type of object store—objects can represent files and
containers can represent directories.

The disadvantage of choosing such a high-level abstraction is the resulting
limitation on the set of supported applications. For instance, consider modeling
a work-queue service with an object store. First, queues have a definite order.
A general object store is a collection of objects and containers, without any se-
manics defining an order. So some mechanism will be needed for imposing the
order required by the queue on the object store: this may be through storing
a special object with the ordering of the queue items; or perhaps an order can
be derived through assigning special significance to the object metadata, such as
object names or modification times, if the object store supports listing contents
sorted by various metadata fields. The second challenge is handling the status of
work-queue objects, particularly those that have been read but not deleted. These
need to be temporarily removed from the queue and reinserted at the front after
a timeout. An object store might model this by having a second object store a
list of these pending items, which would need to be consulted and updated on
every read or deletion. An alternative design might use a separate container for
the pending items, moving them from the normal queue container; some care will
still be needed to handle the metadata associated with the timeout and original
position in the queue correctly.

Even with the file system, although an object store is a much more natu-
ral fit, depending on how the object store handles metadata for objects, storing
file metadata such as permissions and modification time may be inconvenient or
inefficient. In particular, if the object store does not allow arbitrary metadata to
be tagged to objects, then storing the necessary metadata for a file system would
require using separate special objects just to store the metadata, which now de-
mands additional complexity on the file system application as well as additional
operations to keep the metadata consistent with the contents of the files and di-
rectories.
The flexibility of the byte array also has some potential performance costs, as having an interface limited to reading or writing byte ranges typically means that more reads or writes must be issued to perform more complicated actions, which a higher-level abstraction could perhaps perform with a single operation. This performance penalty is not just in the additional latency of having to perform more operations, although that in-and-of itself can be considerable. In the cloud computing context, each interface call involves an RTT across the Internet. The more powerful abstraction may actually be implemented in a more efficient fashion, meaning it actually does not need to perform the same computations required by a client using the weaker abstraction. Consider a file system operation to create a new file implemented using the byte array versus the object store. Using the byte array requires reading the bytes representing the directory contents, allocating a new memory segment for the file contents, adding the address for that segment to the directory, and then writing back the directory entry. For the object store, you simply create a new object and place it in the appropriate directory container (and perhaps set metadata as well, possibly in other objects, depending on the interface). The object store may be backed by a file system or even relational database, both of which provide powerful, high-level abstractions and interfaces, so executing the client request to create a new object is implemented simply and efficiently on the service side, as well as on the client side. With the byte array, the client must compute the new directory structure and re-serialize it to raw bytes before sending it back to the storage service, which requires considerably more complexity.

Next, consider the example of reading from the work-queue, which temporarily removes the item from the queue for a specified timeout unless it is explicitly deleted. Using a byte array, the queue might be represented as a linked list, in which reading from the queue requires reading the head of the list, updating the head pointer to point to the next item, and adding the read item to a pending timeout list so it could be reinserted later. An implementation that supports mini-transactions could potentially complete this in two operations. With an object store abstraction, objects could represent queue items two containers could rep-
resent the queue and the pending timeout list. First, the queue container would have to be read and sorted, then the first object would have to be read, and then it would need to be moved from the queue container to the pending container. This would take at least three operations, one of which—the reading of the container contents—could potentially be expensive.

We felt that a sorted map interface provided the best set of trade-offs. To be explicit, a map data structure is one consisting of <key, value> pairs. Our abstraction allows an arbitrary byte string up to a fixed maximum length for each key and value and only allows unique keys. This abstraction can be provided by multiple map implementations, including ones based off of hash-maps, red-black trees, or B+-trees. We decided to further refine our abstraction to provide sorted maps. These have the property that they 1) allow in-order iteration through the keys in lexicographical order and 2) allow for a lower-bound prefix match lookup that can return the next key that is lexicographically greater than a search string. We describe how we make use of both of these properties with our example applications below. Finally, sorted maps because offer an efficient base implementation on the server side using in-memory sorted maps or B+-trees for disk based storage, with \( O(\log n) \) insertion, lookup, and delete, and linear in-order iteration.

Thus, a sorted map is a powerful abstraction. Applications and services have a common and useful data structure as their baseline abstraction. Sorted maps further provide a flexible abstraction: they provide straightforward and efficient implementations for both an infinite byte array, the less-powerful abstraction, as well as other common data structures, including arrays, stacks and queues. The most simple representation for an infinite byte array uses fixed size “blocks” as values, with the keys denoting the block offset from the beginning of the memory segment. For example, using 1k blocks, a map denoting a byte array of length 4k would have four key-value pairs, with keys 0, 1024, 2048, 3072. The client can statically compute the correct key(s) and the offset(s) within the corresponding block(s) based on the range of bytes requested. Similarly, the other data structures can be represented with a sorted map using simple conventions. The opposite, however, is not true—it is difficult, inefficient, or non-trivial to create a map representation.
using an array, stack, queue, or even memory segment.

**Array** Using integer keys, the key represents the array index and the value is the array value at that index. This representation provides in-order iteration through the array and single-read access to a random index.

**Stack** The keys are binary strings representing monotonic values, and the values are the values on the stack. To push a new value onto the stack, write a key-value pair with a new monotonically greater key. To read from the stack, read the largest key. To pop from the stack, delete the largest key.

**Queue** The queue representation uses the same convention as the stack, except reads and dequeue operations go to the smallest key.

Key-value pairs provide a natural idiom for indirection, a powerful abstraction for building more complicated data structures or representations of data relations.

To further illustrate the usefulness of the map abstraction, we sketch the design of our three very different example applications and show how maps simplify their designs. For practical implementation reasons, **MapStore** maps limit the size of keys and values to a maximum fixed size, which is reflected in the designs below.

**MQS** As described above, a single sorted map can implement a basic queue with the convention of using monotonic keys paired with the inserted values. To read/dequeue from the queue, the smallest key is read/removed. To implement the full semantics of a work-queue that reinserts read values after a timeout, additional support from the client is needed to mark entries as in-progress and to skip to the first unread value. Fortunately, the sorted map allows the client to scan the queue in-order to accomplish this.

**MFS** File systems are a natural fit for the map abstraction and represent an important class of storage applications for cloud computing. A single map can implement a file system using the classical Unix structure. The map contains three types of entries: a file system description block, an inode, or a data block. The file system description block uses a reserved, well-known key and its value is the inode for the root of the file system. Inodes use a unique inode number
for the key, and the value is the contents of the inode, including metadata about
the file. Data blocks are stored under a key which consists of the inode number
concatenated with a block number. Data block values contain either the contents
of a file or part of a directory listing.

**MTABLE** The BigTable abstraction is a map, one that maps \langle row, column,
timestamp \rangle tuples to values. Thus our simple MTable implementation uses a
map to do just this.

Starting with an initial list of target applications, we examined the range
of possible interfaces to our storage service. Any storage service must provide
one or more abstractions for the structure of stored data in the baseline interface.
Choosing this abstraction requires balancing various trade-offs between simplicity
of the interface, power of the interface, simplicity of applications, flexibility, and
performance. A very low-level interface provides the most flexibility, but at the cost
of requiring more effort to build applications. Further, it may not support desired
functionality with optimal performance. A high-level interface will make writing
certain classes of applications very easy and may allow for improved performance
for those applications, but sacrifices flexibility and precludes other optimizations
or abstractions. Our goal was to build a general purpose storage service that can
support a wide range of applications. Thus, we sought the best compromise for
these tensions of flexibility, power, simplicity, and performance as the low-level
building block for other applications.

Perhaps the simplest such abstraction is an infinite byte array [21]. An ex-
tension of this abstraction provides an object store, with each object being a named
byte array, possibly with additional per-object metadata, such as [23]. Adding
more structure to an object store provides a filesystem-like abstraction, with hier-
archies of containers (directories) that can hold objects (files), other containers, or
even pointers (symbolic links) to objects or containers.

We decided against an object store or file system for several reasons. We
felt that these abstractions were already too high-level and were not suitable for
applications that wished to use a specialized structure for storing their data ef-
ciently. The interface to both of these abstractions becomes complicated with
many operations or suffers from possible performance limitations, such as with an object store that does not allow partial writes. Ultimately, we decided that our infrastructure storage service should be as flexible as possible while still providing a powerful interface, and we reasoned that it would be straightforward to build a library providing an object store or file system on top of a lower level byte array or map data structure. Using similar reasoning, we chose the map abstraction over the raw byte array because its interface is no more complicated, but it provides a more powerful abstraction, as it is easier to build the abstraction of a byte array with a map than to build a sorted map with a byte array.

Our abstraction allows an arbitrary number of key-value pairs for any map (limited only by the available storage on the machine storing that map), but each key and value is limited to a fixed maximum size. We chose a sorted map interface over an associate array or hash-map because the applications we were considering would benefit from in-order traversal over the keys. Additionally, having sorted keys allows us to support a lower-bound operation, which enables our service to natively support a broad class of simple database-like applications requiring prefix-matching searches.

5.3 Implementation

We have a complete prototype implementation of MapStore written in Mace [62] and C++. Our prototype matches the descriptions in the preceding sections except for the following outstanding limitations. Currently, none of the allocation, locator, placement, or failure detector services are replicated for fault tolerance. Our current implementation of the allocation service uses basic policies for making assignments, including random and round-robin. Mace uses an event-driven model, and it currently only supports a single thread of execution within service transitions, which limits our utilization of multi-core machines. However, the Mace TCP transports use their own threads, which allows us to exploit some parallelism.
5.3.1 Security

The transport also uses Transport Layer Security (TLS) to provide authenticated and encrypted communication for all wide-area links, including those to the client. The metadata and data store services also query the transport for the client’s certificate, which they use for authentication and access control.

The MapStore security model uses public key cryptography for secrecy, authentication, and access control. Every MapStore client has a certificate signed by the MapStore certificate authority. The transport service uses the certificates to establish TLS sessions for all TCP connections over the wide-area. Recall that the metadata service’s state for a map includes lists of public keys that are authorized for reading and writing (an empty list means that there are no access restrictions, and a list consisting of a single empty key means that no access is allowed). When a client opens a map, the metadata service checks whether the client is authorized for the requested type of access by checking to see if its certificate is contained in the appropriate access lists. If the client is authorized, the metadata server creates a capability, which we call a MapDescriptor, for the client. The capability includes the MapAddr, an expiration time, the client’s certificate, and whether the client has read and/or write access. The entire certificate is signed by the metadata server’s private key.

The separation of the metadata and data store services further helps scalability with respect to security. When a client subsequently sends requests to the data service, it includes the provided MapDescriptor with every request. Before processing the request, the data service must validate that the MapDescriptor grants the requested type of access, that the certificate in the MapDescriptor matches that of the client, that the MapDescriptor has not expired, and finally that the signature on the MapDescriptor is valid. These validations, however, are much faster than those performed by the metadata server on an Open request, specifically because no ACLs need to be checked.
Table 5.2: Summary of important MapStore tables

<table>
<thead>
<tr>
<th>Service</th>
<th>Name</th>
<th>Mapping Stored</th>
<th>Backing Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>metadata</td>
<td>metadata</td>
<td>address name string → metadata</td>
<td>BerkeleyDB async</td>
</tr>
<tr>
<td>data store</td>
<td>addrdb</td>
<td>object name → data store service metadata</td>
<td>BerkeleyDB async</td>
</tr>
<tr>
<td>data store</td>
<td>valuedb</td>
<td>[object name, key] → value</td>
<td>BerkeleyDB async</td>
</tr>
<tr>
<td>Paxos</td>
<td>paxosdb</td>
<td>[log key, index] → Paxos state (promises, accepts, learns) for outstanding proposals</td>
<td>In-Memory Map</td>
</tr>
<tr>
<td>Paxos</td>
<td>logdb</td>
<td>log key → log state (view, applied indices, last notified)</td>
<td>BerkeleyDB async</td>
</tr>
<tr>
<td>Paxos</td>
<td>chosendb</td>
<td>[log key, index] → chosen proposal</td>
<td>In-Memory Map</td>
</tr>
<tr>
<td>Paxos</td>
<td>pendingdb</td>
<td>[log key, index] → pending proposal</td>
<td>In-Memory Map</td>
</tr>
</tbody>
</table>

5.3.2 Storing Service State

We designed the consensus/replicated log, metadata, and data store services to transparently use either in-memory C++ STL maps or file-backed Berkeley DBs to store their respective service state. The important tables used in MapStore are summarized in Table 5.2. paxosdb stores Paxos state messages for outstanding proposals, and stores as many entries per objects as the number of outstanding proposals for that object. This table will be empty in a quiescent system. logdb stores the state of each object’s metadata and data store logs. chosendb stores the chosen proposals for each object. There is one entry in this table for each outstanding proposal for each object; in a quiescent system, this table will be empty. pendingdb stores pending proposals for each object. Theoretically, there are as many entries in this table per object as there are outstanding proposals for that object. In practice, however, this table is usually empty because out-of-order proposals will only be chosen when failures or network errors occur. This table will be empty in a quiescent system.

addrdb maps MapAddr $s$ to data store service metadata, including the other data center replicas, the local data center backups, the current index in the replicated log (used for sanity/consistency checking), the version of the map, and the current version of the metadata service. In steady state, each map name has a corresponding entry in this table, but a map name may have multiple entries when maps with the same name are allocated and freed quickly.

valuedb is a single table that stores all the key-value pairs for all the maps
stored on that server. This design was chosen over the alternative of having a separate data store service map for every MapStore map to simplify persistent storage. There is one entry in this table for each key of each map.

To maintain correctness across failures, all four of the Paxos tables must be synchronously written to disk. The metadb and addrdb tables must also be stored on disk. However, asynchronous updates are sufficient for these because a node recovering from a non-clean shutdown must still replay the Paxos log, as the crash might have occurred before the appropriate update was made to the MapStore service state.

As we show in §5.4.4, storing all of the tables on disk causes MapStore to bottleneck on disk writes and therefore has a significant impact on overall system throughput. Our evaluation shows that even when enabling a large BerkeleyDB cache, an in-memory write-ahead log, and asynchronous disk writes, using disk-backed storage for the tables still results in greatly decreased throughput relative to using in-memory maps.

For many services, being able to survive a machine crash is not necessary. If a machine fails (i.e. stops responding to the failure detector service for more than a timeout) for any reason, another server will be reassigned to store its objects. This approach also simplifies the logic of the failure detection service, since a single policy can control the response for any type of failure event.

While storing all service state in memory yields the fastest performance, it would be impractical for most deployment scenarios because it limits the aggregate capacity of all MapStore objects to the aggregate memory capacity of the servers. Thus, for our default storage configuration, we use a configuration that offers good performance while still allowing the aggregate capacity to be bounded by disk size, not memory. Specifically, we store the metadb, valuedb, and logdb tables in file-backed BerkeleyDBs with asynchronous disk writes and an in-memory write-ahead log because these are the only tables that actually grow as either the number of objects or the number of values per object increases. The other tables—paxosdb, chosendb, and pendingdb—are backed by in-memory maps, since Paxos frees state associated with a request once it has been processed. While this design limits
the total number of simultaneous requests that may be in progress to available memory, we show in §5.4 that, in practice, memory capacity will likely never be the bottleneck resource for MapStore.

5.4 Evaluation

To better characterize the costs of strong consistency, we present a detailed performance analysis of Paxos. We then examine microbenchmarks showing how MapStore performs for various latency, throughput, and scalability tests.

