KURMA: Geo-Distributed Secure Middleware for Cloud-Backed Network-Attached Storage

A Dissertation Proposal Presented

by

Ming Chen

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

Technical Report FSL-15-02

November 2015

Abstract

KURMA: Geo-Distributed Secure Middleware for Cloud-Backed Network-Attached Storage

by

Ming Chen

Doctor of Philosophy Candidate
in
Computer Science

Stony Brook University
2015

Cloud computing is becoming increasingly popular as utility computing is being gradually realized. Still, many organizations cannot enjoy the high accessibility, availability, flexibility, scalability, and cost-effectiveness of cloud systems because of security concerns and legacy infrastructure. A promising solution to this problem is the hybrid cloud model, which fuses public clouds with private clouds and Network-Attached Storage (NAS). Many researchers tried to secure and optimize public clouds, but few studied the unique security and performance problems of such hybrid solutions.

This thesis proposal explores hybrid cloud storage solutions that have the advantages of both public and private clouds. We focus on preserving the strong security and good performance of on-premises storage, while using public clouds for convenience, data availability, and economic data sharing. We propose *Kurma*, an efficient and secure middleware (proxy) system that bridges traditional NAS and cloud storage. Kurma allows legacy NAS-based programs to seamlessly and securely access cloud storage. Kurma optimizes performance by supporting and improving on the latest NFSv4.1 protocol, which contains new performance-enhancing features including compound procedures and delegations. Kurma also caches hot data in order to serve popular I/O requests from the faster, on-premises network.

On-premises Kurma proxies act as sources of trust, and overcome the security concerns caused by the opaque and multi-tenant nature of cloud storage. Kurma protects data from untrusted clouds with end-to-end integrity and confidentiality, and efficiently detects replay attacks while allowing data sharing among geo-distributed proxies. Kurma protects data from compromised clients using anti-virus scanning in addition to NFSv4.1's access controls. Kurma has a modular architecture and is flexible so that security features can be traded-off for performance.

We have thoroughly benchmarked NFSv4.1 and improved its performance by up to 11×. We have designed and implemented an early Kurma prototype with a moderate overhead of 5–66% under several security policies and workloads. We are working on using multiple clouds as backends to tolerate cloud outages, and efficiently prevent replay attacks in geo-distributed settings. We are exploring further performance optimization by taking full advantages of NFSv4.1's compound procedures, which are currently used ineffectively because of POSIX restrictions.

Our thesis is that cloud storage can be made efficient and secure for traditional NAS-based systems utilizing hybrid cloud solutions such as Kurma.

Contents

List of Figures vi

List of Tables viii

Acknowledgments x

1 Introduction 1

2 Benchmarking Network File System 4
2.1 Introduction 4
2.2 Benchmarking Methodology 5
2.2.1 Experimental Setup 5
2.2.2 Benchmarks and Workloads 6
2.3 Data-Intensive Workloads 7
2.3.1 Random Read 7
2.3.2 Sequential Read 9
2.3.3 Random Write 11
2.3.4 Sequential Write 13
2.4 Metadata-Intensive Workloads 13
2.4.1 Read Small Files 13
2.4.2 File Creation 15
2.4.3 Directory Listing 18
2.5 NFSv4 Delegations 19
2.5.1 Granting a Delegation 19
2.5.2 Delegation Performance: Locked Reads 20
2.5.3 Delegation Recall Impact 22
2.6 Macro-Workloads 23
2.6.1 The File Server Workload 23
2.6.2 The Web Server Workload 24
2.6.3 The Mail Server Workload 26
2.7 Related Work 27
2.8 Conclusions 28
2.8.1 Limitations 28

3 SeMiNAS: Single-Cloud Secure Middlewares 30
3.1 Introduction . . . 30
3.2 Background and Motivation . . . 32
3.2.1 A secure middleware for the cloud . . . 32
3.2.2 Security vs. performance . . . 32
3.2.3 File System APIs vs. RESTful APIs . . . 33
3.3 SeMiNAS Design . . . 33
3.3.1 Threat Model . . . 34
3.3.2 Design Goals . . . 34
3.3.3 Architecture . . . 35
3.3.3.1 Mixing Caching and Security Layers . . . 36
3.3.4 Integrity and Confidentiality . . . 36
3.3.4.1 Meta-Data Management . . . 37
3.3.4.2 Key Management . . . 39
3.3.5 Malware Detection . . . 39
3.3.6 Caching . . . 40
3.4 Implementation . . . 41
3.4.1 NFS-Ganesha . . . 41
3.4.2 Authenticated Encryption . . . 41
3.4.3 Malware Detection . . . 42
3.4.4 Caching . . . 42
3.4.5 Lines of Code . . . 42
3.5 Evaluation . . . 43
3.5.1 Experimental Setup . . . 43
3.5.2 Micro-Workloads . . . 44
3.5.2.1 Security and Caching Features . . . 44
3.5.2.2 Integrity Tests . . . 45
3.5.2.3 Encryption Tests . . . 47
3.5.2.4 Anti-virus Tests . . . 48
3.5.3 Macro-Workloads . . . 49
3.6 Related Work . . . 49
3.6.1 Secure Distributed Storage Systems . . . 50
3.6.2 Storage-Based Intrusion Detection Systems . . . 50
3.6.3 Cloud NAS . . . 50
3.6.4 Cloud Storage Gateways . . . 50
3.7 Conclusions . . . 51
3.7.1 Limitations . . . 51

4 Kurma: Multi-Cloud Secure Middlewares 52
4.1 Introduction . . . 52
4.2 Background . . . 53
4.2.1 ZooKeeper: A Distributed Coordination Service . . . 53
4.2.2 Hedwig: A Publish-Subscribe System . . . 54
4.2.3 Thrift: A Cross-Language RPC Framework . . . 55
4.3 Kurma Design . . . 55

4.3.1 Design Goals 55
4.3.2 Architecture 56
4.3.3 Meta-Data Management 57
4.3.4 Security 59
4.3.5 Consistency Model 59
4.3.6 Partition over Multiple NFS Servers 60
4.3.7 Multiple Clouds 61
4.3.8 NFS Transactional Compounds 61
4.3.9 Kurma Operation Examples 62
4.3.9.1 Create a New File 62
4.3.9.2 Open an Existing File 63
4.3.9.3 Write to a File 63
4.3.9.4 Read From a File 63
4.3.9.5 Close a File 63
4.4 Related Work 64
4.4.1 A Cloud-of-Clouds 64
4.4.2 Freshness Guarantees 64
4.4.3 Compound Operations 65

5 Proposed Work **66**
5.1 Kurma Implementation and Evaluation 66
5.2 Development of New Kurma Features 67
5.3 NFS Transactional Compounds 67

6 Conclusions **68**
6.1 Future Work 69

List of Figures

2.1 Random-read throughput with 16 threads and different network delays (varying I/O size). 8

2.2 Random-read throughput with 1MB I/O size, default 2MB TCP maximum buffer size, and different network delays (varying the number of threads per client). 8

2.3 Sequential-read throughputs of individual clients when they were launched one after the other at an interval of one minute. Throughput results of one run of experiments; the higher the better. When all five clients are running (after 4 minitues), the throughputs of the five idential clients are not equal and fall into two clusters, where the throughput of the higher cluster (winners) is about twice of the lower cluster (losers). The winners and losers changed at approximately 4m30s because of re-hashing. The winner-loser pattern is irrelvent to the launch order of the clients; for example, we launched Client2 before Client3, but Client2 is a loser and Client3 is a winner at the end. 9

2.4 Illustration of Hash-Cast. The NIC of the NFS server has six transmit queues (tx). The NFS server is sending data to five clients using one TCP flow for each client. Linux hashes the TCP flows of Client1, Client3, and Client5 into tx3, tx2, and tx5, respectively; and hashes the flows of both Client2 and Client4 into tx0. Therefore, Client2 and Client4 shares one transmit queue and each gets half of the throughput of the queue. 10

2.5 Random-write throughput in a zero-delay network with different I/O size. The higher the better. “Max” is the throughput when running the workload directly on the server side without using NFS. The “Max” throughput serves as a baseline when evaluating NFS’s overhead. 11

2.6 Random-write throughput of a single NFS client in a zero-delay network (log_{10}). The higher the better. The “App” curve is the throughput observed by the benchmarking application, i.e., the writing speed of the application to the file; the “Client” curve is the throughput observed by the NFS client, i.e., the writing speed of the clients to the remote NFS server. 12