5.4.1 Experimental Setup

We use ModelNet [107] to simulate dedicated wide-area links between a set of data centers by passing all wide-area traffic through a ModelNet core running on a dedicated machine. For most of our experiments, we use a configuration with three data centers of ten machines (four switching servers and six metadata/data store servers) each connected in a ring by 15Mbps links with latencies of 20ms, 50ms, and 70ms. We use different topologies for the scalability experiments with data centers of five machines (two switching servers and three metadata/data store servers) each connected in a ring with 15Mbps links with uniform 20ms latencies. Each data center consists of either two machines running the switching, placement, and locator services and three machines running both the metadata and data store services, or four switching machines and six metadata and data store service machines. Up to 60 clients are multiplexed evenly on four machines. Clients communicate with their local data center over a 15Mbps, 3.5ms link.

We have configured ModelNet so that intra-data center traffic uses a physical non-blocking Gigabit Ethernet switch rather than being routed through the ModelNet core. Each machine in the cluster has a 2.13GHz quad-core Intel Xeon processor, 4GB of RAM, and a 1Gbps Ethernet adapter.
5.4.2 Network Bandwidth Overhead

We next measure the network requirements for strong consistency in terms of bytes transferred. We have a client submit 10,000 Write operations of varying size and measure the bytes sent and received by the leader and other replicas. Subtracting the client payload, the leader sends 551 bytes per request per replica, and each replica sends 110 bytes per request. Thus, the total per replica overhead of Paxos is 651 bytes per request, which is independent of the client request payload size.

5.4.3 Microbenchmarks

To examine the performance of Write operations under both strong and weak consistency, we have a single client continuously issue Write operations to a single object for a period of three minutes and measure the average throughput in operations per second and the latencies of operations once the rate control system (cf. §4.6) has stabilized. We vary the maximum number of pipelined write operations the client sends and the size of the values being written.

Figure 5.1a shows the throughput for strongly- and weakly-consistent write operations. Each Paxos proposal incurs a fixed CPU overhead that is not proportional to the size of the value; thus, for small (1B) writes the client can pipeline enough requests to saturate the servers’ CPU. For larger (1KB and 10KB) writes, the throughput is constrained by the bandwidth of the link between the data centers. Other systems [41, 59, 73] show how batching can be used to (arbitrarily) increase peak throughput; while MapStore can be used to batch requests, we focus on the baseline performance of individual requests.

Figure 5.1b shows the latencies measured by both the client and the leader, which measures the time for consensus to complete. We see a consistent consensus latency for requests. Once the client saturates the server’s CPU and the rate control begins to throttle the client, the client latency becomes progressively worse. While the client does self-clock itself to a maximum number of pipelined requests, it does not rate limit itself based on the operation latency.

Figures 5.1 also shows the throughput and latency of weakly-consistent 1B
Figure 5.1: Throughput and latency for strongly-consistent writes as write pipeline width and write size increase
Table 5.3: Comparison of throughput for different table storage options

<table>
<thead>
<tr>
<th>Storage Configuration</th>
<th>Avg. Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Std. Configuration</td>
<td>1929 ops/sec</td>
</tr>
<tr>
<td>In-Memory Maps</td>
<td>2555 ops/sec</td>
</tr>
<tr>
<td>Sync. Berkeley DBs</td>
<td>70 ops/sec</td>
</tr>
</tbody>
</table>

Write requests. Since the local data center responds to requests immediately, the client’s observed latency is much lower. The throughput saturates for the same reason as it did for strongly-consistent writes, but does so for a smaller number of pipelined requests because the lower latency allows the client to issue requests at a much faster rate. The consensus latency observed by the server remains unchanged.

We also ran microbenchmarks to measure the baseline read performance, which can be serviced from a single replica without consensus (cf. §4.1.1). We began by having a single client issue reads with no pipelining for a single byte value. The client can read at a rate of 125 operations/sec and sees an average latency of 7ms (the RTT to its data center). To measure the peak read throughput, we had three clients issue 100 pipelined reads each to the same object. The server delivered 16493 reads/sec, and the clients saw an average latency of 15ms.

Finally, we measured the latency for an Alloc operation by averaging the results of a client issuing sequential Alloc requests for different object names. The client observed a latency of 207ms, and the servers observed a consensus latency of 200ms. This latency reflects that Paxos must run the three round version described in [67] without a designated leader to establish a new replicated log. Thus, it incurs two RTTs from both replicas, thus four times 50ms for the observed 200ms delay.

5.4.4 Comparing Storage Configurations

Next, we examine the costs of using the various table storage options discussed in Section 5.3.2. To evaluate the storage system, we ran the Write operation microbenchmark discussed in the previous section using our standard configuration, storing all tables in in-memory maps, and storing all tables in BerkeleyDB
databases with synchronous writes enabled. Table 5.3 shows that the additional latency imposed by synchronous writes dramatically decreases the client’s average throughput, and that storing all tables in memory provides a substantial increase in performance.

To further explore the costs of using our standard configuration, we ran the Write operation microbenchmark with a maximum of 140 pipelined requests and 1B writes and measured the latency of put and get operations to BerkeleyDB with asynchronous disk writes and an in-memory log. Figure 5.2 shows cumulative distributions of request latencies. We found BerkeleyDB reads to be fast and consistent: 90% of table reads finished in under 50 microseconds. However, we found the latency of put operations to BerkeleyDB to be quite heavily-tailed: 50% of requests finished in under a millisecond, but the slowest 1% of writes experienced latencies in excess of 8 milliseconds.
**Figure 5.3**: Aggregate throughput of clients and servers as a function of number of clients (1-byte writes, 120 pipelined operations, standard storage)

### 5.4.5 Scalability

We measure scalability with two different benchmarks. First, to show the effect of different objects being placed on different servers using a fixed data center topology, we vary the number of clients with each client issuing 1B WRITE requests to a separate object with a maximum of 120 pipelined requests. Figure 5.3 shows that as we increase the number of clients, the aggregate throughput of all clients increases nearly linearly, as each object has a new server to handle the requests.

Second, we evaluate MapStore’s ability to scale as the number of data centers increases. Each data center has three servers to handle requests, so adding more sites increases the total capacity. This experiment measures aggregate throughput in operations per second as clients partitioned evenly among data centers continuously write data to separate objects. Clients pipeline 40 writes at a time to ensure saturation. Beginning with the topology described in Section 5.4.1, we incrementally add data centers to the data center ring. Table 5.4 shows that our scalability is almost perfectly linear, with the slight decrease in aggregate throughput between four and five data centers due to contention for wide-area links.

We next perform a similar experiment, except we measure scalability for reads rather than writes. 30 clients issue READ requests with a maximum of 60 pipelined requests. Each data center in this example consists of a single physical
Table 5.4: Scalability vs. number of data centers (as measured on the servers in operations per second)

<table>
<thead>
<tr>
<th># Data Centers</th>
<th># Servers</th>
<th>Write Throughput</th>
<th>Read Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>W</td>
<td>R</td>
<td>Total</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>3</td>
<td>8069</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>4</td>
<td>10412</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td>5</td>
<td>12905</td>
</tr>
</tbody>
</table>

server running the data store service. Our results in Table 5.4 show a slightly super-linear scalability as the number of data centers increases. This is due to the fact that we issue a large number of simultaneous requests to a small number of replicas and do not perform rate control on read operations.

5.4.6 Fault Tolerance

To measure the effects of failures on MapStore, we had a client continually issue 1 byte writes to a single object with a maximum of 20 pipelined writes and killed various services to see how the system would react.

In the first experiment (shown in Figure 5.4a), we kill the data store and metadata services for the client’s object at its local data center after approximately 15 seconds. We specifically disabled client fail-over to another replica (see below) for this test to show the effect on throughput after recovery. The recovery of the data store service requires a wide-area data transfer, which takes about five seconds, after which the client’s throughput is restored.

In the second experiment (shown in Figure 5.4b), we kill all instances of the switching service at the client’s data center after approximately 15 seconds, effectively rendering the data center unavailable. After approximately 3 seconds, the client times-out all outstanding requests and re-opens the object, which returns a another data center site. The client can begin immediately sending new requests, but the additional latency to the other data center and the temporary unavailability of the leader replica decreases the client’s throughput significantly. The throughput
Figure 5.4: Client throughput over time during failure events
Table 5.5: Breakdown of reasons for incorrect comment reads in the blogging application for serial and weak consistency

<table>
<thead>
<tr>
<th>Read Result</th>
<th>Serial</th>
<th>Weak</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>100.0%</td>
<td>91.9%</td>
</tr>
<tr>
<td>Wrong UID</td>
<td>0%</td>
<td>5.3%</td>
</tr>
<tr>
<td>Wrong Client</td>
<td>0%</td>
<td>2.8%</td>
</tr>
</tbody>
</table>

will be restored when either the leader’s site regains connectivity, or if one of the other replicas elects a new leader after a timeout (not currently implemented in our system). If the client were issuing read requests rather than writes, it throughput would not see this decrease.

5.5 Application Case Studies

To quantify the trade-offs between consistency models for real applications, we implemented and evaluated a BigTable service with a blogging application and a queueing service robust to client failures.

5.5.1 BigTable Blogging Application

BigTable [38] is a distributed storage system developed by Google for managing structured data. Data is collected into tables, each of which is a sparse, multidimensional sorted map. Cells in a table are referenced by a (row, column, timestamp) triple. Users interact with BigTable by creating and deleting tables, reading a single row in a table, iterating over multiple rows and performing atomic read-modify-write transactions on a single row. Rows and columns are created dynamically, but each column must belong to a column family.

We implemented a service using MapStore that supports a subset of the BigTable API. Each table is stored in MapStore as a single object. Since both BigTable tables and MapStore objects are sorted maps, the translation between BigTable and MapStore is quite straightforward. Keys in a table’s object are of
the form \((\text{row\_name, column\_name, timestamp})\) and are lexicographically sorted as such. Transactions over a row in a table are performed using a single \texttt{MULTIOP} operation.

To read a row in a table, the BigTable service interprets the name of the row \(R\) as an integer and issues a \texttt{READRANGE} operation with \(R\) as the inclusive lower bound and \(R + 1\) as the exclusive upper bound. For simplicity, we do not explicitly support column families, and assume that all columns in a table belong to the same column family.

To evaluate our implementation, we wrote a simple blogging application that uses BigTable. All blogs are stored in a single table, and each blog post is a row in this table. Comments for a post are stored in columns named \texttt{comment:UID}, where \(UID\) is a monotonically increasing sequence number whose maximum value is stored in a column named \texttt{max\_id}. We want to ensure that all readers of a blog see all posts and comments in the same order. To write a comment for a post, the client reads the maximum comment ID \(C\) from the post’s \texttt{max\_id} column. It then issues a transactional updates to the post’s row until it successfully writes its comment with \(UID\ C + 1\) and sets \texttt{max\_id} to \(C + 1\).

If the transaction fails (because another commenter has posted a comment and incremented \texttt{max\_id} between the commenter’s read and its write), the BigTable service returns an error along with the contents of the \texttt{max\_id} column at the time the transaction was attempted. The commenter uses this updated value as the new value of \(C\) and retries the transaction.

We first populate the blog table with 600 blog posts distributed evenly among 60 blogs. 60 clients distributed evenly among three data centers interact with these posts by either reading the comments for a post or writing a comment to a post. Clients choose to read or write a post at random with an 80%/20% read/write ratio. For each operation, clients select a blog uniformly at random from the set of blogs, and select the post within that blog based on a Zipfian distribution with \(s = 1.1\) that prefers more recent posts to older posts.

We examine the costs of serial consistency in terms of the throughput and latency of client blog operations. Weak consistency provides slightly better through-
Table 5.6: Number of times each enqueued value was read

<table>
<thead>
<tr>
<th># Times Read</th>
<th>Strong</th>
<th>Weak</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0%</td>
<td>6.2%</td>
</tr>
<tr>
<td>1</td>
<td>100%</td>
<td>87.0%</td>
</tr>
<tr>
<td>2</td>
<td>0%</td>
<td>4.2%</td>
</tr>
<tr>
<td>3</td>
<td>0%</td>
<td>2.6%</td>
</tr>
</tbody>
</table>

put for writes, with clients completing three writes per second for weak consistency versus two for serial (the workload is read-dominated, so clients are not attempting to issue writes at a high rate). Weak consistency, as expected, yields a much lower latency, with clients averaging 130ms versus 220ms for serial consistency.

Table 5.5 summarizes the effects of weak consistency on correctness. After the experiment completes, we read a definitive list of comments for each blog post. We then compare the result of all reads from the experiment to the definitive set. A read is considered correct if the comments a client read are a prefix of the definitive set of comments for that post. A set of \( n \) comment posts is also expected to have a contiguous range of sequence numbers from 0 to \( n \). In the table, “Wrong UID” refers to those reads whose sequence of comment UIDs either had gaps or duplicate values. “Wrong Client” refers to those reads where UIDs were contiguous, but where the wrong client’s comment was returned with a given UID. A strongly consistent BigTable service returns correct results all the time, whereas a weakly-consistent service is incorrect almost 10% of the time.

5.5.2 Queue Service

Amazon’s Simple Queue Service (SQS) [23] provides users with the ability to create an arbitrarily large FIFO queue of messages. When a client removes a message from an SQS queue, that message is considered “locked” by that client for some period of time. If the client does not explicitly indicate that the message has been processed before the specified time period has expired, the message is reinserted into the queue. We have implemented a queue service similar to SQS
This abstraction is particularly useful for environments in which a work queue must be maintained for a large collection of worker tasks which may become unavailable while processing a unit of work. Clients append messages to the queue by appending a queue entry to the object with an Append operation.

Each queue in the queue service is represented by a single object, to which clients Append new entries. Each queue entry has a timestamp field that indicates when the entry was locked by a client; this field is set to zero if no client has locked the entry. Clients remove a message from the queue by first scanning the queue (with a series of bounded ReadRange operations) for an entry whose timestamp is either zero or older than the reinsertion timeout. Once one or more such entries are located, the client locks that item by issuing a MultiOp predicated on the read version of the entry that sets the entry’s timestamp to the client’s local time. If the MultiOp to lock the entry fails, the client starts scanning the queue from the beginning again. Clients dequeue an entry by issuing a Remove operation on the entry’s key. To reduce contention with multiple readers, clients pick a random entry from the first 100 entries rather than always picking the first entry (having the clients always pick the first entry produced similar results, but was slower).

To evaluate the queue service, we have three clients spread across three data centers write a total of 1000 unique values to a queue. Thirty readers read and dequeue values synchronously from the queue as quickly as possible until the queue is empty. Table 5.6 shows a distribution of the number of times each value is read by a client using either a serial- or a weakly-consistent queue. As expected, no enqueued values are read and dequeued more than once with serial consistency. With weak consistency, however, the state of the queue at different data centers may not agree, and some values are read multiple times by different clients. Even worse, 6% of the values enqueued are never read at all due to an unfortunate interleaving of weakly consistent updates. To understand how this may occur, consider two writers that insert values \( v_1 \) and \( v_2 \) on different replicas simultaneously. Both replicas (weakly) assign these different values the same key, \( k \). One of the writers reads \( v_1 \) from its local copy, then deletes \( k \). By this time,
the consensus service has assigned \( v_2 \) key \( k \), and \( v_1 \) key \( k_1 \neq k \). Before any client has a chance to read \( k/v_2 \), the delete \( k \) is serialized and \( v_2 \) is lost.

5.6 Other Applications

In addition to the structure table and work queue applications, we have also built a file system, archival backup, and address book applications.

5.6.1 File System

We implement the file system using a single map, modeled after a traditional Unix file system. The map abstraction provides a natural fit for file systems, as disks also provide a type of map, from block numbers to blocks. Our MapStore file system contains three types of entries—inodes, data blocks, and the file system description block—shown in Figure 5.5. First, we have inode entries, which map from a unique inode id string to the inode metadata, consisting of the file type (mode), size, last modified time, number of blocks, etc. Second, we have data block entries, which have a key consisting of the inode number concatenated with the block number. Data block values contain either contents of a file or part of a

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[id]</td>
<td>[inode metadata]</td>
</tr>
<tr>
<td>[id + offset]</td>
<td>[file data, dir listing]</td>
</tr>
<tr>
<td>[root]</td>
<td>[fs metadata]</td>
</tr>
</tbody>
</table>

Figure 5.5: MapStore file system layout
directory listing with name, inode id pairs. Finally, we have a file system description block, which uses a reserved, well known key to map to a value containing the inode id for the root of the file system and other file system metadata.

Figure 5.5 shows an example set of key-value pairs for two entries. Inode 101 points to a directory, whose contents are stored under key 101:0, which is a list of file name, inode id pairs. In this example, the file foo has inode id 102. The keys 102:0 and 102:1 point to the first two data blocks for this file.

### 5.6.2 Archival backup

We wrote an archival backup service using MapStore as our storage layer. The backup service uses one map to store the contents of all files from all backups, and a separate map for each individual backup snapshot. These three maps are shown in figure 5.6.

Files are split into chunks using Rabin fingerprints [92], and then the chunks are stored into the contents map under the keys of their hashes. Content hashing [45, 80, 91] ensures that only the incremental changes need to be stored. Each backup snapshot maps the absolute path to the file to the metadata needed to reconstruct that file, including the file system metadata properties and an ordered
list of block hashes. As most files are not expected to change very much, the backup client checks whether hashes are contained in the block map before uploading new ones. The sorted property of the backup snapshot allows for depth-first in-order traversal of the file system for efficient restore operations.

Figure 5.7 shows an example layout for the backup application containing two backup snapshots from 02/01 and 02/09. Each snapshot has its own map, which contain the paths for all the files in that backup and their associated blocks. Most of the blocks are shared in the same block map between the two backups.

5.6.3 Address book

For our final example application, we look at the design of our MapStore based address book. We wanted to integrate an always-available address book service that tracks names and their associated email addresses with our email client (mutt [28]), which supports executing a program to perform an address book lookup on a substring and displaying the list of results. The address book
design uses three sorted maps to create a simple table. The first maps emails to person records which contain a person’s first and last name. The other two maps implement reverse indices keyed on first and last names, pointing to lists of email addresses sharing that name. The address book client performs a lower-bound prefix match on all three maps, issuing subsequent lookups for all email addresses returned from the two indices.

5.6.4 Limitations

Using a single map, while conceptually simple, has certain scalability limitations for the maximum size of the file system. To allow arbitrarily large file systems, we designed a second file system using multiple maps, one for each directory, and two for each file. Three types of maps represent the contents of every file and directory: directory maps, file snapshot maps, and file log maps. First, directory maps map directory names to an inode, which contains metadata (entry type, modification time, size), as well as the MapAddrs for the map(s) that store(s) the contents: directory entries recursively point to other directory maps, and file entries point to the snapshot and log maps (symlinks simply contain the path for the link). Second, the snapshot map contains the contents of the files in fixed-size blocks. For files not being written, clients read requests directly from the snapshot by computing the appropriate offsets and reading those keys. Third, the log map contains write updates for the file, which clients append to the log. Periodically, or on demand for a read, clients apply the operations from the log to both the file snapshot and the corresponding inode in the directory map, removing them from the log after succeeding. The log map requires the sorted property of our MapStore maps in order to allow clients to read the first entry from the log without knowing the actual key itself. We chose an append log-based design for files to maximize write throughput with multiple writers, as the metadata only needs to be updated when the log is applied to the snapshot. This ensures that the file snapshot and directory entry are always consistent.

As with the file system, our BigTable implementation will not scale beyond hundreds of gigabytes or possibly terabytes, so we also implemented a scalable
version of MTable that takes advantage of additional layers of indirection to split the table across multiple maps. A more scalable version uses one layer of indirection which uses the row as the key and another map as the value; the value map maps (row, column, timestamp) tuples to values. Assuming a maximum map size of 500GB, 20 columns and 3 timestamps per row, 64 byte row keys, and 64KB values, then this scheme would allow a maximum table of about 8 billion entries spanning about 870 maps using a total of about 43 terabytes of storage.

5.7 Summary

One of the goals of cloud computing is to simplify distributed application development. However, existing cloud storage offerings offer only limited interfaces, requiring complex application logic to use the cloud storage. In this chapter, we have presented a new cloud storage service, MapStore, which provides the powerful interface of a sorted map supporting fine-grained operations. Because we built MapStore using ZANTE replicated state machines, it also provides high performance, high availability, and fine-grained configurable per-object consistency. We demonstrated the power of MapStore by implementing and evaluating several distributed applications, including a structured value store, a work queue service, a file system, and archival backup.

5.8 Acknowledgment

Chapter 5, in part, is currently being prepared for submission for publication of the material. Anderson, James; Rasmussen, Alexander; Vahdat, Amin. The dissertation author was the primary investigator and author of this material.
Chapter 6

CORFU

Previously, we have described how strong consistency is achieved using a Replicated State Machine (RSM) model that uses a consensus algorithm such as Paxos to order all state machine operations [69]. We target real-world deployment with billions of objects [104] stored across tens of data centers, each with tens of thousands of machines. At this scale, there is constantly a need to recover from common machine failures.

In this chapter, we consider a particular problem with managing many RSMs across the wide area. While our work applies to any RSM consensus protocol, we operate in the context of Paxos, which employs $2f + 1$ replicas to tolerate $f$ failures. We identify a window of vulnerability where upon the $f$th failure, the state machine as a whole needs to block if RSM state is not synchronized with a new replica before a subsequent failure takes place. Thus, despite the fact that there are always at least $f + 1$ Paxos replicas available, application-level RSM state is not synchronized sufficiently quickly to safely allow continuing updates to the RSM. This window of vulnerability becomes particularly problematic when objects are large or synchronization takes place over relatively slower wide-area links: exactly the scenarios we target.

The principal contribution of this chapter is CORFU, a system architecture designed to reduce this window of vulnerability. CORFU uses a novel technique that we call local recovery, involving asynchronous replication within a data center along with synchronous replication across data centers to increase state machine
availability. Corfu further increases availability by insulating common-case machine failures within a single data center from other data centers. In particular, we employ a scalable software-based switch to hide the internal organization of storage within each data center such that common case replica failures may be recovered without requiring a replica set change operation. Local recovery can reduce recovery time by an order of magnitude, thereby significantly increasing availability of replicated strongly consistent services.

6.1 Motivation

RSMs provide the illusion of a single highly available state machine by maintaining a copy of that state machine at multiple replicas [67, 83, 98]. Any replica may receive updates at any time. To maintain the same state, each RSM uses a consistent replicated log [35, 36, 63, 70] to totally order updates with a consensus protocol such as Paxos. The replicated log assigns every update a sequential index at which it was chosen in the totally-ordered sequence of updates. The consensus safety property states that at most one update will be chosen for any index. As we focus on the Paxos consensus algorithm, we refer to this safety property as Paxos safety. The consensus liveness property states that the system can only make progress when a quorum of replicas are available. To be more precise, to maintain liveness, a quorum of replicas must be available for every index.

If we consider a permanent failure model where replicas irrevocably lose their state, such as with a total disk failure, then the system becomes vulnerable
to losing state, which would force the RSM to block forever. For example, consider the example shown in Figure 6.1 with replicas A, B, C. At time $t_0$, some instances of consensus have already completed and replica C lags behind A and B (perhaps because it never received messages for index 2 due to a transient network error). At $t_1$, replica A permanently fails. To regain fault tolerance, replicas B and C add a replacement replica $A'$ to the group at $t_2$ for index 3. $A'$ has only learned the chosen value for the index that added it to the group, but it can participate in consensus for indices $> 3$. To acquire the rest of the state, $A'$ begins copying the prior chosen values from B and C. At $t_3$, B fails and loses its state. Replica C has not yet copied index 2, and replica $A'$ has only partially copied indices 0..2. The system has lost the chosen value for index 2. To maintain safety, it must block permanently or request a human to resolve potential conflicts.\footnote{Index 3 for consensus could have blocked until $A'$ had recovered all of its state or C has copied index 2, but B's failure would still cause the system to lose state and block permanently.}

One can reduce the probability of losing state by decreasing the time between replica failure and the full availability of a new replica. Corfu’s design focuses on decreasing this time in a wide-area setting where recovery may be slowed by limited bandwidth over wide-area links.

At first glance, a simple approach for increasing system availability would be to add more replicas, thereby increasing $f$. However, the message complexity of consensus protocols is linear with the number of replicas, and the latency is bounded by the slowest replica in the consensus quorum. While these requirements may not matter when the replicas are on the same LAN, these factors can significantly hurt performance with replicas separated by high-latency, low bandwidth links. For example, with $f = 1$, the consensus latency is only that of the closer of the two replicas; increasing $f = 2$ requires waiting for two wide-area replies, with increasing possibility of one taking longer than the other. Note that this cost is borne even when there are no failures. Ideally, we want a system that does not incur this additional cost in latency or bandwidth.

In a wide-area setting, a better approach is to localize recovery of a replica within the site of the failed replica. This way, the number of wide-area message exchanges is kept small in both the failure-free case and when there are failures.
This implies a hierarchical approach: Paxos-based replication across sites and some kind of replication within sites. For replication within a site, using a consensus protocol such as Paxos would both add latency to client requests and require additional nodes \((2f + 1\) for each data center). Hence, Corfu uses asynchronously updated backups and a distributed software switching service to avoid adding any additional latency in the failure-free case. Furthermore, massive failure within a site can be masked by using recovery with the cross-site replication protocol. The resulting architecture dramatically increases the availability of the RSM without adding any latency in the failure-free case.

While we consider a permanent failure model in this paper, we note that, in the common case, it may be possible to simply wait for the machine to reboot to recover missing state. In this scenario, the amount of wide-area communication required to re-synchronize the replica would be limited to any missing updates during the reboot period. Unfortunately, machine reboot can take several tens of seconds in the best case and since the fundamental problem with the window of vulnerability remains, the system as a whole could become unacceptably prone to unavailability as further analyzed in §6.5.

6.2 Overview

Before describing the protocol in detail, we provide our system model and an overview of Corfu.

6.2.1 System Model

Corfu provides the abstraction of a collection of highly-available RSMs, which higher-level services can read or update. Each RSM has its own Paxos-based replicated log and associated replica set. The state machine stores the persistent state and application data needed by cloud-based services. To meet the demands of typical applications deployed across data centers, Corfu must be able to support a large number of RSMs simultaneously; for example, one might imagine the billions of objects stored in Amazon’s S3 [104] each being backed by an RSM.
To achieve its target of high availability, Corfu replicates each RSM across multiple data centers. We assume that each data center contains up to tens of thousands of machines [39], and that each machine may be responsible for hosting the relevant application and replicated log state for many RSMs.

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

We assume links connecting data centers, or RSM replicas, are well provisioned and do not contribute to any significant downtime. For simplicity, we ignore link downtime from our analytic evaluation.

**Failure and Recovery Model** Traditionally, consensus protocols assume nodes fail by crashing, i.e., a machine will fail by becoming unresponsive, but may later recover with its state intact. In this work, we assume a stronger notion of failure in which a machine irrevocably loses all of its state when it fails (such as a disk loss). We assume such a failure model because it is a pessimistic assumption that allows us to explore worst-case recovery times (§6.5). Specifically, we define a failed server to be one that some other server believes to have crashed and will never recover. As such, failed servers must be replaced to maintain the same degree of fault tolerance. Like Paxos, Corfu does not require a perfect failure detector.

**Consistency Model** Unlike other wide-area replicated services that offer high availability but only provide weak consistency guarantees [48], Corfu uses a consensus algorithm to provide strong consistency. Requests to a given RSM are sequentially consistent, i.e. totally ordered.

### 6.2.2 Corfu Overview

The primary contribution of Corfu is a technique that we call local recovery for increasing availability of strongly consistent services replicated across
the wide-area by reducing recovery time, which imposes no additional latency over standard consensus. This technique leverages one or more backup servers located in the same data center as each replica, or primary. The primary asynchronously sends updates to the backups for values in the replicated log. CORFU uses an asynchronous replication method to avoid additional latency on the critical path for client operations; however, this means that the backups may lag slightly behind the primary. Thus, we leverage a distributed software switching service as the final component to enable local recovery. The switching service directs messages to the appropriate servers, the same as front-end load balancers present in most data centers. In addition, the switching service also acts as a message cache for certain Paxos messages sent by the primary. Should a primary fail, CORFU can promote one of its backups to become the new primary by sending it the Paxos state in the switch message cache. This local recovery is transparent to the other replicas and can be completed in milliseconds for even very large objects, requiring no wide-area messages.

6.3 CORFU Protocol

We describe CORFU in a progression of four steps, starting from the Paxos state machine implementation [67].

Step 0: Paxos

The Paxos consensus algorithm can be used to consistently order updates for an RSM. For every index, Paxos provides the safety property that at most a single value will ever be chosen among a set of $2f + 1$ replicas and can maintain liveness with up to $f$ crash-failures. Briefly, Paxos runs in two phases: 1) A proposer collects a quorum of Promise messages in response to a Prepare; and 2) then sends Accept messages with either a value returned in a Promise response or with an unconstrained value otherwise. A value is chosen when it has been accepted by a quorum of replicas. If more than $f$ Paxos replicas permanently fail simultaneously, then Paxos must block forever to maintain safety. Paxos and
Corfu also support the optimization of a leader to avoid the first phase after the leader has been elected. The leader also serves as a distinguished learner that sends Chosen messages to the other replicas after receiving a quorum of Accepted responses.

Before we begin, we describe two changes to Paxos. First, we make a small optimization so that if an acceptor receives a Prepare or Accept for which its learner knows the chosen value, it responds with a Chosen message containing this value. Second, in Paxos, if a server believes that another server $S$ has failed, it may propose a command to change the replica set by removing $S$ from the state machine and adding some other available server $S'$. Eventually, $S$ is removed and $S'$ is initialized and added to the state machine. We call this a Paxos reconfiguration [68]; it is described in more detail in our tech report [25].

In a Paxos RSM, the acceptor, proposer, and learner combine to form a Paxos server, or replica. In Corfu, each replica has three designated processes for each data center: a primary, a backup, and a switch. These processes run on separate machines in the same replica site (data center). Figure 6.2 illustrates our model for three sites, each with one replica. We describe the state shown with each process in Step 4. We assume each site has a pool of machines available to host replicas.

The primary implements the traditional Paxos server. The primary can also replace its switch should it believe that it has failed. A backup is a standby used to speed the replacement of a failed primary. The switch intercepts and forwards Paxos messages between its primary and the switches at the other replica sites. The switch works with the backup to enable local recovery: recovery without the reconfiguration and state transfer required in Paxos.

**Step 1: Role of the switch**

In the first step, for simplicity, the backup is not updated with any state until the primary fails. The switch intercepts Paxos messages to recover the primary should it fail. To recover a primary, the new replica must have the same learned values and acceptor state as the failed primary, as determined by the messages it
has sent the switch. Note that if the failed acceptor has made a promise or accepted a value but it has not sent a message reflecting this change, then we do not need to recover this update, as it could not have been observed by the rest of the system. The switch saves outgoing PROMISE and ACCEPTED messages. Initially, we also have the switch save incoming CHOSEN messages, but we will eliminate this requirement in Step 3.