2.7 Throughput of reading small files with one thread in a zero-delay network. 13

2.8 Aggregate throughput of reading small files with 16 threads in a 10ms-delay network ($\log_{10}$). The “V4.1” curve is the throughput of vanilla NFSv4.1; the “V4.1p” cruve is the throughput of the patched NFSv4.1 with our fix. 14

2.9 Aggregate throughput of creating empty files in a 10ms-delay network with different numbers of threats per client. 15

2.10 Average number of outstanding requests when creating empty files in a 10ms-delay network. . . . 16
2.11 Rate of creating empty files in a zero-delay network. . . . 17
2.12 Average waiting time for V4.1p's session slots of ten experimental runs. Error bars are standard deviations. . . . 17
2.13 Directory listing throughput ($\log_{10}$). . . . 18
2.14 Running time of the locked-reads experiment ($\log_{10}$). Lower is better. . . . 21
2.15 File Server throughput (varying network delay) 23
2.16 Number of NFS requests made by the File Server 23
2.17 Web Server throughput (varying network delay). . . . 24
2.18 Web Server throughput in the zero-delay network (varying thread count per client). 25
2.19 Mail Server throughput (varying network delay) 26
2.20 Mail Server throughput (varying client count) 26

3.1 SeMiNAS high level architecture 31
3.2 SeMiNAS layered architecture 35
3.3 GCM for integrity and optional encryption 37
3.4 SeMiNAS meta-data management 38
3.5 NFS end-to-end data integrity using DIX 38
3.6 Throughput of different combinations of security and caching features with 4KB I/O size and 30ms network latency 45
3.7 Relative speed of I over P with 4KB I/O size and 1MB file size. 100% (dashed line) is the baseline P. 46
3.8 Relative speed of IC over C with 4KB I/O size and 1MB file size. 100% is the baseline with only caching (C). 46
3.9 Relative speed of ICE over IC with 64KB I/O size and 1MB file size. 100% (dashed line) is the baseline with integrity and caching (IC). Note the Y-axis starts from 80%. 47
3.10 Relative speed of ICEA over ICE with 4KB I/O size under different file sizes, read-write (n:m) ratios, and network latencies. 100% (dashed line) is the baseline ICE. 48
3.11 Filebench results with 30ms network latency 49

4.1 Architecture of Kurma. Kurma consists of geo-distributed proxies in regions where end-users reside. Each region has one single proxy. Although the single proxy may be physically distributed among multiple machines, Kurma coordinates these machines using ZooKeeper to ensure they behave as a single logical proxy. All proxies are inter-connected by a trusted communication link; Kurma uses the links to replicating meta-data among proxies. 56

4.2 Components of a Kurma Proxy. The arrows indicate interaction among the components. A proxy comprises of three services: NFS, PCache, and Kurma FS. The NFS server runs NFS-Ganesha and exports cached and cloud-backed files to clients. The PCache service manages the persistent write-back cache. Kurma FS serves as the NFS server's secure proxy to the cloud back-end, and maintains a global namespace by replicating meta-data across all proxies. Kurma FS depends ZooKeeper, BookKeeper, and Hedwig for meta-data storage, distributed coordination, and meta-data replication. 58
4.3 Sequence Diagram of File Creation. The sequences of other operations follow a similar direction of moving from client to FSAL_PCACHE, FSAL_KURMA, Kurma FS, and then gradually return back to the client. 62

List of Tables

2.1 NFS operations performed by each client for NFSv3 and NFSv4.1 (delegations on and off). Each NFSv4.1 operation represents a compound procedure. For clarity, we omit trivial operations (e.g., PUTFH) in compounds. NFSv3's LOCK and LOCKU come from the Network Lock Manager (NLM). 20

3.1 Lines of code of the SeMiNAS prototype 42
3.2 Combinations of security and caching features. The names of the configurations are combinations of the first letter of enabled features. For example, IC means the configuration when both Integrity and Caching are enabled. 44

Acknowledgments

This work would not be possible without the helpful people around me. I thank my parents for supporting me and urging me to pursue a Ph.D. degree, and my wife Xia and daughter Tianlan for loving and accompanying a busy and poor Ph.D. student. A Ph.D. study is long, challenging, and stressful. But with them, the journey is much more meaningful and joyful.

I am immensely grateful to my advisor Prof. Erez Zadok. I first met Erez when he was giving a guest lecture in my operating system class. I was so amazed by his passion about and deep understanding of Linux file systems that I immediately decided to do research in storage systems although I had no system background at all and was nervous of dealing with complex computer systems. I am lucky and grateful to be accepted as his Ph.D. student. He helps me in every aspect of the Ph.D. study, including research methodology, reading and writing research paper, polishing technical talks, writing beautiful code, fixing nasty kernel bugs, and more. He consistently gave me invaluable suggestions and instructions in every progress of this Ph.D. program. I also appreciate his patience and kindness when I was slow and confused.

I thank other FSL students who also contributed to this work. Soujanya Shankaranarayana helped me a lot with experiments and plotting when benchmarking NFS. Arun Olappamanna Vasudevan and Kelong Wang helped implementing anti-virus and integrity features in SeMiNAS. Garima Gehlot helped evaluating SeMiNAS and is still helping me improving NFSv4.1 support in NFS-Ganesha. Bharat Singh and Ashok Sankar Harihara Subramony have been helping me in implementing transactional NFS compounds. Although not collaborating on this work, I also benefited a lot from discussion with and help of other FSL labmates including Zhen Cao, Deepak Jain, Zhichao Li, Sonam Mandal, Justin Seyster, Vasily Tarasov, and Sun (Jason) Zhen.

I am also grateful to external collaborators on this work including Prof. Geoff Kuenning from Harvey Mudd College and Dr. Dean Hildebrand from IBM Research—Almaden. They are very knowledgeable and helped this work significantly with insightful comments and valuable feedbacks. They also taught me a lot about NFS, cloud computing, and English writing. I appreciate the help of Lakshay Akula and Ksenia Zakirova for their help on this work during the summers.

I thank Prof. Don Porter and Prof. Scott Stoller for serving on the committee of my thesis proposal and providing invaluable advices. I also thank NSF awards CNS-1223239, CNS-1251137, and CNS-1302246 for the financial support of this study.

Chapter 1

Introduction

Cloud storage has many desirable traits including high accessibility (from multiple devices, at multiple locations), availability, flexibility, scalability, and cost-effectiveness [8, 81, 138]. For instance, the availability of Google Cloud Storage in 2014 is higher than five nines (99.999%) [138]. However, cloud storage providers need to improve the integrity and confidentiality of customers' data. For example, some customer data got silently corrupted in the cloud [131]. Silent data corruption could be disastrous, especially in healthcare and financial industries. Privacy and confidentiality are other serious security concerns as increasingly more organizations and people are moving their enterprise and private data to the cloud. The significance is emphasized by high-profile incidents such as leakage of intimate photos of celebrities [7] and theft of patient records [87].

In addition to security concerns, legacy infrastructure is another obstacle to cloud storage adoption. It is difficult to impossible for many organizations to switch to all-cloud infrastructures: traditional NAS-based systems are incompatible with some clouds' eventual-consistency semantics [28] of cloud storage. Moreover, the cost of a full migration of infrastructure can be prohibitive. Higher performance is also a reason to keep legacy on-premises infrastructure because cloud accesses incur round trips in wide-area networks and are thus slow. In contrast, on-premises infrastructure uses local-area networks and is thus much faster.

Hybrid cloud computing is a new computing paradigm that takes advantages of both on-premises infrastructure and public clouds. In the hybrid-cloud model, a portion of computing and storage goes to the cloud (public clouds) for high accessibility, availability, and scalability—while the rest remains on premises (private clouds) for high performance and stronger security. Enjoying the best of both worlds, hybrid clouds are becoming more popular. For instance, many storage appliances and hyper-convergence platforms [89, 90, 101] now have cloud integration so that public clouds can be used for backup, expansion, disaster recovery, and web-tier hosting—whereas other workloads still stay on-premises. However, most existing hybrid cloud systems [89, 90, 101] use public clouds as a separate tier for specific workloads such as backup, web-tier hosting. Hybrid clouds, although enjoying strong security and high performance for generic workloads, have not been studied much.