Once the switch believes that the primary has failed, it isolates the primary by suspending message forwarding. This site will not participate in consensus or respond to client requests until recovery completes. The switch will never again forward any messages to or from the “failed” primary, so that even if the server has not actually failed, it cannot interfere with the new replica group (it will eventually be notified of its removal and terminate). The switch then promotes the backup by sending it the stored messages. The backup applies these messages to its state, in the order received by the switch, as follows. For Paxos index $i$, the backup keeps a tuple $(\text{BallotPromised}, \text{ValueAccepted}, \text{BallotAccepted})$ and the learned state. For each PROMISE message, the backup updates the promised ballot for that index. For each ACCEPTED message, the backup updates the value and ballot accepted for that index. For each CHOSEN message, the backup adds the value to the learned

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig_corfu_overview.png}
\caption{Corfu protocol overview}
\end{figure}
state. Upon applying all messages, the backup will have the same state as the failed primary. This new primary notifies the switch that recovery has completed, and the switch resumes forwarding messages. Client requests and Paxos messages from other sites will now be forwarded to the new primary.

If the primary believes its switch has failed, it replaces the switch and initializes it. The new switch blocks by not forwarding any messages until the primary has sent it a copy of its state. This is important: if the switch tries to recover a backup before having all the primary’s state, the backup will not have the appropriate state to maintain safety. For each index that the primary has learned the chosen value, it sends a CHOSEN message. For all other indices, the primary sends ACCEPTED and/or PROMISE messages.

If the switch and primary both fail, then eventually a server at another replica site will detect the failure and replace them through the standard Paxos reconfiguration and wide-area state transfer.

Step 2: Limiting switch state: discarding prefixes

In the second step, we limit the amount of state that the switch needs to store. Upon learning a chosen value, the primary sends that value to the backup. When the backup has received the chosen values for all contiguous indices through \( i \), it notifies the switch. The switch remembers this value \( i \) in a state variable \( \text{MaxForgotten} \), initialized to 0, and may then discard its Paxos messages for all indices through \( i \).

If the primary believes the backup has failed, it initializes a new one by sending it all of its chosen values.

If the switch believes the primary has failed, it must first check that the backup is itself not in the process of recovering so that local recovery may be performed. It does this by sending the backup \( \text{MaxForgotten} \). If the backup has all the chosen values through index \( \text{MaxForgotten} \), then it will copy the switch messages as discussed in Step 1 and assume the role of the primary.

If the backup is missing chosen values for any of these indices (because it was replaced after a failure but had not completed recovering) then local recovery
cannot be performed. The backup will be promoted to a primary through a Paxos reconfiguration and initialized through state transfer from the other replica sites.

After a backup assumes the role of primary—either through local recovery or a Paxos reconfiguration—it will initialize a new backup. Recovering from switch failure is identical to Step 1.

**Step 3: Limiting switch state: discarding CHosen**

In Step 3, we remove the requirement for the switch to store CHosen messages, further decreasing the amount of state stored. We expect the backup has most of the chosen values, but it might be missing some because the primary does not synchronously update it. However, the switch will contain the Paxos messages for all of these indices, as they are guaranteed to be greater than \textit{MaxForgotten}. Thus, when being promoted to primary, the backup will acquire the same acceptor state as the failed primary even if it does not have the same learned state for some indices. This new primary will be able to learn the chosen values after recovery completes, e.g., by running instances of Paxos for these indices.

**Step 4: Limiting replica state with snapshots**

Now that we have shown how to limit the state stored by the switch, we introduce application snapshots that allow primaries and backups to discard Paxos state. We define the \textit{snapshot} at index $i$ as the state resulting from executing all state machine commands through index $i$. Replicas periodically exchange the indices of their snapshots. When a replica learns that a quorum of replicas have a snapshot at index at least $i$, then it may discard all Paxos state up to $i$. Replicas will no longer respond to Paxos messages for indices that have been discarded; rather, we use a snapshot state transfer mechanism similar to [63, 70].

Let $k$ be the highest index proposed by the system, $i$ be the primary’s snapshot index, and $j$ be the backup’s snapshot index. \textsc{Corfu} maintains the invariant that for any site, $\text{MaxForgotten} \leq j \leq i \leq k$ (illustrated in Figure 6.2). This property states that the messages stored in the switch plus the snapshot and chosen values stored by the backup will equal the required primary state for local
recovery. When a primary initializes a new backup, it copies its snapshot and any
chosen values larger than the snapshot index. Should the primary fail before this
copy completes, the backup notifies the switch that it does not have the state prior
to MaxForgotten and will fall back to a Paxos reconfiguration and state transfer
from remote replicas across the wide-area.

6.4 Design Details

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

6.4.1 Controller

As described in §6.2.1, we require Corfu to support potentially billions of
RSMs, and each site will have its own primary, backup, and switch servers for every
RSM. To coordinate the assignment of server processes to physical machines and
RSMs to servers, we use a logically centralized controller per data center site that
has knowledge of the pool of available machines in the data center. The controller
maintains two key tables. The first maps server processes to physical machines.
The second maps RSMs to their appropriate servers. A centralized implementation
does not bottleneck the system because the controller does not participate in most
RSM requests. Rather, the controller is only consulted when an RSM is created or
destroyed or a switch needs to look up an RSM-to-server mapping. The controller
also initiates recovery, as described below, when machines fail.

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

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

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

6.4.3 Failure Detection

Corfu uses an active heartbeat-based failure detector [99] as part of a failure detection service that monitors every machine within a site. If a machine has not sent a heartbeat within a given timeout, the system considers it failed; we do not distinguish between actual failure and slowness for local failure detection. If a failed server resumes sending heartbeats, the failure detector instructs it to deallocate its local state so that it can be reassigned to another RSM. Because the switch forwards all messages to the new primary, a server that was incorrectly declared to have failed cannot interfere with consensus; it will never receive any new consensus messages or RSM operations. Switches believed to have failed are similarly isolated.
6.5 Analysis

In this section, we evaluate the benefits of Corfu, in terms of reducing the window of vulnerability, by comparing with two other design alternatives.

6.5.1 Design Alternatives

In Corfu's design, the application and Paxos state stored in every primary is replicated either on a local backup or on a local software switch. We evaluate the significance of each of these components by comparing the LocalRecovery provided by Corfu with two other designs—NoBackup and WARBackup.

In the NoBackup design, an RSM's state resides on one machine in each data center. If a replica in one data center fails, the service will execute a Paxos reconfiguration to include a new replica replacing the failed one. After reconfiguration completes, the new replica will restore its state from replicas at another data centers.

The WARBackup design reduces recovery time by having several machines host the state machine at each data center—one serving as the primary replica and the others acting as backups. This design is essentially LocalRecovery without the switches, so every primary failure entails a Paxos reconfiguration. Since the promoted backup may not have the complete state of the failed primary, some state may still need to be copied from a remote data center before the new primary can participate in operations of the RSM. If all of the backups fail before the new primary can be brought up-to-date, the system will need to fall back to the
The NoBackup design. Figure 6.3 shows the advantages of our LocalRecovery design by comparing the total time taken to recover, as well as the window of vulnerability for losing state during recovery. We note that contention for the next available index may prolong the reconfiguration window (§6.6.3) for NoBackup and WARBackup.

6.5.2 Markov Models

To compare our LocalRecovery design with other design alternatives, we constructed a Markov model for each design; see [57] for details on the modeling framework. The goal of these models is to compare the designs in terms of the availability and mean time to first failure (MTFF) of RSMs. An RSM becomes unavailable when it reaches a state wherein proceeding with operations on the RSM will violate Paxos safety. To reflect this state of unavailability, we use the expected time until first failure, MTFF = \(E[T_{FF}]\), as our primary metric, and by introducing manual repair from the failed state we also obtain the availability for the different designs.

![Markov model for NoBackup](image)

Figure 6.4: Markov model for NoBackup

An RSM is distributed over \(2f+1\) data centers. In the models however, we assume a fixed \(f = 1\), except for NoBackup where we also consider \(f = 2\). All state is stored on one primary per data center. Each machine may act as the primary for multiple RSMs. In the WARBackup and LocalRecovery designs, there is also one backup for each RSM per data center.

In modeling the different designs (Figure 6.4 and Figure 6.5), we examine all the operational states (denoted by circles) and the down states (squares). Each up
Figure 6.5: Markov model for WARBackup

state is identified by $XpYb$, where $X$ and $Y$ represent the number of primaries and backups, respectively. States annotated with a * indicate that wide-area recovery is necessary. In the models, the order in which the replicas fail is not important, and hence we can represent several failure modes with fewer states. The failure rates, denoted $\lambda$, out of a state represent the accumulated failure rate in this state. In each model different failure rates lead to different states, except NoBackup where there is only one failure rate, weighted by the number of replicas that may fail from that state. The repair rates, denoted $\mu$, represent the expected time to recover from a state with failures to another state with fewer failures. To determine the availability of a design we introduce $\mu_m$, denoting manual repair from a down state to the fully operational state. Table 6.1 describes the notation used in the models.

The model for LocalRecovery involves three different entities and thus more complicated failure patterns appear. We programmatically generated a model with 512 states.
Table 6.1: Description of failure/repair rates.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda_p$</td>
<td>Failure rate of a primary</td>
</tr>
<tr>
<td>$\lambda_b$</td>
<td>Failure rate of a backup</td>
</tr>
<tr>
<td>$\lambda_s$</td>
<td>Failure rate of a switch</td>
</tr>
<tr>
<td>$\mu_{pb}^{-1}$</td>
<td>Time to promote backup to primary</td>
</tr>
<tr>
<td>$\mu_{np}^{-1}$</td>
<td>Time to replace primary (wide-area copy)</td>
</tr>
<tr>
<td>$\mu_{nb}^{-1}$</td>
<td>Time to replace backup (local copy)</td>
</tr>
<tr>
<td>$\mu_{ns}^{-1}$</td>
<td>Time to replace a switch (local initialization)</td>
</tr>
<tr>
<td>$\mu_{tr}^{-1}$</td>
<td>Time to perform local recovery</td>
</tr>
<tr>
<td>$\mu_m^{-1}$</td>
<td>Time for manual repair</td>
</tr>
</tbody>
</table>

6.5.3 Analysis Results

Using the Markov models described above, we evaluate their expected time to first failure and unavailability under three different operating regimes. The results were obtained using a Mathematica package for symbolic and numerical dependability analysis of Markov models.

We assume an idealized scenario in which no logical failures or failure of the recovery mechanism takes place. We also assume a pessimistic MTTF of 50 days [46], i.e., $\lambda_p^{-1} = \lambda_b^{-1} = \lambda_b^{-1} = 50d$. Further, we fix the parameter for manual repair to a generous $\mu_m^{-1} = 5h$ for both designs. In the WARBackup evaluation, we fix the time to replace a primary across the wide-area to $\mu_{np}^{-1} = 5h$, and to replace a backup from a local primary to $\mu_{nb}^{-1} = 1h$. This represents an RSM with a large state, requiring significant time to recover.

The results of our analysis is given in Tables 6.2, 6.3, and 6.4. For the NoBackup design, $\mu_{np}^{-1}$ varies from 5 hours to 5 minutes, representing different RSM state sizes, by the time it takes to recover from a failure over the wide-area.

For WARBackup, we employ a $\mu_{pb}^{-1}$ that varies from 5 minutes to 10 seconds, representing the time to promote a backup to become primary.

We evaluated LocalRecovery (Table 6.4) using two ways of computing availability. In the first model, we treat the state in which the primary has failed but the
Table 6.2: MTFF and unavailability for NoBackup

<table>
<thead>
<tr>
<th>( \mu_{np}^{-1} )</th>
<th>MTFF</th>
<th>( U )</th>
<th>MTFF</th>
<th>( U )</th>
</tr>
</thead>
<tbody>
<tr>
<td>5h</td>
<td>5.6y</td>
<td>1.0 ( \cdot 10^{-3} )</td>
<td>136y</td>
<td>4.2 ( \cdot 10^{-6} )</td>
</tr>
<tr>
<td>1h</td>
<td>27.5y</td>
<td>2.1 ( \cdot 10^{-4} )</td>
<td>3310y</td>
<td>1.7 ( \cdot 10^{-7} )</td>
</tr>
<tr>
<td>5m</td>
<td>329y</td>
<td>1.7 ( \cdot 10^{-6} )</td>
<td>473687y</td>
<td>1.2 ( \cdot 10^{-9} )</td>
</tr>
</tbody>
</table>

Table 6.3: MTFF and unavailability for WARBackup

<table>
<thead>
<tr>
<th>( \mu_{pb}^{-1} )</th>
<th>MTFF</th>
<th>( U )</th>
</tr>
</thead>
<tbody>
<tr>
<td>5m</td>
<td>329y</td>
<td>1.7 ( \cdot 10^{-6} )</td>
</tr>
<tr>
<td>1m</td>
<td>1644y</td>
<td>3.5 ( \cdot 10^{-7} )</td>
</tr>
<tr>
<td>10s</td>
<td>9863y</td>
<td>5.8 ( \cdot 10^{-8} )</td>
</tr>
</tbody>
</table>

switch and backup have not as a down configuration for that data center. This is technically accurate, because only the primary can respond to requests. However, the backup and switch together have the same state as the failed primary, so we also evaluated the model treating this state as an available state, given that local recovery will complete in a few seconds at most to restore the primary. For both models, we use the same parameters as for WARBackup and set \( \mu_{ns}^{-1} \) to 10 seconds and \( \mu_{tr}^{-1} \) to 3.6 seconds. Although our experimental evaluation shows that local recovery itself can be completed in tens of milliseconds, we used this larger value of 3.6 seconds to account for failure detection time.

As the results show, the MTFF for NoBackup with \( f = 1 \) is significantly shorter than WARBackup. Although NoBackup seems to offer “good enough” numbers for a practical deployment, recall that these numbers represent idealized failure assumptions and thus it is desirable to further increase the MTFF. The distinguishing characteristic between these designs is the time to recover from a
Table 6.4: MTFF and unavailability for LocalRecovery

<table>
<thead>
<tr>
<th>LocalRecovery (f = 1)</th>
<th>MTFF</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switch + Backup Down</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>2961y</td>
<td>1.5 · 10^{-10}</td>
</tr>
<tr>
<td>No</td>
<td>1.97 · 10^6y</td>
<td>7.5 · 10^{-11}</td>
</tr>
</tbody>
</table>

failure. For comparison, we also evaluate the NoBackup design with $f = 2$, i.e., replicas at five data centers. We note that both models for LocalRecovery (with $f = 1$) yield better availability than even NoBackup with $f = 2$.

Other factors not accounted for in our analysis, such as correlated failures, logical failures, power outages, and security threats will likely lead to significantly lower availability than what is shown in Figure 6.2. However, what our analysis does show is that LocalRecovery significantly reduces recovery time, thus shifting the “bottleneck” for RSM availability to these other factors.

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

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

WARBackup and LocalRecovery provide a further benefit by minimizing
disruption experienced by clients. Services try to direct clients to the data center that will provide them the best performance, often significantly reducing the latency for client requests. While the primary at that data center is recovering, client requests must be redirected to a different replica, increasing the network delay for those requests. Because the LocalRecovery design can recover very quickly, on the order of milliseconds, a small amount of buffering the software switches will allow services to completely mask a primary failure from the client, providing significantly improved responsiveness and performance.

6.6 Evaluation

With our experimental evaluation, we hope to demonstrate that 1) Corfu significantly decreases the time to recover relative to other techniques and 2) Corfu increases the availability of all RSMs for extended deployments. We implemented Corfu and a replicated object store using the C++-based MACE [62] programming language and toolkit for distributed systems.