This thesis proposal focuses on secure, efficient, and generic hybrid-cloud storage solutions that integrate seamlessly with traditional NAS-based solutions. Specifically, we propose *Kurma*, a hybrid-cloud storage system that enables geo-distributed offices of an organization to efficiently and securely store and share data in public clouds. Depending on configuration, Kurma can tolerate failure of a single or more cloud providers by storing data across multiple cloud providers, using

replication or erasure coding. Kurma typically consists of one node for each office, and each Kurma node is an on-premises proxy that sits between local clients and remote clouds. Kurma combines the advantages of both private and public clouds, with an emphasis on strong security and high performance. In Kurma's threat model, public clouds are not trusted, clients are semi-trusted, and only the Kurma proxies are fully trusted; Kurma can provide strong security when public clouds are malicious and clients are compromised by malware. Kurma achieves high performance by reducing high-latency cloud accesses with local caching, and by using optimized NFSv4.1 for communication with clients.

Kurma ensures end-to-end data integrity and confidentiality using authenticated encryption before sending data to the cloud. Data stays in encrypted form in the cloud, and is not decrypted until clients retrieve the data from clouds. End-to-end integrity and confidentiality protect data from not only potential attacks during data transmission over the Internet, but also misbehaving cloud servers and storage devices. Using a simple key-exchange scheme, Kurma can share files securely among multiple geo-distributed Kurma nodes without relying on any trusted third-party. Kurma's cryptographic scheme is robust to replay and swap attacks. Kurma records the version of each data block to detect replay attacks that attempt to overwrite a file with older versions of the file. Kurma synchronizes the version numbers of data blocks among all its nodes so that clients always get the latest version of files after they are modified by a remote Kurma node. Kurma is also secure against intra- and inter-file swap attacks that swap two different data blocks. Kurma also performs anti-virus scanning to catch infected clients and to stop the spread of viruses.

In addition to strong security, Kurma optimizes performance with three mechanisms. First, Kurma minimizes the performance overhead of its security features. Kurma embeds security meta-data (e.g., Message Authentication Codes) into the data so that storing security meta-data does not incur any extra high-latency cloud accesses. Kurma allows clients turn on each security feature (integrity, encryption, and anti-virus) separately according to desired security policies; this facilitates flexible trade-offs between performance and security when, for example, some files need only integrity but not encryption or anti-virus. Second, each Kurma node contains a persistent write-back cache that stores recently-used data and coalesces writes to the cloud. Caching allows Kurma to handle most file operations in the on-premises network, without accessing any remote cloud servers. Kurma carefully integrates caching with the security modules to pursue the right balance between security and performance. For example, Kurma stores plain-text data in the cache so that no encryption or decryption is needed for cached operations, and Kurma also delays anti-virus scanning for as long as it is safe when inserting a dirty file into the cache. Third, Kurma uses an optimized NFSv4.1 implementation for communication with clients. Compared to the still popular NFSv3 [79], the latest NFSv4.1 has new and advanced features such as compound procedures and delegations that can improve performance.

We began this work by benchmarking NFSv4.1 to find out how good or bad NFSv4.1 performs in local- and wide-area networks; along the way, we found and fixed a number of problems, which significantly improved NFSv4.1's performance. We then implemented and evaluated an early Kurma prototype called *SeMiNAS*, which uses a single NFS-based public cloud (instead of multiple clouds) and is not yet secure against replay attacks. Based on the early prototype, we are adding defense to replay attacks when data is shared among geo-distributed Kurma proxies; the main idea to achieve that is to synchronize and check version numbers of data blocks among proxies. We also propose to use multiple public clouds as back-end to avoid data being locked to a single cloud provider. Meanwhile, we are exploring how to further improve NFSv4.1's perfor-

mance by taking full advantages of compound procedures, which we found to be under-utilized due to POSIX limitations [21]. We propose to design and implement a convenient and efficient file system API that facilitates combining multiple small operations into fewer but larger compound procedures.

Our thesis is that cloud storage can be made both efficient and secure for generic workloads, and seamlessly integrate with traditional NAS-based systems. We strive for the following five contributions in the design, implementation, and evaluation of Kurma:

- a comprehensive and in-depth performance analysis of NFSv4.1 and its unique features (statefulness, sessions, delegations, etc.) by comparison to NFSv3 under low- and high-latency networks, using a wide variety of micro- and macro-workloads;
- significant performance improvements of NFSv4.1 by fixing Linux's NFSv4.1 implementation and taking full advantages of NFSv4.1's compound procedures;
- a geo-distributed hybrid cloud system that allows on-premises NFS clients to store and share data in public clouds in a secure, seamless, efficient, and flexible manner;
- a powerful cryptographic data-sharing scheme that is efficient and robust against replay attacks without using traditional Merkle trees [82], which are expensive in cloud environments;
- insights into complex interactions and the trade-off between caching and security features, including integrity, confidentiality, and anti-virus.

The rest of this thesis proposal is organized as follows. Chapter 2 presents the performance benchmarking of NFSv4.1 in comparison to NFSv3, as well as our improvement to Linux's NFSv4.1 implementation that boosts performance by up to $11\times$. Chapter 3 details the design, implementation, and evaluation of SeMiNAS, an early Kurma prototype as our first attempt. SeMiNAS provides integrity, confidentiality, and anti-virus, but uses a single NAS-based cloud as the back-end and is insecure under replay attacks. Chapter 4 describes the design of Kurma that uses multiple clouds and is secure under replay attacks in geo-distributed settings. Chapter 5 proposes the work to be finished in the dissertation including the implementation and evaluation of Kurma, and exploration of NFS compound procedures. Chapter 6 concludes and discusses future work beyond this proposed thesis.

Chapter 2

Benchmarking Network File System

2.1 Introduction

Before the cloud era, over 90% of enterprise storage capacity was served by network-based storage [142], and Network File System (NFS) represents a significant proportion of that total [124]. NFS has become a highly popular network-storage solution since its introduction more than 30 years ago [108]. Faster networks, the proliferation of virtualization, and the rise of cloud computing all contribute to continued increases in NFS deployments [1]. In order to inter-operate with more enterprises, Kurma supports an NFS interface and its proxies appear as NAS appliances to clients. Using NFS, instead of vendor-specific cloud storage APIs, as the storage protocol also improves application portability and alleviates the vendor lock-in problem of cloud storage [8]. In this chapter, we focus our study on NFS. Specifically, we performed a comparative benchmarking study of the NFS versions to choose the NFS version(s) to be supported in Kurma.

Network File System is a distributed file system initially designed by Sun Microsystem [108]. In a traditional NFS environment, a centralized NFS server stores files on its disks and exports those files to clients; NFS clients then access the files on the server using the NFS protocol. Popular operating systems, including Linux, Mac OSX, and Windows, have in-kernel NFS support, which allows clients access remote NFS files using the POSIX API as if they are local files. By consolidating all files in once server, NFS simplifies file sharing and storage management significantly.

The continuous development and evolution of NFS has been critical to its success. The initial version of NFS is known only internally within Sun Microsystems, the first publicized version of NFS is NFSv2 [108,119], which supports only UDP and 32-bit file sizes. Following NFSv2 (which we will refer to as V2 for brevity), NFSv3 (V3) added TCP support, 64-bit file sizes and offsets, asynchronous COMMITs, and performance features such as READDIRPLUS. NFSv4.0 (V4.0), the first minor version of NFSv4 (V4), had many improvements over V3, including (1) easier deployment with one single well-known port (2049) that handles all operations including file locking, quota management, and mounting; (2) stronger security using RPCSEC_GSS [107]; (3) more advanced client-side caching using delegations, which allow the cache to be used without lengthy revalidation; and (4) better operation coalescing via COMPOUND procedures. NFSv4.1 (V4.1), the latest minor version, further adds Exactly Once Semantics (EOS) so that retransmitted non-idempotent operations are handled correctly, and pNFS, which allows direct client access to multi-

ple data servers and thus greatly improves performance and scalability [56, 107]. NFS's evolution does not stop after NFSv4.1; NFSv4.2 is under development with many new features and optimizations [120] already proposed.

V4.1 became ready for production deployment only a couple of years ago [37, 79]. Because it is new and complex, V4.1 is less understood than older versions; we did not find any comprehensive evaluation of either V4.0 or V4.1 in the literature. (V4.0's RFC is 275 pages long, whereas V4.1's RFC is 617 pages long.) However, before adopting V4.1 for production, it is important to understand how NFSv4.1 behaves in realistic environments. To this end, we thoroughly evaluated Linux's V4.1 implementation by comparing it to V3, the still-popular older version [79], in a wide range of environments using representative workloads.