6.6.1 Object Store

We implemented a simple object store using our Paxos and Zante services. The object store service provides highly available storage for arbitrary blob objects on top of Corfu. Clients use this service through the following API:

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

6.6.2 Experimental Setup

We use ModelNet [108] to simulate the environment necessary to perform our evaluation. In our experimental setup, three data centers (A, B, and C) are
pairwise-connected by 100 Mbps links. To simulate a realistic geographic separation between data centers, we assign 10ms of latency between $A$ to $B$, 20ms between $A$ and $C$, and 50ms between $B$ and $C$. Each data center consists of one switch and four servers each running on dedicated physical machines. For experiments using backups, we use one backup per RSM for each data center. Clients are connected to data centers over an 8 Mbps, 2ms link, and are multiplexed onto three machines, up to four clients per machine.

We have configured ModelNet so that traffic local to a data center just uses the physical nonblocking GbE switch rather than going through the ModelNet core. Each machine is powered by a 2.13 GHz quad-core Intel Xeon processor and has 4 gigabytes of RAM and a 1 Gbps Ethernet.

To evaluate the three different recovery designs, we use the same system implementation with varying configurations for the number of backup servers, including none, which we support in the controller service. Because we use the same system for all designs, our evaluation includes a version of $NoBackup$ and $WARBackup$ using switches (that do not maintain Paxos state), even though these designs could also operate without a data center application-layer switch. We note where the switch causes an effect on the results throughout this section.

### 6.6.3 Time to Recover

We measured the breakdown of recovery time for the three design points $NoBackup$, $WARBackup$, and $LocalRecovery$, for two different object sizes of 1MB and 100MB. For each of these six experiments, a single client connected to data center $B$ begins issuing 4KB pipelined updates with up to 64 outstanding, up to a total of 15000 updates. At approximately 10 seconds into the run, we kill the server acting as the primary replica at data center $B$. Figure 6.6 shows the resulting recovery timeline for the six experiments.

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

To understand the timing for each phase of recovery, we plot timeline graphs for each experiment. Local recovery completes in just two phases: 1) time to detect failure, and 2) copy Paxos state (too small to plot on a second time scale). We break wide-area recovery into five phases:

1) **Failure Detection** Approximately 500ms.

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

3) **Unsuccessful Replica Set Change Attempts** A replica set change proposal may not be chosen at the requested index because of contention with other pro-
posals. This contention is still possible even when using a leader that limits its number of outstanding proposals to some fixed amount $\alpha$. One of the other replicas may be lagging, then believe that the leader has failed and attempt to initiate a membership change. Because the replica was lagging, it does not know the last index at which the leader issued a proposal, so it cannot compute the correct next uncontested index, even though it knows that should be $\alpha$ after the last index.

4) **Successful Replica Set Change** This phase shows the time for three-phase Paxos to choose the replica set change for an uncontested index. At the completion of this phase, the replacement replica has been added to the replica set.

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

Figure 6.7 shows the effect of the replica’s failure on the client’s throughput. The failure of the client’s local replica represents the worst possible failure condition from that client’s perspective, as some of the client’s outstanding requests are lost when the failure occurs. Throughput is sampled once per second. The effect of the failure is much more pronounced for a 100MB object, as the service is unavailable from the client’s perspective while the wide-area data transfer portion of recovery is taking place. We note that for any of the recovery scenarios, the client could have maintained throughput even during recovery by sending its requests to one of the other replicas.

### 6.6.4 Model Validation

Our model in §6.5 showed that the probability of an RSM becoming unavailable is several orders of magnitude greater with either NoBackup or WARBackup relative to LocalRecovery. To validate our analysis, we compared the operation of the system with the three different designs.

To evaluate an actual deployment of the system, we instantiated three data
Figure 6.7: Client throughput during failure of its local replica.
Figure 6.8: Failure and recovery for *NoBackup* (left) and *LocalRecovery* (right).

centers, each with 11 physical servers. One server at each data center executed the software switching service with 10 switch processes. The remaining 10 servers at every data center each executed processes corresponding to 100 machines. Therefore, our experiment is representative of each data center provisioned with 10 machines serving as switches and 1000 machines serving as primaries and backups. We bootstrapped this deployment with 5000 objects, each of size 1MB, thus having every machine serve as the primary for an expected 5 objects. To account for the fact that the number of objects for which a server is assigned as the primary is significantly less than what one would expect in a real deployment, we limit the bandwidth between data centers. We do so by capping the bandwidth between all pairs of servers in the switching infrastructure to 10Mbps.

We then run the above setup for 300 seconds. To have this period emulate a day’s worth of operation, we proportionately reduced the MTTF of every machine from 50 days to 4 hours. Therefore, every second, we kill any server with a probability of \( \frac{1}{(4 \times 3600)} \). Failed servers are not re-introduced into the setup. Over the course of execution we track the failures of servers and the recovery of RSMs
that were consequently triggered.

We ran the above experiment with two system designs—NoBackup and LocalRecovery. We do not consider the WARBackup design for this experiment since the primary contribution for the window of vulnerability in that case is the replica set change, and our setup would not reflect that latency since we do not model the wide-area latency between data centers. Figure 6.8 shows the results of our validation.

Figure 6.8a depicts cumulative frequency of events for the two configurations, and Figure 6.8b shows these events on a per-object granularity throughout the run. With the LocalRecovery design, no object suffered a quorum of replica failures, whereas 136 objects became unavailable with the NoBackup design. To interpret Figure 6.8b, recognize that each column represents the failure of a server. Each server has some number of objects, and for CORFU, serves as both a primary and backup for different objects. When the server fails, those objects need to be recovered. The plot for LocalRecovery shows the distribution of those recoveries, as no objects were lost. For NoBackup, each square shows an object that was successfully recovered, or an “x” marks the failure that renders the object unavailable.

We observe that for NoBackup, as the run progresses, the success rate of recovery declines until almost every failure results in the loss of objects. This is due to the high load caused by running 10 switch servers on a single physical machine. Even though objects are being lost, a given site cannot distinguish this from other sites being slow or a transient network error. Thus, the remaining site(s) continue to periodically attempt recovery of the objects, even though that recovery will never succeed. Because the machines running the switches are already operating near maximum capacity, these extra futile recovery attempts end up causing queues to grow and all messages to be delayed. Consequently, objects that might otherwise have been able to recover are lost because of the additional delay; these failures, in turn, add yet more load to the system. Although we believe that for this particular experiment using more machines to run the switches could help alleviate this problem and would show a more even distribution of failures and recoveries, this experiment highlights a more general and very difficult problem:
how often should a replica attempt to recover a failed object if the first attempt
does not succeed? Delaying subsequent recovery attempts will increase the window
of vulnerability assuming that the first attempt(s) failed because of a transient
condition. However, retrying too often adds additional load to the system and
requires more bandwidth and server resources. We leave this fundamental tension
open to future research.

6.7 State Transfer

In this section, we describe how to perform state transfer to recover a replica
in Simple Paxos.

Consider three replicas \{A, B, C\} running an instance of Paxos [67]. Ini-
tially, the three acceptors all have \((ballot, value, promise)\) equal to \((0, \bot, 0)\).

Suppose there are two proposers. One proposer \(P_1\) sends 1A message (1)
and the other proposer \(P_2\) sends 1A message (2). Let \(A\) receive the first one and
\(B\) receive the second: \(A\)'s acceptor is \((0, \bot, 1)\) and \(B\)'s acceptor is \((0, \bot, 2)\).

Consider three possibilities:

1. The 1A message from \(P_2\) is delayed to \(A\) and \(C\). \(C\) received the 1A message
   from \(P_1\). \(P_1\) has constructed the quorum \\{A, C\} and sends the 2A message
   \((1, 1)\) which \(C\) receives. So, \(C\)'s state is \((1, 1, 1)\). When \(A\) receives the same
   message, 1 will be decided.

2. The 1A message from \(P_2\) is not delayed to \(C\). \(P_2\) has constructed a quorum
   \\{B, C\} and sends the 2A message \((2, 2)\) which \(C\) receives. So, \(C\)'s state is
   \((2, 2, 2)\). When \(B\) receives the same message, 2 will be decided.

3. \(C\) behaves like one of the scenarios above and then crashes. When \(C'\) joins
   in a later instance of Paxos, it obtains \((0, \bot, 1)\) from \(A\) and \((0, \bot, 2)\) from
   \(B\). \(C'\) cannot distinguish between the two cases, and so cannot safely set it's
   acceptor state.

What could work is for some proposer, say \(P_3\), to send the 1A message
\((3)\) to \\{A, B\} and then send the appropriate 2A message to \\{A, B\}. Let \(C'\) be
a learner for this Phase 2. Once $C'$ receives the 3A messages from $A$ and $B$, $C$ knows the consensus value for that instance.

Note that there is another possibility in which $C$ did not receive any 2A message. Thus, consensus was not reached. The above forces consensus to be reached.

So, state transfer is the following. If the instance $C'$ joins is $j$, then each replica $R$ sends a 3A message for instance $j1$ to $C'$ that contains, for each instance $i < j1$:

- the value decided for $i$ if $R$ knows it
- $\bot$ otherwise.

When $C'$ receives the 3A messages from a majority, it learns (a) that decision value for instance $j1$ and (b) for each instance $i < j1$, the decision value if any replica knew it. Otherwise, that instance is undecided.

For each undecided instance $u$, $C'$ runs Paxos on that instance with $\{A, B\}$ and acts as a learner for that instance. It continues to do this until it learns the decision value for $u$.

6.8 Summary

We considered a practical problem in the deployment of replicas in a distributed service that supports strong consistency: how does one recover from server failures? We show that the solution to this problem has a significant impact on the availability of the service. Our solution, Corfu, provides a highly scalable implementation of local recovery that reduces the time for individual failures by more than an order of magnitude over conventional recovery techniques. We show that it is better than the obvious approach of increasing the number of replicas a small amount, which decreases the probability of critical failures at a cost of more replication. We derive the architecture of Corfu from Paxos and evaluate the resulting availability both analytically and under emulation. Our results show that Corfu is able to reduce the recovery time for individual failures by more
than an order of magnitude over conventional recovery techniques, thus providing high availability. We are able to achieve these availability improvements while still maintaining high performance.

6.9 Acknowledgment

Chapter 6, in part, is currently being prepared for submission for publication of the material. Anderson, James; Rasmussen, Alexander; Meling, Hein; Madhyastha, Harsha; Marzullo, Keith; Vahdat, Amin. The dissertation author was the primary investigator and author of this material.
Chapter 7

xOMB

Previously, we introduced the software switching and message routing service needed by systems such as ZANTE, MapStore, and Corfu. In this chapter, we describe our design and implementation of such a service, and more generally, an approach for building scalable, extensible middleboxes and network appliances running on commodity hardware.

Network appliances and middleboxes performing forwarding, filtering, and transformation based on traffic contents have proliferated in the Internet architecture. Examples include load balancing switches [6–8] and reverse proxies [3, 11, 18, 20], firewalls [17, 89], and protocol accelerators [14, 18]. These middleboxes typically accept connections from potentially tens of thousands of clients, read messages from the connections, perform processing based on the message contents, and then forward the (potentially modified) messages to destination servers. While prior work [49, 79] examined programmable and scalable packet-oriented routers, this chapter focuses on forwarding logical application-level messages within byte streams.

Middleboxes form the basis for scale-out architectures in modern data centers, being used in three main roles. First, they perform static load balancing and possibly filtering, whereby they distribute (or drop) messages to server pools based on a fixed configuration. For example, a load balancing switch might forward requests to different server pools based on URL. Second, middleboxes perform dynamic request routing and application integration, where they execute service logic
and use dynamic service state to forward requests to specific application servers and often compose replies from many application servers for the response. An example would be front-end servers that use object ids to direct requests to the back-end servers storing the objects. Third, middleboxes perform protocol acceleration by caching/compressing data and responding to requests directly from their caches. An example would be a service deployed across multiple data centers using middleboxes to cache content from remote data centers.

Unfortunately, the architecture of commercial hardware middleboxes consists of a mixture of custom ASICs, embedded processors, and closed-source, proprietary software with, at best, limited extensibility. While firewalls, NATs, load balancing switches, VPN gateways, protocol accelerators, and other middleboxes perform logically similar functionality, they are individually designed by niche providers with non-uniform programming models. These boxes often command a significant price premium because of the need for custom hardware and software and their limited production volumes. Extending functionality to new protocols may require new custom hardware. Worse, expanding the processing or bandwidth capacity of a given middlebox may require replacing it with a higher-end model. Similarly, software reverse proxy middleboxes are specialized for specific protocols and, like their hardware counterparts, offer limited scalability and extensibility.

At first blush, software routers such as Click [79] or Route Bricks [49] may be employed to achieve extensible middlebox functionality. However, these pioneering efforts are focused on per-packet processing, making them less applicable to the stream or flow-based processing common to the class of middleboxes we target in this work. For example, flow-based processing requires operating on a byte stream rather than individual packets and may require communication among multiple network elements to perform dynamic forwarding and rewriting. Further, they provide no specific support for managing and rewriting a large number of concurrent flows, instead focusing on high-speed pipeline processing of individual packets.

Hence, this chapter presents xOMB (pronounced zombie), an eXtensible Open MiddleBox software architecture for building flexible, programmable, and
incrementally scalable middleboxes based on commodity servers and operating systems. xOMB employs a general programmable pipeline for network processing, composed of xOMB-provided and user-defined C++ modules responsible for arbitrary parsing, transforming, and forwarding messages and streams. Modules can store state and dynamically choose different processing paths within a pipeline based on message content. A control plane automatically configures middleboxes and monitors both middleboxes and destination servers for fault tolerance and availability. xOMB provides a single, unified platform for implementing the various functions of static load balancing/filtering, dynamic request routing, and protocol acceleration.

Several additional xOMB features add power and ease of programming to the simple pipeline model. Asynchronous processing modules and independent, per-connection processing efficiently support network communication (e.g., to retrieve dynamic state) as part of message processing. Arbitrary per-message metadata allows modules to store and pass state associated with each message to other modules in the pipeline. Finally, xOMB automatically manages client and server connections, socket I/O, message data buffers, and message buffering, pairing, or reordering.

We implement and evaluate three sample pipelines: an HTTP load balancing switch (static load balancing), a front end to a distributed storage service based on Eucalyptus [82] implementing the S3 interface [1] (dynamic request routing), and an NFS [97] protocol accelerator. Forwarding through xOMB presents little overhead relative to direct access to back-end servers. More importantly, xOMB scales its network and processing performance with additional commodity servers and provides transparent support for dynamically growing the pool of available middleboxes. We also compare the performance of our xOMB load balancing switch against a commercially available hardware load balancing switch and the leading open source reverse proxy. xOMB matches or outperforms the commercial switch and open source proxy in most cases, while providing a more flexible and powerful programming model.
7.1 Overview

Middleboxes can be defined as network elements that process, forward, and potentially modify traffic on the path between a source and a destination. With this generic definition, routers and switches can also be classified as middleboxes. We identify three key features differentiating middleboxes from traditional routers and switches: middleboxes i) understand the application semantics of network data, ii) may modify or even completely replace the contents of that data, and iii) despite being “in the middle,” middleboxes may terminate connections and initiate new connections.

In the traditional picture of network communication, such active processing was left to end hosts explicitly named as the destination for a particular network packet [96]. Functionality in the middle of the network was explicitly defined to be simple; routers and switches were responsible for forwarding packets to their final destination. This picture has shifted to a landscape replete with middleboxes because: i) managing functionality such as security can be simpler at a central point where it is guaranteed to take place for all network communication, ii) service-oriented architectures, scale-out architectures, and extreme availability requirements mean that dynamic pools of servers provide the functionality traditionally named by a single IP address, and iii) routers and switches now have the requisite processing power to perform active packet processing at line rate within enterprise and data center networks.

Although the xOMB design is general to a variety of middlebox applications, we primarily focus on their use in data centers in this paper as this deployment scenario stresses all aspects of our design and presents a particularly challenging use case, with strict requirements for performance and reliability. We begin by examining load balancing switches (LBSs), specialized middleboxes widely used to distribute requests among dynamic server pools in data centers.
7.1.1 Load Balancing Switches

Companies use data centers as platforms for deploying Internet services and applications to a set of global users. These services may have hundreds of millions of users and receive billions of requests per day, and may be served from one or more data centers. Each data center contains collections of application servers and storage servers. These services are designed for incremental scalability, using scale-out architectures that increase capacity with additional commodity servers.

In addition to these servers, data center services also rely on specialized hardware LBSs (current software load balancers/reverse proxies do not meet the performance/scale requirements for large services). These switches serve as the access point for services at a particular data center, abstracting back-end service topology and server membership and enabling incremental scalability and fault tolerance of server pools. When a client initiates a request, rather than returning the IP address of an application server, the service instead returns the IP address of a LBS. Client packets must then pass through the LBS on the way to their ultimate destination.

LBSs may operate at the packet-header or packet-payload granularity. In the first case, they forward packets to back-end servers by transparently rewriting IP destination addresses in packet headers. Care must be taken to ensure that all packets belonging to the same flow are mapped to the same server. Many commodity switches provide basic hardware support for appropriate flow hashing to deliver such functionality. A straightforward design might employ OpenFlow [12] coupled with commercially available switches to map flows to back-end servers by monitoring group membership and load information. Hence, we focus on the more challenging (and commercially relevant) instance where LBSs must operate at the granularity of packet-payloads, performing arbitrary processing on application-level messages before forwarding them to appropriate back-end servers.

Performance optimizations, used either independently or in addition to protocol acceleration, are another example of LBS functionality. For example, an LBS will maintain persistent TCP connections to back-end web servers, re-writing HTTP 1.0 requests as 1.1 if necessary to use one of the existing connections.
Switches may also implement connection collapsing, in which many incoming client TCP connections are multiplexed onto a small number of persistent TCP connections at the application servers. These functions provide several benefits. First, they eliminate the overhead of establishing new TCP connections and the delay for it to ramp up to the available bandwidth, reducing overall latency. Second, servers incur some degree of per-connection processing overhead, so if every client connection requires a connection from the middlebox to the back-end, then both the middleboxes and backends incur this cost. By “absorbing” the client connections through connection collapsing, the middlebox reduces back-end load and also reduces its own load by having fewer back-end connections to manage. Connection collapsing and request rewriting for persistent connections is one of the most popular uses for commercial LBSs [9, 87].

While some commercial LBSs can perform byte stream processing, this capability is typically limited to a small set of protocols (e.g., HTTP) with restricted programming models. Further, running at the necessary speeds often requires specialized hardware employing custom ASICs. Finally, it is typically impossible for LBSs to perform message forwarding or rewriting based on state maintained across connections or to initiate RPCs to look up non-local state for dynamic request routing, requiring large-scale service providers to employ proprietary software solutions. The goal of our work is to address these challenges with a scalable, easy-to-program architecture built entirely from commodity server components.

Additionally, some services require finer control over which servers receive which requests. Specifically, these services must direct the request to a particular server based on a unique identifier within the request. Using hardware LBSs to implement this functionality may be impractical, because the logic to determine which server should receive the request may depend on a considerable amount of state. Thus, these services must implement this switching algorithm in software, either as a separate specialized service, or as part of the application service logic that can handle forwarding requests to the correct machine. This functionality logically belongs in the load balancing switch, but placing it there is impractical because the required hardware is too expensive or programming the application
logic is too complex.

### 7.1.2 Architecture

Figure 7.1 shows the xOMB architecture. The major components include commodity hardware switches, our front-end software middleboxes, back-end application servers, and a controller for coordination. The software middleboxes communicate with each other, with the controller, and with *agent processes*—used for collecting statistics such as machine load on each back-end server—using an RPC framework.

**Hardware switches.** While our software middleboxes can perform application-level packet inspection to aid forwarding decisions, we can optionally leverage existing commodity hardware switches to act as the single point of contact for client requests. These commodity switches employ line-rate hashing to map flows to an array of our programmable software middleboxes [106].

**Software middleboxes.** Commodity servers with software middleboxes function as the front-end switches for xOMB. They parse, process, and forward streams of requests and responses between clients and the back-end application servers. xOMB flexibly supports arbitrary protocol and application logic through user-defined processing modules. Deployments can scale processing capacity by “stacking” xOMB servers either vertically (every server runs the same modules) or horizontally (servers run different modules and form a processing chain). Additionally, the middleboxes provide distributed failure detection and load monitoring.

**Application servers.** Application servers (e.g., webservers) form the back end of
our system and process and respond to client requests according to the protocol(s) they are serving. These servers may be grouped into logical pools with related resources or functionality.

**Controller.** The controller provides a central rendezvous point for managing front-end and back-end configuration and membership. In addition to coordinating and storing server membership, the controller also stores a limited amount of service hard state. The controller may be implemented as a replicated state machine for fault tolerance.

### 7.2 Design

The middleboxes form the core of the xOMB framework, with the primary goals of flexibility, programmability, and performance. They provide complete control over data processing, allowing them to work with any protocol, including proprietary, institution-specific ones. Our high-level approach to arbitrary byte stream processing is to terminate client TCP connections at the middlebox, execute the appropriate *modular processing pipeline* (pipeline) containing user-defined processing logic on an incoming byte stream, and then transmit the resulting byte stream emanating from the pipeline over a new TCP connection to the appropriate back-end server. We leave extensions to message-oriented protocols such as UDP to future work. A separate control plane configures the middleboxes, manages membership, performs monitoring, and schedules and executes timers.

xOMB automatically handles low-level functionality necessary for high-performance processing, allowing the programmer to concentrate on the application logic. xOMB abstracts connection management, socket I/O, and data buffering. Additionally, xOMB targets *request-oriented* protocols, which comprise most of today’s Internet services. In these protocols, request *messages* have exactly one logical response message, and responses should be returned in the order requested. xOMB tracks requests and buffers and reorders responses to meet this requirement. Although our design allows arbitrary data processing, our aim is to make handling common processing patterns and standard protocols as simple as possible.
7.2.1 Pipelines and Modules

xOMB divides data stream processing into three logical stages: protocol parsing, message filtering/transformation, and forwarding. Each stage is composed of an arbitrary collection of modules that dynamically determine the processing path of a message. As each module can process different messages independently and concurrently, we refer to the complete DAG of modules and any additional control-flow logic as the pipeline. Because requests and responses require different handling, middleboxes use separate pipelines for each direction.

Each pipeline module represents a single processing task, and modules may be composed of other modules. Modules may be either synchronous or asynchronous; asynchronous modules allow processing tasks to retrieve state over the network without blocking. A simple API consists of methods for initializing state, receiving failure/membership notifications, and a process() method to execute the module’s task. Middlebox programmers must only define a set of modules and the pipeline to link them together.

When a middlebox reads a chunk of data from a socket into a message buffer, it passes the buffer to the pipeline, which successively invokes module process() methods until either a module halts processing or every module has been executed. Modules may store soft state in memory, schedule timers, and, for asynchronous modules, make RPCs to retrieve or store shared persistent state. Modules pass processing results to other modules, or between requests and responses, through arbitrary metadata pointers associated with each message. For example, a parser module may set a data structure representing the parsed message as a metadata value.

xOMB assigns every connection its own pipeline object to increase throughput and simplify programming. The xOMB concurrency model uses strands [5] based on each connection, meaning that at most one thread will execute pipeline or I/O instructions for a given connection at a time. The advantage of this model over explicit locking is that it elegantly allows multiple cores to execute processing or I/O for different connections in parallel. Moreover, if one connection’s pipeline makes an RPC (which is asynchronous), control will immediately trans-
fer to another connection’s pipeline until receiving the reply, further increasing throughput and utilization. Because many modules need to track per-message or per-connection state (e.g., the number of bytes parsed), per-connection pipelines have the additional advantage of simplifying programming by automatically giving each module private state. xOMB also provides support for modules storing state across connections (cf. §7.2.1).

Example Pipeline: HTTP

Figure 7.2 shows an example request and response pipeline for processing HTTP traffic, broken into parse, filter, and forward stages.

Protocol Parsing. The protocol parsing stage transforms the raw byte stream into discrete application-defined messages and sets application-specific metadata in the message structure. On a parse error, the module sets a message flag to close the connection. The parser indicates when it has parsed a complete message along with the total bytes parsed. The middlebox reads these fields to create a new buffer pointing to any remaining bytes for the next message.
Parsers can be chained together to simplify the logic for different subsets of a complicated protocol. For example, parsing XML-RPC might use an HTTP parser followed by an XML parser. Our sample HTTP parser sets a metadata structure representing an HTTP request, including the protocol version, request method, path, headers, etc.

**Filtering and Transformation.** Filter modules perform arbitrary transformations on messages. Our example request pipeline has three filters that illustrate common uses. First, an *AttackDetector* checks the request against a set of attack signatures. Our particular attack filter uses string matching expressions loaded from Snort [17] rules. If the message matches a rule, the filter sets the “drop connection” flag.

Next, the pipeline uses a *Cache* module for protocol acceleration. In the request pipeline, the module checks whether a cache entry exists for the path set in the HTTP metadata. If so, the module sets the message buffer pointers to the cached response and sets the message destination to the client. In the response pipeline, the cache module checks the response headers in the HTTP metadata and stores the response if permitted.

The final filtering step (omitted in Figure 7.2) performs HTTP version rewriting to allow middleboxes to maintain persistent connections to the back-end web servers, even when clients do not support them. If the HTTP metadata version is 1.0, the request pipeline module rewrites the headers to version 1.1 and adds the appropriate “Host” field. The response pipeline also uses a version filter that rewrites the response back to HTTP/1.0 for requests that were transformed. Similarly, because cached responses will always be version 1.1, the request pipeline sends these through a response version filter as well.

**Forwarding.** The forwarding stage sets the message destination based on metadata set by the previous stages. Forwarding can be as simple as selecting a back-end server from a pool to as complicated as computing the destination from a dynamically populated forwarding table. Response pipelines do not have forwarding modules; the middlebox automatically sends responses to the requesting client.

Our example request pipeline uses two forwarding modules. The *URLPool*
class RandomForwardModule : public Module {
    public:
    boost::tribool process(MessagePtr m) {
        MembershipSet members =
            Membership::getMembers(m->getPool());
        if (members.empty()) { return false; } 
        MembershipSet::const_iterator i =
            scommon::random(members);
        m->setDest(scommon::Endpoint(i->addr()));
        return true;
    }
};

Figure 7.3:
A forwarding module implementing random forwarding to pools of servers as specified in the message metadata.

module partitions back-end server pools based on the paths from the URL that they may serve. This module periodically reads configuration state from the controller that maps URL path prefixes to server pools. For example, paths beginning with “/image” go to one set of servers and paths beginning with “/video” go to another set. By using the HTTP metadata path, the module sets a metadata field with the pool for the longest prefix match. The LoadBalance module selects a destination server from the designated pool using a specified load balancing policy. We have implemented simple load balancers that use round-robin or random selection.

Figure 7.3 shows a sample module implementing random forwarding. The module sets the message destination to a random server in the specified pool. If no servers for the specified pool are found, the module returns false, which signals the pipeline logic to initiate alternate processing or discard the message.

Module State and Configuration

Integrating programmable middleboxes into complex distributed protocols requires that the middleboxes can access potentially large amounts of service state necessary for making forwarding decisions. We distinguish two kinds of module
state: configuration state and dynamic state. Configuration state specifies parameters such as rates, cache sizes, numbers of connections, etc., and any global state that changes infrequently. Examples of configuration state include the set of fingerprints used by the *AttackDetector* module or the path prefix to server pool mapping used by the *URLPool* module. *xOMB* uses the controller to manage all global configuration state, stored as a map from state name to opaque binary value. This map can be queried through an RPC interface by a middlebox when it starts, allowing modules to retrieve necessary configuration parameters. Optional metadata includes a version number and time duration, which tells modules how long they should use the current value before checking for a new version.

Dynamic state consists of any unique state that a module references or retrieves for each message. Consider an object store directory that maps billions of object ids to back-end servers. Modules typically cannot prefetch and store a complete copy of such forwarding state because the total state is too large, keeping a consistent copy would be too expensive, or both. *xOMB* modules may dynamically construct forwarding tables during message processing and may control the rate at which dynamic state is updated. Modules can retrieve required state on demand by making asynchronous RPCs to application services such as a back-end metadata server. The ability to build dynamic forwarding tables is a significant advantage afforded by the general programmability of *xOMB* middleboxes relative to less flexible callback-based models (cf. §7.2.3).

Modules store global state, including both configuration and dynamic state, in memory that persists across connections. To allow modules to manage memory effectively, the middlebox passes membership and failure notifications to every module so that they may discard unneeded state. Additionally, modules may set timers to perform periodic state maintenance to optimize storage or purge stale state.

### 7.2.2 Control Plane

Although pipelines and modules form the core of *xOMB*, a number of other components complete the system functionality and convenience.
Membership

xOMB middleboxes and back-end servers require the current server membership for various pools. As middleboxes and servers join and leave, updated membership must be disseminated efficiently. In addition to basic pool membership, xOMB must assign both back-end and middlebox servers to be monitored by one or more middleboxes. These assignments should remain balanced as middleboxes and servers are added and removed. Finally, we also require that there be no manual configuration for adding or removing servers—all membership pools and monitoring assignments must be updated automatically.

To achieve these requirements, the xOMB controller manages pool membership and monitors assignments. Although using a centralized controller may not scale to the largest systems, it is a simple solution and should be sufficient for thousands to tens-of-thousands of servers [52]. The controller may be implemented as a replicated state machine for high availability, or replaced with a coordination system such as ZooKeeper [59].

When a new server comes online, its agent process makes a join RPC to the controller, registering itself as either a middlebox or an application server, and specifies the sets of pools to which it should be added and any pools for which it requires membership updates. The controller records this request and informs all servers who have registered interest in membership updates for that pool. When failure detectors inform the controller of a server failure, it similarly informs all registered servers.

The controller keeps assignment data structures for failure detection and load monitoring. Each structure stores a mapping from a middlebox to a set of servers, the reverse mapping for efficiently removing assignments, and a sorted list of assignment sizes. New servers are added to the middlebox with the least number of assigned servers. New middleboxes are reassigned members from the middlebox with the most servers. The data structure maintains the invariant that the smallest assignment is no less than half the largest assignment, which avoids rebalancing servers every time a single server is added or removed.
Monitoring

Services must respond to failures and changing server loads. Front-ends such as xOMB implement this functionality as they direct requests to back-end servers: middleboxes can avoid sending requests to failed machines and shift traffic to less loaded servers. To effectively minimize service time for requests, the middleboxes need the current liveness and load status of all servers, generally including other middleboxes.

Each middlebox collects load information from a set of servers assigned by the controller at a configurable interval. The xOMB agent on each server reports machine-level information, such as load, CPU, network, and memory utilization, but application server agents may report more detailed application information, such as the number of active connections or operations per second.

Failure Detector

xOMB employs an active failure detector to quickly detect unresponsive servers. The controller assigns every middlebox a set of servers to monitor. Middleboxes ping each of their monitored servers at a configurable period. If the middlebox has not received a response within a configurable timeout, it notifies the controller that it suspects the server to have failed. For more reliable failure detection, xOMB supports assigning multiple middlebox failure detectors for every server. A threshold parameter specifies how many middleboxes must report a suspected failure before the controller declares server failure.