Our NFS benchmarking study has four contributions: V4.1 in low- and high-latency networks, using a wide variety of micro- and macro-workloads; **(1)** performance analysis that clearly explains how underlying system components (networking, RPC, and local file systems) influence NFS's performance; **(2)** a deep analysis of the performance effect of V4.1's unique features (statefulness, sessions, delegations, etc.) in its Linux implementation; and **(3)** fixes to Linux's V4.1 implementation that improve its performance by up to 11×. This benchmarking study has been published in ACM SIGMETRICS 2015 [21].

Some of our key findings are:

- How to tune V4.1 and V3 to reach up to 1177MB/s aggregate throughput in 10GbE networks with 0.2–40ms latency, while ensuring fairness among multiple NFS clients.

- When we increase the number of benchmarking threads to 2560, V4.1 achieves only 0.3× the performance of V3 in a low-latency network, but is 2.9× better with high latency.

- When reading small files, V4.1's delegations can improve performance up to 172× compared to V3, and can send 29× fewer NFS requests in a file-locking workload;

The rest of this chapter is organized as follows. Section 2.2 describes our benchmarking methodology. Sections 2.3 and 2.4 discuss the results of data- and metadata-intensive workloads, respectively. Section 2.5 explores NFSv4's delegations. Section 2.6 examines macro-workloads using Filebench. Section 2.7 overviews related work. We conclude and discuss limitations in Section 2.8.

2.2 Benchmarking Methodology

This section details our benchmarking methodology including experimental setup, software settings, and workloads.

2.2.1 Experimental Setup

We used six identical Dell PowerEdge R710 machines for this study. Each has a six-core Intel Xeon X5650 2.66GHz CPU, 64GB of RAM, and an Intel 82599EB 10GbE NIC. We configured five machines as NFS clients and one as the NFS server. On the server, we installed eight Intel DC S3700 200GB SSDs in a RAID-0 configuration with 64KB stripes, using a Dell PERC 6/i

RAID controller with a 256MB battery-backed write-back cache. We measured read throughputs of up to 860MB/s using this storage configuration. We chose these high speed 10GbE NICs and SSDs to avoid being bottlenecked by the network or the storage. Our initial experiments showed that even a single client could easily overwhelm a 1GbE network; similarly, a server provisioned with HDDs or even RAID-0 across several HDDs quickly became overloaded. We believe that NFS servers' hardware and network must be configured to scale well and that our chosen configuration represents modern servers; it reached 98.7% of the 10GbE NICs' maximum network bandwidth, allowing us to focus on the NFS protocol's performance rather than hardware limits.

All machines ran CentOS 7.0.1406 with a vanilla 3.14.17 Linux kernel. Both the OS and the kernel were the latest stable versions at the time we began this study. We chose CentOS because it is a freely available version of Red Hat Enterprise Linux, which is often used in enterprise environments. We manually ensured that all machines had identical BIOS settings. We connected the six machines using a Dell PowerConnect 8024F 24-port 10GbE switch. We enabled jumbo frames and set the Ethernet MTU to 9000 bytes. We also enabled TCP Segmentation Offload to leverage the offloading feature of our NIC and to reduce CPU overhead. We measured a round-trip time (RTT) of 0.2ms between two machines using `ping` and a raw TCP throughput of 9.88Gb/s using `iperf`.

Many parameters can affect NFS performance, including the file system used on the server, its format and mount options, network parameters, NFS and RPC parameters, export options, and client mount options. Unless noted otherwise, we did not change any default OS parameters. We used the default `ext4` file system, with default settings, for the RAID-0 NFS data volume, and chose Linux's in-kernel NFS server implementation. We did not use our Kurma NFS server to avoid any potential problems in our implementation and to draw conclusions that are reproducible and widely applicable. We exported the volume with default options, ensuring that `sync` was set so that writes were faithfully committed to stable storage as requested by clients. We used the default RPC settings, except that `tcp_slot_table_entries` was set to 128 to ensure the client could send and receive enough data to fill the network. We used 32 NFSD threads, and our testing found that increasing that value had a negligible impact on performance because the CPU and SSDs were rarely the bottleneck. On the clients, we used the default mount options, with the `rsize` and `wsize` set to 1MB, and the `actimeo` (attribute cache timeout) set to 60 seconds. Because our study focuses on the performance of NFS, in our experiments we used the default security settings, which do not use RPCSEC_GSS or Kerberos and thus do not introduce additional overheads.

2.2.2 Benchmarks and Workloads

We developed a benchmarking framework named *Benchmaster*, which can launch workloads on multiple clients concurrently. To verify that Benchmaster can launch time-aligned workloads, we measured the time difference by NTP-synchronizing client clocks and then launching a program that simply writes the current time to a local file. We ran this test 1000 times and found an average delta of 235ms and a maximum of 432ms. This variation is negligible compared to the 5-minute running time of our benchmarks.

Benchmaster also periodically collects system statistics using tools such as `iostat` and `vmstat`, and by reading `procfs` entries such as `/proc/self/mountstats`. The `mountstats` file provides particularly useful details of each individual NFS procedure, including counts of requests,

the number of timeouts, bytes sent and received, accumulated RPC queueing time, and accumulated RPC round-trip time. It also contains RPC transport-level information such as the number of RPC socket sends and receives, the average request count on the wire, etc.

We ran our tests long enough to ensure stable results, usually 5 minutes. We repeated each test at least three times, and computed the 95% confidence interval for the mean using the Student's t-distribution. Unless otherwise noted, we plot the mean of all runs' results, with the half-widths of the confidence intervals shown as error bars. We focused on system throughput and varied the number of threads in our benchmarking programs in our experiments. Changing the thread count allowed us to (1) infer system response time from single-thread results, (2) test system scalability by gradually increasing the number of threads, and (3) measure the maximum system throughput by using many threads.

To evaluate NFS performance over short- and long-distance networks, we injected delays ranging from 1ms to 40ms using `netem` at the NFS clients side. Using `ping`, we measured 40ms to be the average latency of Internet communications within New York State. We measured New York-to-California latencies of about 100ms, but we do not report results using such lengthy delays because many experiments operate on a large number of files and it took too long just to initialize (pre-allocate) those files. For brevity, we refer to the network without extra delay as "zero-delay," and the network with $n$ms injected delay as "$n$ms-delay" in the rest of this thesis proposal.

We benchmarked four kinds of workloads:

1. Data-intensive micro-workloads that test the ability of NFS to maximize network and storage bandwidth (Section 2.3);

2. Metadata-intensive micro-workloads that exercise NFS's handling of file metadata and small messages (Section 2.4);

3. Micro-workloads that evaluate delegations, which are V4's new client-side caching mechanism (Section 2.5); and

4. Complex macro-workloads that represent real-world applications (Section 2.6).

2.3 Data-Intensive Workloads

This section discusses four data-intensive micro-workloads that operate on one large file: random read, sequential read, random write, and sequential write.

2.3.1 Random Read

We begin with a workload where five NFS clients read a 20GB file with a given I/O size at random offsets. We compared the performance of V3 and V4.1 under a wide range of parameter settings including different numbers of benchmarking threads per client (1–16), different I/O sizes (4KB–1MB), and different network delays (0–40ms). We ensured that all experiments started with the same cache states by re-mounting the NFS directory and dropping the OS's page cache before each experiment. For all combinations of thread count, I/O size, and network delay, V4.1 and V3 performed equally well because these workloads were exercising the network and storage bandwidth rather than the differences between the two NFS protocols.

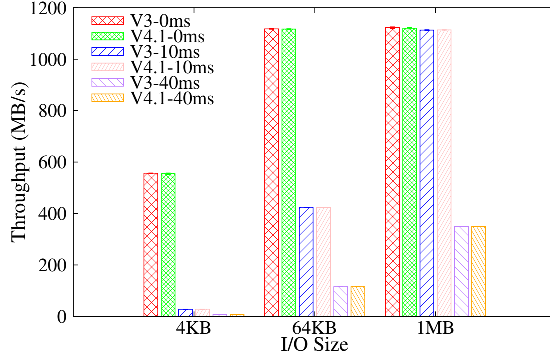


Figure 2.1: Random-read throughput with 16 threads and different network delays (varying I/O size).

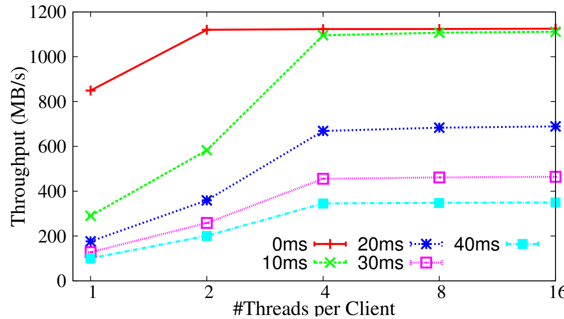


Figure 2.2: Random-read throughput with 1MB I/O size, default 2MB TCP maximum buffer size, and different network delays (varying the number of threads per client).