7.2.3 Design Discussion

While a general modular/pipeline approach is common in system design [79, 110], it represents a novel architecture for programmable middleboxes, which typically use a layered approach with protocol-specific callbacks [8, 20]. For example, a conventional middlebox may provide callbacks for processing a new TCP flow, part of an HTTP request (such as the URL), or a complete HTTP request.

The primary advantage of callbacks is that, as long as the product supports
your protocol, they make it straightforward to implement simple protocol-specific handling for a particular set of events. Because vendors tailor the set of callbacks to only specific supported protocols, they can provide a high level of integration for switch programmers, abstracting details such as protocol parsing and loading shared libraries—the programmer only needs to provide bodies of the desired event handlers. Additionally, the callbacks take as arguments the relevant fields for the event, eliminating the need for metadata objects attached to messages.

In contrast, xOMB modular pipelines provide four important advantages over callbacks. First, asynchronous modules allow message processing to perform RPCs to retrieve or store state over the network. Current programmable middleboxes have a fundamental limitation that callbacks must run to completion and must not block, thus precluding this critical functionality for implementing dynamic request routing. Second, xOMB pipelines are more flexible because they are not limited to a fixed set of protocols or callbacks. Third, xOMB pipelines elegantly allow modules to pass arbitrary per-message state to other modules through the message metadata, enabling cross-module processing logic. While it may be possible to achieve similar functionality with callbacks, doing so would require setting global variables, a much more complicated and error-prone approach, and may limit processing parallelism if accessing these global variables requires locking. Finally, xOMB pipelines are potentially more efficient, because parsing modules only need to parse the minimal amount of bytes necessary to complete the desired processing. Furthermore, the pipeline can be programmed to immediately begin processing message fragments rather than waiting for the complete message, potentially reducing latency and overhead for large messages that can be processed with streaming logic.

xOMB pipelines can be structured to provide all of the convenience of callbacks while maintaining the above advantages. For example, we envision including parsers for popular protocols as part of the xOMB distribution. Further, pipelines can emulate a set of callbacks by using a series of modules with methods for each callback. These modules can wait for a desired event to occur and invoke the respective handler method with arguments from the message metadata.
xomb::Pipeline* newRequestPipeline() {
    xomb::Pipeline* p = new xomb::Pipeline();
    p->add(new HttpRequestParseModule());
    p->add(new HttpAttackFilterModule());
    p->add(new HttpRequestCacheModule());
    p->add(new HttpRequestVersionFilterModule());
    p->add(new UrlForwardModule());
    p->add(new RoundRobinForwardModule());
    return p;
}

Figure 7.4: The HTTP request pipeline from §7.2.1.

7.3 Implementation

We now discuss xOMB's implementation in more detail.

7.3.1 Pipeline Libraries

To simplify pipeline programming, xOMB separates module and pipeline implementation from the core xOMB C++ implementation. The user provides pipelines written in C or C++ as shared-object libraries linked with the required modules, also written in C/C++. xOMB allows users to define simple pipelines by just adding modules in the desired order to a pipeline module list. Figure 7.4 shows how to write the pipeline described in §7.2.1. More complicated pipelines may be constructed manually with arbitrary logic and module compositions.

7.3.2 Control Module

The middlebox implements its portion of the control plane in a global control module. Upon startup, the control module first initializes the middlebox by creating threads for asynchronous I/O dispatch [3], and starts an RPC server to receive calls from the controller. Next, it dynamically loads specified pipeline shared-object libraries and begins listening for client connections on the data plane. Finally, it joins the controller and retrieves any global configuration parameters.
As described in §7.2.2, the middlebox receives server pool membership and monitoring assignments from the controller. The control module stores the pool assignments in a shared membership module and schedules timers for its monitoring tasks. A shared load monitor module queries servers for load information and stores the results. When the control module receives a failure notification from the controller, it notifies all pipelines so that the modules can update their state and respond appropriately. However, services such as health and load monitoring are optional. While we expect such functionality in many production environments, simpler deployments may not have this requirement.

The initialization module sets up any global data structures the middlebox uses to store application state. The initialization module can also retrieve state from either the middlebox controller or application servers to populate initial forwarding state.

The middlebox software architecture includes a control module that coordinates the processing through the data pipelines. The control module is composed of several submodules that handle initialization, configuration (and reconfiguration), periodic events via timers, and responding to errors.

The configuration module primarily manages group membership information. By communicating with the controller, it can provide lists of application servers, the other middleboxes, as well as any other services that use the middlebox. This module may also implement callbacks that the framework will invoke when memberships change, allowing application-specific logic to update any global data structures stored by the middlebox and to potentially rebalance state with other middleboxes.

The control module allows the implementation of timers to perform periodic tasks. These might include scanning through data structures and discarding old entries, or making RPCs to relevant data, or rebalancing/compacting state for improved efficiency.

Finally, the control module must implement logic to handle network errors. The middlebox must handle socket errors from clients, including gracefully managing any open TCP connections to back-end servers, as well as removing any
state specific to that client. The middlebox might also respond to errors from application-servers or other middleboxes by updating or removing state. The error handling routines may share logic with the membership change handlers, and the framework supports code sharing. However the middlebox may need to take special actions in handling network errors in addition to updating state for the removal of a server.

7.3.3 Data Plane

The data plane listens for client connections on one or more ports, each of which has its own request and response pipeline. Although the xOMB architecture is general to any protocol, we focus on TCP-based protocols in this paper. Upon accepting a client TCP connection, the middlebox creates a client connection object that holds the client socket, the request pipeline, and a data structure to buffer and reorder responses. xOMB then creates a new message structure and buffer and reads data from the socket. We will refer to the state snapshots in Figure 7.5 to illustrate pipeline processing.

Messages and Buffers

The middlebox creates a message data structure for each request and response to buffer message data while processing it. Buffer management—how buffers are allocated, accessed, and copied—is a critical design detail for building a high performance middlebox. xOMB uses memory efficiently by avoiding user-level copying and freeing buffered data once it has been sent, even if the message is not complete. We use reference-counted buffer pointers to simplify memory management for data that persists across multiple messages.

Each message structure maintains a list of pointers to fixed-size buffers. Because buffers may not be full or may contain data for multiple messages, the message also has a list of segments—contiguous substrings in a buffer—each holding a pointer to the start of the segment (an offset into the buffer) and the segment length. The structure also holds fields for the total buffered data size and the number of bytes parsed, queued, and sent. The segment list and byte counts represent
shifting windows of data to be processed and sent.

To maximize memory efficiency, the middlebox always fills every buffer by using scatter/gather I/O (reading into multiple buffers with one system call) and passing allocated but unused buffers to the next message. When a read completes, the middlebox adds segments to the message pointing to the newly read bytes. Snapshot A of Figure 7.5 shows message M1 after reading 28K into buffer B0.
Message processing proceeds one segment at a time. The pipeline returns one of three results: either the message is complete, an error occurred, or the pipeline needs more data. If the message is complete, the middlebox constructs a new message with any remaining unparsed buffer pointers and segments. On an error, the middlebox discards the message and closes the connection. If the pipeline returns incomplete, the middlebox performs another read.

Snapshot $B$ of Figure 7.5 shows $M_1$ after the middlebox has processed $B_0$. The HTTP parser has set metadata representing the request, including the total request size of 42K, $URLPool$ has set the destination pool, and $LoadBalance$ has set the destination server. Because $M_1$ was not complete, the pipeline returned that it needed more, and the middlebox read another 30K into buffer $B_1$.

Once the pipeline has determined the destination for the message, the middlebox sends any processed segments to the destination. As the middlebox reads and processes new buffers and segments, it simultaneously sends previous ones. By discarding buffers after sending them, the middlebox uses only a small amount of memory for each message, regardless of the total message size. The middlebox limits the amount of data buffered for any message by not reading on a connection while the total buffered size exceeds a threshold; the middlebox eventually closes connections for reads that take too long. Because message data typically will not fall on buffer boundaries, when the pipeline has processed a complete message, xOMB copies pointers for any buffered but unprocessed bytes from the completed message into a new message and invokes the pipeline with the new message before attempting another read.

Figure 7.5 $C$ shows the state after processing $B_1$ through the end of $M_1$. While the pipeline processed $B_1$, the middlebox concurrently sent $B_0$ and then freed it. Because $M_1$ is complete but $B_1$ is not empty, the middlebox creates a new message $M_2$ with initial buffers and segments as shown and empty metadata (not shown). The middlebox will process $M_2$ before attempting to read more from the socket because $M_2$ may be complete and, if so, processed before blocking on further data.
Connection Pool

xOMB maintains a pool of connections to back-end servers, with a configurable number of reusable connections per server. The connection collapsing performed by the middleboxes can significantly increase back-end server efficiency by multiplexing a large number of client connections onto a small number of server connections (§7.5.2). Because the middlebox may interleave client requests on server connections, it automatically demultiplexes and reorders the responses to the appropriate clients.

7.4 Example: Distributed Object Store

For a detailed example of dynamic request routing, we describe an object store service, xOS, based on Amazon’s S3 service [1]. To be interface compatible with S3, we used the unmodified Eucalyptus [82] Walrus storage components for our back-end application servers. However, as of the latest version (2.0), Eucalyptus does not support more than a single Walrus storage server. By using xOMB middleboxes together with a distributed metadata service, we transparently overcome this limitation while maintaining a unified, scalable storage namespace. We quantify xOS scalability in §7.5.3.

xOS hosts objects stored in buckets named by unique keys. Eucalyptus uses a single cloud controller to manage the authentication and metadata for all storage requests. Multiple storage servers will work independently when configured with the same Eucalyptus cloud controller, so we built a xOMB pipeline to consistently forward requests for a given user/bucket to the same storage server.

We implement bucket → server placement with a distributed metadata service that maps \( \langle \text{userid}, \text{bucket} \rangle \) pairs to storage servers. Metadata servers subscribe to the storage server pool on the xOMB controller. Each metadata server is configured with a portion of a 160-bit key space, which it registers with the xOMB controller. The xOS forwarding module retrieves the key-space to metadata server mapping as part of its configuration state.

The xOS request pipeline consists of the standard HTTP parse module
followed by our xOS forwarding module. To process a request, the xOS forwarding module takes the SHA1 hash of the \langle \text{userid}, \text{bucket} \rangle string parsed from the HTTP headers and URL. Using this hash, the module computes the metadata server responsible for that portion of the key-space and makes an RPC to retrieve the storage server. When a metadata server receives a lookup request, it either returns an existing assignment if found or otherwise chooses a new storage server and stores the assignment. The forwarding module caches these assignments to avoid subsequent lookups for the same bucket.

Basing xOS on distributed Walrus servers presents an additional challenge for the front-end middleboxes. The S3 interface contains a ListBuckets request to list all of a user’s buckets. However, no single server contains this information. To support the complete interface, the xOS forwarding module recognizes the ListBuckets message and makes an RPC to each metadata server requesting all the user’s buckets. Once the forwarding module has received all responses, it generates the HTTP and XML response by combining the separate bucket lists. Although not difficult to implement, we note that such functionality would be impossible in designs limited to event-handler callbacks.

In this example design, middlebox servers only maintain soft state. This has the advantage of making middlebox recovery very simple, in that a new middlebox can begin operating immediately without needing to copy persistent state. Each middlebox maintains the following soft state. When the middlebox boots, it contacts to the controller to receive the table containing the assignment of id space partitions to middleboxes, the table containing the assignment of id space partitions to metadata servers, and the set of servers that it should monitor. As the middlebox begins receiving requests, it initially must contact the appropriate metadata server to obtain the object store servers for the id. The middlebox then caches those results, so subsequent requests for that id can be forwarded directly. The middlebox also pushes monitoring status updates to the metadata servers, so that when a new object is created, the metadata server can choose appropriate object store servers based on current load and capacity.

Failure recovery works as follows. When a middlebox fails, the controller,
which receives liveness updates for all middleboxes, reassigns that middleboxes id space to other middleboxes and broadcasts the new mapping to all the middleboxes. When a metadata server fails, its id space must also be assigned to other servers to regain the same degree of fault tolerance. The controller also performs this remapping and updates the middleboxes with a broadcast. When an object store server fails, the metadata servers update the replica set for that object by assigning other object store servers for each of the objects hosted on the failed server. Middleboxes can update their mapping from ids to object store servers lazily, in that if they attempt to forward a request to a failed server and receive a socket error, then they will contact the metadata servers to retrieve an updated replica set.

7.5 Evaluation

In this section, we evaluate the main xOMB design goals of scalability, performance, and programmability. For these experiments, we use servers with Intel 2.13 GHz Xeon quad-core processors and 4 GB DRAM running Linux 2.6.26. All machines have 1 Gbps NICs connected to the same 1Gpbs Ethernet switch.

7.5.1 Programmability

First, we give a sense of xOMB’s programmability. Table 7.1 shows the number of lines of C/C++ code for modules we implemented. The majority of the modules are short, although the HTTP parsing module includes 1700 lines of an HTTP parsing library [15], and the NFS parse and Cache modules both use code generated from the XDR protocol file. In addition, all of the pipelines use the default pipeline construction process, meaning they are only a handful of lines each.
Table 7.1: Module Code Lengths

<table>
<thead>
<tr>
<th>Module</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Parse</td>
<td>111 (+ 1699)</td>
</tr>
<tr>
<td>Round-Robin Forward</td>
<td>73</td>
</tr>
<tr>
<td>Random Forward</td>
<td>37</td>
</tr>
<tr>
<td>URL Pool Forward</td>
<td>31</td>
</tr>
<tr>
<td>HTTP Attack Filter</td>
<td>99</td>
</tr>
<tr>
<td>HTTP Version Filter</td>
<td>72</td>
</tr>
<tr>
<td>xOS Forward</td>
<td>277</td>
</tr>
<tr>
<td>NFS Parse</td>
<td>235 (+ 2906)</td>
</tr>
<tr>
<td>NFS Cache</td>
<td>413 (+ 602)</td>
</tr>
</tbody>
</table>

Figure 7.6: xOMB HTTP throughput

7.5.2 HTTP Performance

We first evaluate xOMB performance with HTTP pipelines. We used Apache 2.2.9 [2] running the MPM worker module, serving files of various sizes. All throughput measurements include the bytes transferred for HTTP headers. In addition, our HTTP client is pipelined, but limits itself to 10 outstanding re-
quests. In these experiments, xOMB uses a basic HTTP pipeline configured to parse requests and forward them, using a “client sticky” round-robin forwarding module, across the available web servers. Our forwarding module ensures that all of a client’s requests go to the same web server, although each client is assigned to a web server round-robin. In addition, we perform connection collapsing down to at most five connections from each middlebox to each webserver.
We compare our performance against Apache directly, the popular open-source reverse proxy nginx [11], and a programmable hardware switch from F5 Networks [8]. The F5 BIG-IP Local Traffic Manager 1600 (LTM) we used has an Intel 1.8 GHz E2160 dual-core processor and 4 GB DRAM and is running OS Version 9.4.8 and has a single 1 Gb/s NIC. We refer to the LTM simply as “F5” in our experiments. It is difficult to compare the processor employed in the F5 relative to our servers. While our machines are three years old and based on older microarchitecture, they do have four cores and a higher clock speed. One challenge with specialized hardware such as the F5 switch is integrating the latest processors and motherboards into a specialized hardware and software environment, a downside endemic to non-commodity solutions.

Figure 7.6 shows a comparison of client throughput to Apache, through a single xOMB middlebox, a single nginx reverse proxy, and through the F5 switch. For each file size, we used 100 clients, which was enough to maximize throughput. Compared to Apache, xOMB does quite well, only losing out slightly at large file sizes due to the fact that the xOMB middlebox must allocate some of its 1 Gb/s of NIC bandwidth to forwarding the clients’ requests to the back-end webserver. Nginx performs noticeably slower in almost all cases, only able to surpass xOMB slightly with 1 MB files. Finally, xOMB performs similarly to the F5 switch for larger files, but it is outperformed handily with smaller files. Unfortunately, we have been unable to determine how the F5 is able to outperform direct Apache connections by such a large margin, even though we have disabled caching.

Although the presence of a single xOMB middlebox shows reduced performance compared to the F5 switch for small files, one of the key components of our design is the ability to scale well. Figure 7.7 shows client throughput when requesting 1 KB files with eight back-end web servers behind differing numbers of middleboxes. A single middlebox is not able to handle the extra capacity additional back-ends provide, but we see near-perfect scalability as we add middleboxes.

In our next experiment, clients request a simple CGI application that computes the SHA-1 hash of a 1 MB file on disk. Instead of being limited by middlebox processing capacity, we are now limited by back-end capacity. Figure 7.8 shows...
client throughput, now measured in requests per second, of 100 clients making requests to the CGI program with varying numbers of web servers behind a single xOMB middlebox. We see that xOMB, nginx, and the F5 switch are all able to achieve near-perfect scalability as back-end resources are added.

One of the most difficult aspects of middlebox design is performance under extremely high numbers of concurrent connections. We ran between 1 and 10,000 clients requesting 1 KB files against xOMB, nginx, the F5, and Apache, with the results shown in Figure 7.9. We first note that Apache by itself does not perform well with 1,000 clients, and we could not get it to serve 10,000 concurrent clients at all. Both xOMB and nginx show similar curves, although we outperform nginx for all connection sizes. However, the F5 shows very unusual behavior. We see it is able to perform extremely well with between 100 and 1000 clients. Additionally, it appears to outperform xOMB and nginx, although we saw around 6,000 of the 10,000 clients' connections closed prematurely by the F5. We were unable to determine the cause of this performance anomaly, so it is unclear who the performance leader is for 10,000 clients.

Our final benchmark for HTTP traffic is a pipeline for filtering potential malicious requests (§7.2.1). We compare against the F5, which we programmed
to perform the same checks. Although F5 offers a firewall module that can filter attack traffic, we chose to implement an attack filter for the F5 in their provided scripting language to both gain experience programming the F5 and to compare the exact same set of rules for both systems.

Our attack filter module loads 285 Snort [17] rules from the controller, each
containing a regular expression to search for in the URL of a web request. We check each URL request against all 285 rules. Although we did not introduce any malicious requests in this experiment, we measure the performance hit of checking every request against all rules. Figure 7.10 compares xOMB’s performance against the F5 running this attack filter against a single webserver. Not only does xOMB outperform the F5 across all tested numbers of clients with 1 MB and 4 KB files, but it sustains the throughput achieved when not running the attack filter with 1 MB files and comes within 10% for 4 KB files.

7.5.3 xOS Performance

Next, we evaluate the performance of xOS with dynamic request routing as described in §7.4. We ran two experiments, one using a single instance of Walrus by itself, and the other using xOS. For both experiments, we used varying numbers of clients to repeatedly write 4 KB blocks to the storage system. We used the Amazon S3 curl client to make the write requests, which do not support pipelining. We had each client create and write to its own unique bucket, a workload that causes the middlebox to do the most dynamic request routing. For the xOS experiment, we used two metadata nodes, eight Walrus nodes, and a single xOMB middlebox.

The results of both experiments are shown in Figure 7.11. We see the Walrus write throughput max out at around 74 operations per second. Throughput decreases as the number of clients increases past 30, and tests with more than 90 clients did not finish correctly as we began having connection issues with Walrus. In contrast, xOMB allows xOS clients to write to the eight Walrus backends in parallel, easily scaling past the capacity of a single Walrus machine.

7.5.4 NFS Acceleration

To evaluate xOMB in a different context than HTTP load balancing, we implemented an NFS protocol accelerator pipeline. Our basic accelerator, designed to speed up wide-area access to an NFS server, caches file attribute structures from lookups and file data from reads or writes. We did not attempt to write a
comprehensive accelerator. Rather, our intent is to demonstrate that xOMB can effectively process diverse protocols with varying goals. A limitation of our implementation is the assumption that all client requests pass through the middlebox; it does not attempt to reconcile the cache with the server if some clients connect to the server directly. Some protocol accelerators increase write throughput by responding to the client immediately before forwarding the write to the server; we opted not to implement this.

The file attribute (getattr) and lookup calls form a significant portion of NFS traffic [97]. Typically, clients cache these attributes for a short duration (3 sec). Also, clients need to ensure that file attributes are up to date while serving reads from its local buffer cache. The large number of round trips due to attribute lookups causes significant performance degradation for NFS over a wide-area network. Having NFS clients connect to a xOMB middlebox on the same LAN, which in turn connects to the server, results in better response time for file system operations by caching file attributes and data.

We evaluate performance with the Postmark benchmark [13], modified to bypass the kernel buffer cache. For this experiment, we compare the performance of direct NFS versus xOMB when the clients connect over both a LAN and the an emulated wide-area link. We emulate a 100ms delay using NetEm [58]. To be a worst-case test for NFS, we mount the file system using synchronous writes. Postmark creates 60 files with sizes ranging from 1B to 10KB and performs 300 transactions. Table 7.2 shows the throughput, average operation latency, and total runtime results for the four runs. xOMB adds a small amount of latency on the LAN, as it must process all the NFS traffic and copy data into the cache. However, the xOMB cache significantly improves the performance of file lookups and read operations with a wide-area delay, completing the workload almost twice as quickly.

7.6 Discussion

We are encouraged by our experience with xOMB and its evaluation. However, there are a number of important issues that require work for programmable
Table 7.2: NFS Latency and Throughput Comparison

<table>
<thead>
<tr>
<th>Operations</th>
<th>LAN</th>
<th>100ms RTT WAN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>xOMB</td>
<td>NFS</td>
</tr>
<tr>
<td>Read (KB/s)</td>
<td>48</td>
<td>50</td>
</tr>
<tr>
<td>Write (KB/s)</td>
<td>55</td>
<td>57</td>
</tr>
<tr>
<td>Create (ms)</td>
<td>1.9</td>
<td>0.3</td>
</tr>
<tr>
<td>Open (ms)</td>
<td>0.6</td>
<td>0.2</td>
</tr>
<tr>
<td>Remove (ms)</td>
<td>0.9</td>
<td>0.2</td>
</tr>
<tr>
<td>Read (ms)</td>
<td>0.6</td>
<td>0.2</td>
</tr>
<tr>
<td>Write (ms)</td>
<td>7.6</td>
<td>7.8</td>
</tr>
<tr>
<td>Total time (s)</td>
<td><strong>23</strong></td>
<td><strong>22</strong></td>
</tr>
</tbody>
</table>

middle boxes to be successful in general. We discuss these in turn below.

**Performance.** Our focus has been on scalability and extensibility. While our single node is reasonable (and in many cases superior to commercial product offerings), it will fundamentally be limited by our user-level implementation. One could imagine alternate, higher-performance xOMB implementations in kernel or even in programmable line cards. While reasonable for certain scenarios, we believe that such architectures will fundamentally limit the expressibility of the available programming model.

**Load Balancing.** Devising good load balancing algorithms is a difficult research and engineering challenge by itself. For evaluation, we implemented simple algorithms with no claims of novelty. The goal of this work is not to innovate in developing better algorithms directly, but rather to provide a framework where it becomes easier to innovate in load balancing algorithms.

**Debugging.** All of our data processing modules have been of moderate complexity thus far, but debugging the behavior of a middle box is a challenge in
general. xOMB facilitates pipeline debugging by allowing the programmer to enable progressively more verbose switch logging. Because xOMB is written in C++, programmers can leverage standard logging techniques and tools such as gdb to assist with debugging. More specialized middleboxes typically do not have the same rich set of open source tools or perhaps even the ability to log state over the network or to local disk.

**Resource Allocation and Isolation.** xOMB currently provides no support to isolate individual processing pipelines from one another or to isolate the processing of one flow from another. For example, a rarely-exercised code path could cause the entire pipeline to fail or to slow processing for concurrent flows. Similarly, an administrator may wish to allocate varying amounts of bandwidth or CPU resources to different flows or pipelines. A range of possible techniques are possible for delivering the necessary isolation, from heavyweight solutions employing entire virtual machines on a per-pipeline basis, to individual processes on a per-flow basis, to in-kernel queueing disciplines limiting the bandwidth available to any individual flow. We plan to explore these and other techniques as part of our ongoing work.

**Fault Tolerance.** Our architecture supports the detection and recovery from individual middlebox failure. However, all flows that were being processed through the failed middlebox would also fail, requiring a retry on the client side. While this represents the current state of the art even for commercial products, it would be interesting to determine whether it would be possible to replicate the necessary TCP and application state information to enable handover of TCP connections from one middlebox to another within the TCP timeout window. This would represent a significant design and engineering challenge, but would perhaps be a necessity for deployments with the most stringent availability requirements.

### 7.7 Summary

Middleboxes have played a controversial role in the evolution of the Internet architecture, shifting from discouraged devices to something of a "necessary evil."
Middleboxes have seen widespread adoption because they provide useful and often critical functionality, improving network performance, security, and manageability in many cases. However, existing middleboxes are based around specialized hardware and software, often focus on a small set of protocols, and provide only limited extensibility.

xOMB demonstrates a new design and architecture for building scalable, extensible middleboxes. We show that programmability need not come at the expense of performance; for instance, the xOMB implementation of a load balancing switch achieves performance comparable to a commercial load balancing switch. Beyond load balancing, we have shown how extensible middleboxes can be used to build scalable services by constructing dynamic forwarding tables based on application service state and the effectiveness of a xOMB protocol accelerator for NFS.

Prior chapters have described how middleboxes performing message routing services based on contents of the messages form a critical component in cloud service architectures. These software message routers enable demand dynamic replica binding in ZANTE (and MAPSTORE) and perform the message inspection and caching necessary to implement local recovery in CORFU.

7.8 Acknowledgment

Chapter 7, in part, is currently being prepared for submission for publication of the material. Anderson, James; Braud, Ryan; Kapoor, Rishi; Vahdat, Amin. The dissertation author was the primary investigator and author of this material.
Chapter 8

Conclusions

Building correct, high performance, and reliable distributed applications remains a significant challenge. Despite this, more and more, computation for everyday applications is being performed by distributed systems, often by servers running in data centers remote from the users. As wireless networks and mobile devices with limited computation, storage, or power resources become ubiquitous, the trend to execute complex application logic and store important and large amounts of application data "in the cloud" will continue to accelerate.

However, delivering these applications and data to a global set of users while meeting scalability, performance, availability, and efficiency requirements poses significant challenges. The cloud computing model is attractive because it promises access to vast computing resources from anywhere with network connectivity, with the goal of simplifying distributed application development by providing abstractions to make applications highly available, scalable, and deliver good performance.

This work has presented techniques and systems that attempt to come closer to fulfilling the promise of cloud computing as a means of simplifying distributed applications. We have shown how scalable replicated state machines can provide fine-grained strong or weak consistency as the basis for wide-area distributed applications. We have also presented a protocol and technique for increasing the availability of these systems by greatly reducing the time to recover from failures. Finally, we have described a framework for building the programmable and scalable software message routing services required by these systems.
8.1 Contributions

This dissertation makes the following contributions:

- First, this dissertation argues that certain classes of distributed applications require strong consistency, and many applications would benefit from the simplified semantics it affords. This dissertation argues that consistency needs to be an explicit option exposed by services, allowing application developers to choose the consistency model that best meets their needs, and shows how this can be achieved with an implementation.

- Second, this dissertation argues that the costs of using strong consistency in a wide-area system are acceptable in modern deployments, and further describes a full implementation of PAXOS and includes a detailed performance study and evaluation of these costs.

- Third, this dissertation argues that fine-grained replicated state machines can be used to provide strongly consistent wide-area services. To support the scale at which cloud services operate, this dissertation introduces a technique for building scalable, efficient, and high performance replicated state machines using dynamic demand binding. This dissertation presents the design and implementation of this system with ZANTE and demonstrates the power of this approach with the storage system MAPSTORE and several example distributed applications based on deployed cloud services.

- Fourth, this dissertation argues that with strongly consistent wide-area replicated services, the limiting factor for availability is the time to recover a replica after a failure. This dissertation describes a technique called local recovery that maintains strong consistency while greatly reducing recovery time for wide-area services and implements local recovery in the CORFU protocol.

- Fifth, this dissertation argues that the message routing layer cloud services require can be more easily and cost-effectively implemented using scalable,
programmable software middleboxes running on commodity servers. This
dissertation describes XOMB, an architecture and implementation for build-
ing these programmable and extensible middleboxes.

8.2 Summary

As more applications and their associated data migrate to cloud computing
based services, the stakes for downtime, disrupted service, or slow performance
have never been higher. Because the cloud offers many cost savings and conve-
nience benefits, entirely new classes of distributed applications for domains that
traditionally avoided distributed deployments will seek to leverage these benefits.
To meet user expectations and business requirements, and to enable these new
applications, cloud providers must not only replicate their services across multiple
data centers, but they must offer the stronger consistency models demanded by
these applications. The systems presented in this dissertation demonstrate the
feasibility of strongly consistent cross data center replication and can serve as an
architecture and model for future systems.
Bibliography


[29] Braden, R. Requirements for internet hosts - communication layers.


[46] Dean, J. Designs, lessons and advice from building large distributed systems, 2009. LADIS.


[67] Lamport, L. Paxos Made Simple.


[76] MILLER, R. Cable Cut Cited in Silicon Valley Outage.  

[77] MILLER, R. Car Crash Triggers Amazon Power Outage.  
http://www.datacenterknowledge.com/archives/2010/05/13/
car-crash-triggers-amazon-power-outage/.

[78] MILLER, R. Explosion at The Planet Causes Major Outage.  
http://www.datacenterknowledge.com/archives/2008/06/01/explosion-
at-the-planet-causes-major-outage/.

[79] MORRIS, R., KOHLER, E., JANNOTTI, J., AND KAASHOEK, M. F. The  
Click modular router. In SOSP (1999).

[80] MUTHTITACHAROEN, A., CHEN, B., AND MAZIERES, D. A low-bandwidth  
network file system.

[81] NARASIMHAN, P., MOSER, L. E., AND MELLIAIR-SMITH, P. M. Eternal  
- a Component-Based Framework for Transparent Fault-Tolerant CORBA.  

[82] NURMI, D., WOLSKI, R., GRZEGORCZYK, C., OBERTELLI, G., SOMAN,  
S., YOUSEFF, L., AND ZAGORODNOV, D. The Eucalyptus Open-Source  

[83] OKI, B. M., AND LISKOV, B. H. Viewstamped Replication: A New Primary  
Copy Method to Support Highly-Available Distributed Systems. In PODC  
(1988).

Pig Latin: A Not-So-Foreign Language for Data Processing. In SIGMOD  
(2008).

[85] OMG. Fault Tolerant CORBA Specification. OMG Document ptc/00-04-04,  

[86] PAESSLER AG. CloudClimate: Amazon S3.  
http://www.cloudclimate.com/s3/.

[87] PAI, V. S., ARON, M., BANGA, G., SVENDSEN, M., DRUSCHEL, P.,  
ZWAENEPOEL, W., AND NAHUM, E. Locality-Aware Request Distribution  


[95] Rowstron, A. I. T., and Druschel, P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems.


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