We found that increasing the number of threads and the I/O size always improved a client's throughput. We also found that network delays had a significant impact on throughput, especially for smaller I/O sizes. As shown in Figure 2.1, a delay of 10ms reduced the throughput by 20× for 4KB I/Os, but by only 2.6× for 64KB ones, and did not make a difference for 1MB I/Os. The throughputs in Figure 2.1 were averaged over the 5-minute experiment run, which can be divided into two phases demarcated by the time when the NFS server finally cached the entire 20GB file. NFS's throughput was bottlenecked by the SSDs in the first phase, and by the network in the second. The large throughput drop for 4KB I/Os (20×) was because the 10ms delay lowered the request rate far enough that the first phase did not finish within 5 minutes. But with larger I/Os, even with 10ms network delay the NFS server was able to cache the entire 20GB during the run. Note that the storage stack performed better with larger I/Os: the throughput of our SSD RAID is 75.5MB/s with 4KB I/Os, but 285MB/s with 64KB I/Os (measured using direct I/O and 16 threads), largely thanks to the SSDs' inherent internal parallelism.

However, when we increased the network delay further, from 10ms to 40ms, we could not saturate the 10GbE network (Figure 2.2) even if we added more threads and used larger I/O sizes. As shown in Figure 2.2, the curves for 20ms, 30ms, and 40ms reached a limit at 4 threads. We found that this limit was caused by the NFS server's maximum TCP buffer sizes (`rmem_max` and `wmem_max` size), which restricted TCP's congestion window (i.e., the amount of data on the wire). To saturate the network, the `rmem_max` and `wmem_max` sizes must be larger than the network's

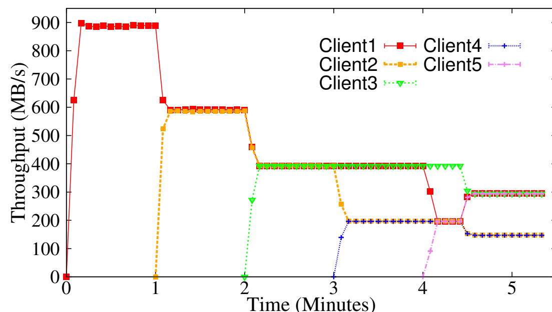


Figure 2.3: Sequential-read throughputs of individual clients when they were launched one after the other at an interval of one minute. Throughput results of one run of experiments; the higher the better. When all five clients are running (after 4 minitues), the throughputs of the five idential clients are not equal and fall into two clusters, where the throughput of the higher cluster (winners) is about twice of the lower cluster (losers). The winners and losers changed at approximately 4m30s because of re-hashing. The winner-loser pattern is irrelvent to the launch order of the clients; for example, we launched Client2 before Client3, but Client2 is a loser and Client3 is a winner at the end.

bandwidth-delay product. After we changed those values from their default of 2MB to 32MB (larger than $\frac{10Gb/s \times 40ms}{5}$ where 5 is the number of clients), we achieved a maximum throughput of 1120MB/s when using 8 or more threads in the 20ms- to 40ms-delay networks. These experiments show that we can come close to the maximum network bandwidth for data-intensive workloads by tuning the TCP buffer size, I/O size, and the number of threads for both V3 and V4.1. To avoid being limited by the maximum TCP buffer size, we used 32MB for `rmem_max` and `wmem_max` for all machines and experiments in the rest of this proposal.

2.3.2 Sequential Read

We next turn to an NFS sequential-read workload, where five NFS clients repeatedly scanned a 20GB file from start to end using an I/O size of 1MB. For this workload, V3 and V4.1 performed equally well: both achieved a maximum aggregate throughput of 1177MB/s. However, we frequently observed a *winner-loser pattern* among the clients, for both V3 and V4.1, exhibiting the following three traits: (1) the clients formed two clusters, one with high throughput (winners), and the other with low throughput (losers); (2) often, the winners' throughput was approximately double that of the losers; and (3) no client was consistently a winner or a loser, and a winner in one experiment might became a loser in another.

The winner-loser pattern was unexpected given that all the five clients had the same hardware, software, and settings, and were performing the same operations. Initially, we suspected that the pattern was caused by the order in which the clients launched the workload. To test that hypothesis, we repeated the experiment but launched the clients in a controlled order, one additional client every minute. However, the results disproved any correlation between experiment launch order and the winners. Figure 2.3 shows that Client2 started second but ended up as a loser, whereas Client5 started last but became a winner. Figure 2.3 also shows that the winners had about twice the throughput of the losers. We repeated this experiment multiple times and found no correlation

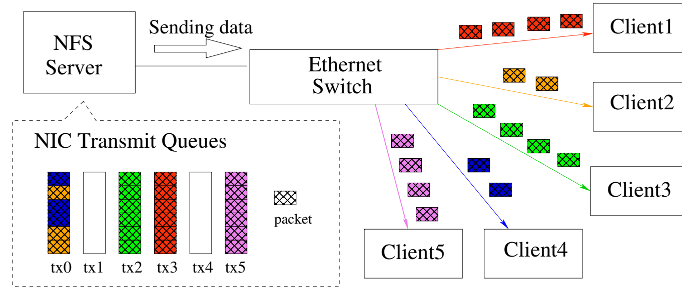


Figure 2.4: Illustration of Hash-Cast. The NIC of the NFS server has six transmit queues (tx). The NFS server is sending data to five clients using one TCP flow for each client. Linux hashes the TCP flows of Client1, Client3, and Client5 into tx3, tx2, and tx5, respectively; and hashes the flows of both Client2 and Client4 into tx0. Therefore, Client2 and Client4 shares one transmit queue and each gets half of the throughput of the queue.

between a client's start order and its chance of being a winner or loser.

By tracing the server's networking stack, we discovered that the winner-loser pattern is closely related to the server's use of physical queues in its network interface card (NIC). Every NIC has a physical transmit queue (`tx-queue`) holding outbound packets, and a physical receive queue (`rx-queue`) tracking empty buffers for inbound packets [105]. Many modern NICs have multiple sets of `tx-queues` and `rx-queues` to allow networking to scale with the number of CPU cores (each queue can be configured to interrupt a specific core), and to facilitate better NIC virtualization [105]. Linux uses hashing to decide which `tx-queue` to use for each outbound packet. However, not all packets are hashed; instead, each TCP socket has a field recording the `tx-queue` the last packet was forwarded to. If a socket has any existing packets in the recorded `tx-queue`, its next packet is also placed in that queue. This approach allows TCP to avoid generating out-of-order packets by placing packet n on a long queue and $n+1$ on a shorter one. However, a side effect is that for highly active TCP flows that always have outbound packets in the queue, the hashing is effectively done per-flow rather than per-packet. (On the other hand, if the socket has no packets in the recorded `tx-queue`, its next packet is re-hashed, probably to a new `tx-queue`.)

The winner-loser pattern is caused by uneven hashing of TCP flows to `tx-queues`. In our particular experiments, the server had five flows (one per client) and a NIC configured with six `tx-queues`. If two of the flows were hashed into one `tx-queue` and the rest went into three separate `tx-queues`, then the two flows sharing a `tx-queue` got half the throughput of the other three because all `tx-queues` were transmitting data at the same rate. We call this phenomenon—hashing unevenness causing a winner-loser pattern of throughput—*Hash-Cast*, which is illustrated in Figure 2.4.

Hash-Cast explains the performance anomalies illustrated in Figure 2.3. First, Client1, Client2, and Client3 were hashed into `tx3`, `tx0`, and `tx2`, respectively. Then, Client4 was hashed into `tx0`, which Client2 was already using. Later, Client5 was hashed into `tx3`, which Client1 was already using. However, at 270 seconds, Client5's `tx-queue` drained and it was rehashed into `tx5`. At the experiment's end, Client1, Client3, and Client5 were using `tx3`, `tx2`, and `tx5`, respectively, while Client2 and Client4 shared `tx0`. Hash-Cast also explains why the losers usually got half the throughputs of the winners: the {0,0,1,1,1,2} distribution is the most likely hashing result (we calculated its probability as roughly 69%).

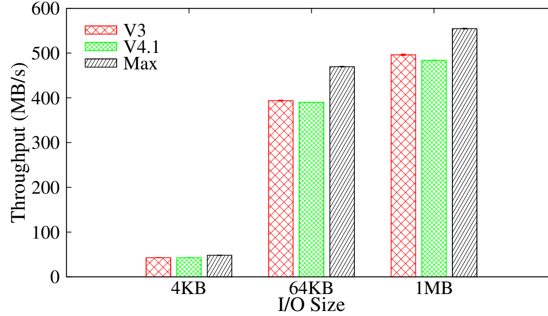


Figure 2.5: Random-write throughput in a zero-delay network with different I/O size. The higher the better. "Max" is the throughput when running the workload directly on the server side without using NFS. The "Max" throughput serves as a baseline when evaluating NFS's overhead.

To eliminate hashing unfairness, we evaluated the use of a single `tx-queue`. Unfortunately, we still observed an unfair throughput distribution across clients because of complicated networking algorithms such as *TSO Automatic Sizing*, which can form feedback loops that keep slow TCP flows always slow [23]. To resolve this issue, we further configured `tc qdisc` to use Stochastic Fair Queueing (SFQ), which achieves fairness by hashing flows to many software queues and sends packets from those queues in a round-robin manner [80]. Most importantly, SFQ used 127 software queues so that hash collisions were much less probable compared to using only 6 queues. To further alleviate the effect of collisions, we set SFQ's hashing perturbation rate to 10 seconds using `tc qdisc`, so that the mapping from TCP flows to software queues changed every 10 seconds.

Note that using a single `tx-queue` with SFQ did not reduce the aggregate network throughput compared to using multiple `tx-queues` without SFQ. We measured only negligible performance differences between these two configurations. We found that many of Linux's queuing disciplines assume a single `tx-queue` and could not be configured to use multiple ones. Thus, it might be desirable to use just one `tx-queue` in many systems, not just NFS servers. To ensure fairness among clients, for the remainder of experiments in this thesis proposal we used SFQ with a single `tx-queue`. The random-read results shown in Section 2.3.1 also used SFQ.

2.3.3 Random Write

The random-write workload is the same as the random-read one discussed in Section 2.3.1 except that the clients were writing data instead of reading. Each client had a number of threads that repeatedly wrote a specified amount (I/O size) of data at random offsets in a pre-allocated 20GB file. All writes were in-place and did not change the file size. We opened the file with `O_SYNC` set, to ensure that the clients write data back to the NFS server instead of just caching it locally. This setup is similar to many I/O workloads in virtualized environments [124], which use NFS heavily.

We varied the I/O size from 4KB to 1MB, the number of threads from 1 to 16, and the injected network delay from 0ms to 10ms. We ran the experiments long enough to ensure that the working sets, including in the 4KB I/O case, were at least 10 times larger than our RAID controller's cache size. As expected, larger I/O sizes and more threads led to higher throughputs, and longer network delays reduced throughput. V4.1 and V3 performed comparably, with V4.1 slightly worse (2% on average) in the zero-delay network.

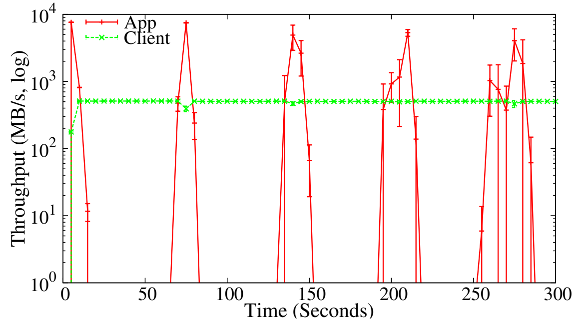


Figure 2.6: Random-write throughput of a single NFS client in a zero-delay network (log_{10}). The higher the better. The "App" curve is the throughput observed by the benchmarking application, i.e., the writing speed of the application to the file; the "Client" curve is the throughput observed by the NFS client, i.e., the writing speed of the clients to the remote NFS server.

Figure 2.5 shows the random-write throughput when we varied the I/O size in the zero-delay network. V4.1 and V3 achieved around the same throughput, but both were significantly slower than the maximum performance of our SSD RAID (measured on the server side without NFS). Neither V4.1 nor V3 achieved the maximum throughput even with more threads. We initially suspected that the lower throughputs were caused by the network, but the throughput did not improve when we repeated the experiment directly on the NFS server over the loopback device. Instead, we found the culprit to be the `O_SYNC` flag, which has different semantics in `ext4` than in NFS. The POSIX semantics of `O_SYNC` require all meta-data updates to be synchronously written to disk. On Linux's local file systems, however, `O_SYNC` is implemented so that only the actual file data and the meta-data directly needed to retrieve that data are written synchronously; other meta-data remains buffered. Since our workloads used only in-place writes, which updated the file's modification time but not the block mapping, writing an `ext4` file did not update meta-data. In contrast, the NFS implementation more strictly adheres to POSIX, which mandates that the server commit both the written data and "all file system metadata" to stable storage before returning results. Therefore, we observed many meta-data updates in NFS, but not in `ext4`. The overhead of those extra updates was aggravated by `ext4`'s journaling of meta-data changes on the server side. (By default `ext4` does not journal changes to file data.) The extra updates and the journaling introduced numerous extra I/Os, causing NFS's throughput to be significantly lower than the RAID-0's maximum (measured without NFS). This finding highlights the importance of understanding the effects of the NFS server's implementation and the underlying file system that it exports.

We also tried the experiments without setting `O_SYNC`, which generated a bursty workload to the NFS server, as shown in Figure 2.6. Clients initially realized high throughput (over 1GB/s) since all data was buffered in their caches. Once the number of dirty pages passed a threshold, the throughput dropped to near zero as the clients began flushing those pages to the server; this process took up to 3 minutes depending on the I/O size. After that, the write throughput became high again, until caches filled—and the bursty pattern then repeated.

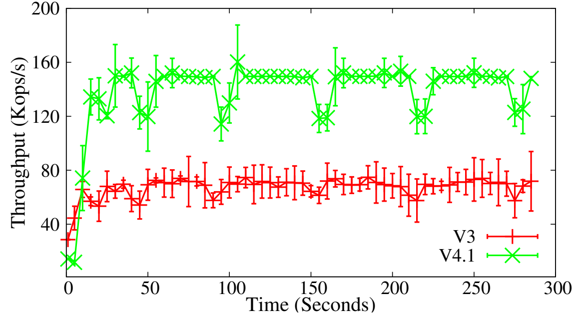


Figure 2.7: Throughput of reading small files with one thread in a zero-delay network.

2.3.4 Sequential Write

We also benchmarked sequential-write workloads, where each client had a single thread writing sequentially to the 20GB file. V4.1 and V3 again had the same performance. However, the aggregate throughputs of single-threaded sequential-write workloads were lower than the aggregate throughputs of their multi-threaded counterparts because our all-SSD storage backend has internal parallelism [3], and favors multi-threaded I/O accesses. For sequential writes, the O_SYNC behavior we discussed in Section 2.3.3 had an even stronger effect if the backend storage used HDDs, because the small disk writes generated by the meta-data updates and the associated journaling broke the sequentiality of NFS's writes to disk. We measured a 50% slowdown caused by this effect when we used HDDs for our storage backend instead of SSDs [22].

2.4 Metadata-Intensive Workloads

The data-intensive workloads discussed so far are more sensitive to network and I/O bandwidth than to latency. This section focuses on meta-data-intensive workloads, which are critical to NFS's overall performance because of the popularity of uses such as shared home directories, where common workloads like software development and document processing involve many small- to medium-sized files. We discuss three micro-workloads that exercise NFS's meta-data operations by operating on a large number of small files: file reads, file creations, and directory listings.

2.4.1 Read Small Files

We pre-allocated 10,000 4KB files on the NFS server. Figure 2.7 shows the results of the 5 clients randomly reading entire files repeatedly for 5 minutes. The throughputs of both V3 and V4.1 increased quickly during the first 10 seconds and then stabilized once the clients had read and cached all files. V4.1 started slower than V3, but outperformed V3 by 2× after their throughputs stabilized. We observed that V4.1 made 8.3× fewer NFS requests than V3 did. The single operation that caused this difference was GETATTR, which accounted for 95% of all the requests performed by V3. These GETATTRs were being used by the V3 clients to revalidate their client-side cache. However, V4.1 rarely made any requests once its throughput had stabilized. Further investigation revealed that this was caused by V4.1's delegation mechanism, which allows client-side caches

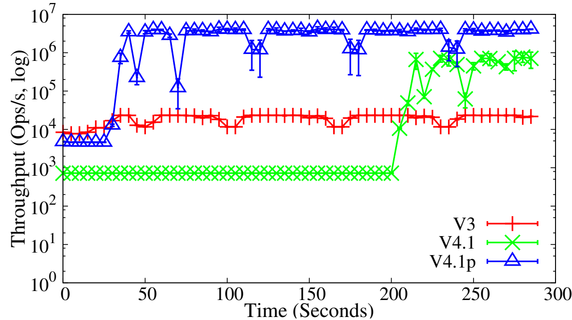


Figure 2.8: Aggregate throughput of reading small files with 16 threads in a 10ms-delay network ($\log_{10}$). The "V4.1" curve is the throughput of vanilla NFSv4.1; the "V4.1p" cruve is the throughput of the patched NFSv4.1 with our fix.

to be used without revalidation. We discuss and evaluate V4's delegations in greater detail in Section 2.5.

To investigate read performance with fewer caching effects, we used a 10ms network delay to increase the time it would take to cache all of the files. With this delay, the clients did not finish reading all of the files during the same 5-minute experiment. We observed that the client's throughput dropped to under 94 ops/s for V3 and under 56 ops/s for V4.1. Note that each V4.1 client made an average of 243 NFS requests per second, whereas each V3 client made only 196, which is counter-intuitive given that V4.1 had lower throughput. The reason for V4.1's lower throughput is its more verbose stateful nature: 40% of V4.1's requests are state-maintaining requests (e.g., OPENs and CLOSEs in this case), rather than READs. State-maintaining requests do not contribute to throughput, and since most files were not cached, V4.1's delegations could not help reduce the number of stateful requests.

To compensate for the lower throughput due to the 10ms network delay, we increased the number of threads on each client, and then repeated the experiment. Figure 2.8 shows the throughput results (log scale). With 16 threads per client V3's throughput (the red line) started at around 8100 ops/s and quickly increased to 55,800 ops/s. After that, operations were served by the client-side cache; only GETATTR requests were made for cache revalidation. V4.1's throughput (the green curve) started at only 723 ops/s, which is eleven times slower than that of V3. It took 200 seconds for V4.1 to cache all files; then V4.1 overtook V3, and afterwards performed 25× faster thanks to delegations. V4.1 also made 71% fewer requests per second than V3; this reversed the trend from the no-latency-added single-thread case (Figure 2.7), where V4.1 had lower throughput but made more requests.

To understand this behavior, we reviewed the `mountstat` data for the V4.1 tests. We found that the average RPC queuing time of V4.1's OPEN and CLOSE requests was as long as 223ms, while that average queuing time of all V4.1's other requests (ACCESS, GETATTR, LOOKUP, and READ) was shorter than 0.03ms. (The RPC queuing time is the time between when an RPC is initialized and when it begins transmitting over the wire.) This means that some OPENs and CLOSEs waited over 200ms in a client-side queue before the client started to transmit them.

To diagnose the long delays, we used `Systemtap` to instrument all the `rpc_wait_queues` in Linux's NFS client kernel module and found the culprit to be an `rpc_wait_queue` for `seqid`, which is an argument to OPEN and CLOSE requests [107]; it was used by V4.0 clients to notify the

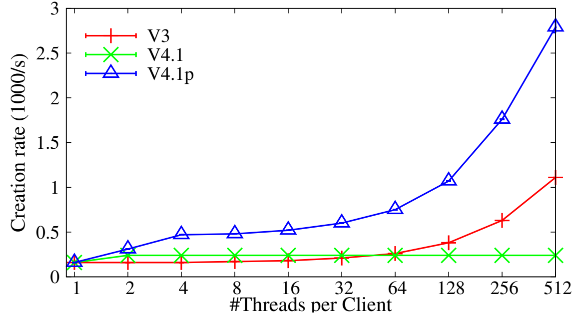


Figure 2.9: Aggregate throughput of creating empty files in a 10ms-delay network with different numbers of threats per client.

server of changes in client-side states. V4.0 requests that needed to change the `seqid` were fully serialized by this wait queue. The problem is exacerbated by the fact that once entered into this queue, a request is not dequeued until it receives the server's reply. However, `seqid` is obsolete in V4.1: the latest standard [107] explicitly states that "The 'seqid' field of the request is not used in NFSv4.1, but it MAY be any value and the server MUST ignore it."

We fixed the long queuing time for `seqid` by avoiding the queue entirely. (We have submitted a patch to the kernel mailing list.) For V4.0, `seqid` is still used and our patch does not change its behavior. We repeated the experiments with these changes; the new results are shown as the blue curve in Figure 2.8. V4.1's performance improved by more than 6× (from 723 ops/s to 4655 ops/s). V4.1 finished reading all the files within 35 seconds, and thereafter stabilized at a throughput 172× higher than V3 because of delegations. In addition to the higher throughput, V4.1's average response time was shorter than that of V3, also because of delegations. For brevity, we refer to the patched NFSv4.1 as V4.1p in following discussions.

We noted a periodic performance drop every 60 seconds in Figures 2.7 and 2.8, which corresponds to the `actimeo` mount option. When this timer expires, client-cached metadata must again be retrieved from the server, temporarily lowering throughput. Enlarging the `actimeo` mount option is a way to trade cache consistency for higher performance.

2.4.2 File Creation

We demonstrated above that client-side caching, especially delegations, can greatly reduce the number of NFS meta-data requests when reading small files. To exercise NFS's meta-data operations more, we now turn to a file-creation workload, where client-side caching is less effective. We exported one NFS directory for each of the five clients, and instructed each client to create 10,000 files of a given size in its dedicated directory, as fast as possible.

Figure 2.9 shows the speed of creating empty files in the 10ms-delay network. To test scalability, we varied the number of threads per client from 1 to 512. V4.1 started at the same speed as V3 when there was only one thread per client. Between 2–32 threads, V4.1 outperformed V3 by 1.1–1.5×, and V4.1p (NFSv4.1 with our patch) outperformed V3 by 1.9–2.9×. Above 32 threads, however, V4.1 became 1.1–4.6× *slower* than V3, whereas V4.1p was 2.5–2.9× *faster* than V3.

As shown in Figure 2.9, when the number of threads per client increased from 1 to 16, V3 sped up by only 12.5%, V4.1 by 50%, and V4.1p by 225%. In terms of scalability (1–16 threads),

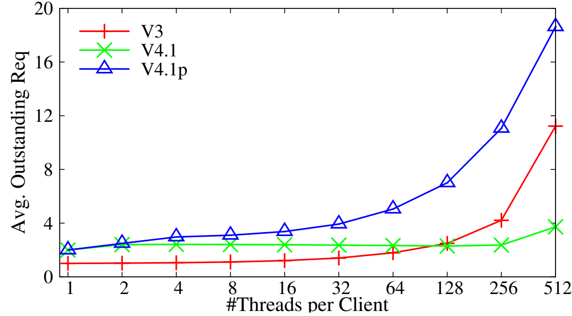


Figure 2.10: Average number of outstanding requests when creating empty files in a 10ms-delay network.

V3 scaled poorly, with an average performance boost of merely 3% each power-of-two step in the thread count. V4.1 scaled slightly better, with an average 10% boost per step. But, because of the `seqid` synchronizing bottleneck explained in Section 2.4.1, its performance did not improve at all once the thread count increased beyond two. With the `seqid` problem fixed, V4.1p scaled much better, with an average boost of 34% per step. With 16–512 threads, V3's scalability improved significantly, and it achieved a high average performance boost of 44% per step; V4.1p also scaled well with an average boost of 40% per step.

V4.1p always outperformed the original V4.1, by up to 11.6× with 512 threads. Therefore, for the rest of this thesis proposal, we only report figures for V4.1p, unless otherwise noted.

In the 10ms-delay network, the rates of creating empty, 4KB, and 16KB files differed by less than 2% when there were more than 4 threads, and by less than 27% with fewer threads; thus, they all had the same trends. As shown in Figure 2.9, depending on the number of threads, V4.1p created small files 1.9–2.9× faster than V3 did. To understand why, we analyzed the `mountstats` data and found that the two versions differed significantly in the number of outstanding NFS requests (i.e., requests sent but not yet replied to). We show the average number of outstanding NFS requests in Figure 2.10, which closely resembles Figure 2.9 in overall shape. This suggests that V4.1p performed faster than V3 because the V4.1p clients sent more requests to the server at one time. We examined the client code and discovered that V3 clients use synchronous RPC calls (rpc_call_sync) to create files, whereas V4.1p clients use asynchronous calls (rpc_call_async) that go through a work queue (nfsiod_workqueue). We believe that the asynchronous calls are the reason why V4.1p had more outstanding requests: the long network delay allowed multiple asynchronous calls to accumulate in the work queue and be sent out in batch, allowing networking algorithms such as TCP Nagle to efficiently coalesce multiple RPC messages. Sending fewer but larger messages is faster than sending many small ones, so V4.1p achieved higher rates. Our analysis was confirmed by the `mountstats` data, which showed that V4.1p's OPEN requests had significantly longer queuing times (up to 30×) on the client side than V3's CREATEs. (V3 uses CREATEs to create files whereas V4.1p uses OPENs.) Because V3's server is stateless, all its mutating operations have to be synchronous; otherwise a server crash might lose data. V4, however, is stateful and can perform mutating operations asynchronously because it can restore states properly in case of server crashes [88].

In the zero-delay network, there was not a consistent winner between the two NFS versions (Figure 2.11). Depending on the number of threads, V4.1p varied from 1.76× faster to 3× slower

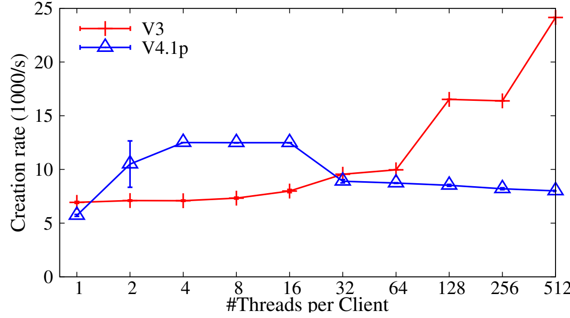


Figure 2.11: Rate of creating empty files in a zero-delay network.

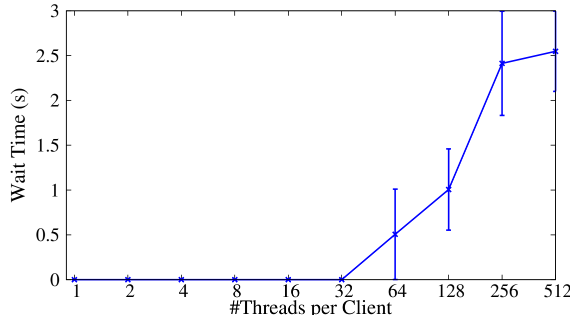


Figure 2.12: Average waiting time for V4.1p's session slots of ten experimental runs. Error bars are standard deviations.

than V3. In terms of scalability, V3's speed increased slowly when we began adding threads, but jumped quickly between 64 and 512 threads. In contrast, V4.1p's speed improved quickly at the initial stage, but plateaued and then dropped when we used more than 4 threads.

To understand why, we looked into the `mountstats` data, and found that the corresponding graph (not shown) of the average number of outstanding requests closely resembles Figure 2.11. It again suggests that the lower speed was the result of a client sending requests rather slowly. With further analysis, we found that V4.1p's performance drop after 32 threads was caused by high contention for session slots, which are V4.1p's unique resources the server allocates to clients. Each session slot allows one request; if a client runs out of slots (i.e., has reached the maximum number of concurrent requests the server allows), it has to wait until one becomes available, which happens when the client receives a reply for any of its outstanding requests. We instrumented the client kernel module and collected the waiting time on the session slots. As shown in Figure 2.12, waiting began at 32 threads, which is also where V4.1p's performance began dropping (Figure 2.11). Note that with 512 threads, the average waiting time is 2500ms, or 12,500$\times$ the 0.2ms round-trip time. (The 10ms-delay experiment also showed waiting for session slots, but the wait time was short compared to the network RTT and thus had a smaller effect on performance.)

We note that Figure 2.12 had high standard deviations above 32 threads per client. This behavior results from typical non-real-time scheduling artifacts in Linux, where some threads can win and be scheduled first, while others wait longer. Even when we ran the same experiment 10 times, standard deviations did not decrease, suggesting a non-Gaussian, multi-modal distribution [122]. In addition to V4.1p's higher wait time in this figure, the high standard deviation means that it

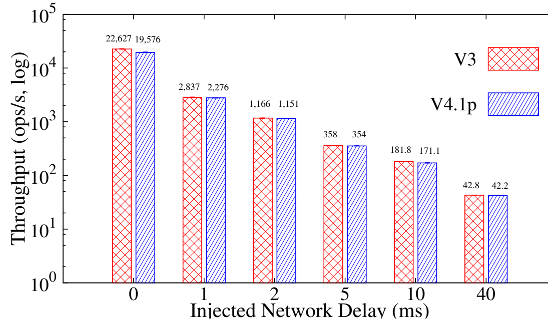


Figure 2.13: Directory listing throughput ($\log_{10}$).

would be harder to enforce SLAs with V4.1p for highly-concurrent applications.

With 2–16 threads (Figure 2.11), V4.1p's performance advantage over V3 was because of V4.1p's asynchronous calls (the same as explained above); V4.1p's OPENs had around 4× longer queuing time, which let multiple requests accumulate and be sent out in batch. This queuing time was not caused by the lack of available session slots (Figure 2.12). This was verified by evaluating the use of a single thread, in which case V4.1p performed 17% slower than V3 because V4.1p's OPEN requests are more complex and took longer to process than V3's CREATE (see Section 2.4.3).

One possible solution to V4.1p's session-slot bottleneck is to enlarge the number of server-side slots to match the client's needs. However, slots consume resources: for example, the server must then increase its *duplicate request cache* (DRC) size to maintain its *exactly once semantics* (EOS). Increasing the DRC size can be expensive, because the DRC has to be persistent and is possibly saved in NVRAM. V3 does not have this issue because it does not provide EOS, and does not guarantee that it will handle retransmitted non-idempotent operations correctly. Consequently, V3 outperformed V4.1p when there were more than 64 threads (Figure 2.11).

2.4.3 Directory Listing

We now turn to another common meta-data-intensive workload: listing directories. We used Filebench's directory-listing workload, which operates on a pre-allocated NFS directory tree that contains 50,000 empty files and has a mean directory width of 5. Each client ran one Filebench instance, which repeatedly picks a random subdirectory in the tree and lists its contents.

This workload is read-only, and showed behavior similar to that of reading small files (Section 2.4.1) in that its performance depended heavily on client-side caching. Once all content was cached, the only NFS requests were for cache revalidations. Figure 2.13 (log scale) shows the throughput of single-threaded directory listing in networks with different delays. In general, V4.1p performed slightly worse than V3. The biggest difference was in the zero-delay network, where V4.1p was 15% slower. `mountstats` showed that V4.1p's requests had longer round-trip times, which implies that the server processed those requests slower than V3: 10% slower for READDIR, 27% for GETATTR, 33% for ACCESS, and 36% for LOOKUP. This result is predictable because the V4.1p protocol, which is stateful and has more features (EOS, delegations, etc.), is substantially more complex than V3. As we increased the network delay, the processing time of V4.1p became less important: V4.1p's performance was within 94–99% of V3. Note that Linux does not support directory delegation.

With 16 threads, V4.1p's throughput was 95–101% of V3's. Note that V4.1p's asynchronous RPC calls did not influence this workload much because most of this workload's requests did not mutate states. Only the state-mutating V4.1p requests are asynchronous: OPEN, CLOSE, LOCK, and LOCKU. (WRITE is also asynchronous, but this workload does not have any WRITEs.)

2.5 NFSv4 Delegations

In this section, we discuss delegations, an advanced client-side caching mechanism that is a key new feature of NFSv4. Caching is essential to good performance in any system, but in distributed systems like NFS caching gives rise to consistency problems. V2 and V3 explicitly ignored strict consistency [20, p. 10], but supported a limited form of validation via the GETATTR operation. In practice, clients validate their cache contents frequently, causing extra server load and adding significant delay in high-latency networks.

In V4, the cost of cache validation is reduced by letting a server *delegate* a file to a particular client for a limited time, allowing accesses to proceed at local speed. Until the delegation is released or recalled, no other client is allowed to modify the file. This means a client need not revalidate the cached attributes and contents of a file while holding the delegation of the file. If any other clients want to perform conflicting operations, the server can recall the delegation using *callbacks* via a server-to-client back-channel connection. Delegations are based on the observation that file sharing is infrequent [107] and rarely concurrent [69]. Thus, they can boost performance most of the time, although with performance penalty in the rare presence of concurrent and conflicting file sharing.

Delegations have two types: *open delegations* of files, and *directory delegations*. The former comes in either "read" or "write" variants. We will focus on read delegations of regular files because they are the simplest and most common type—and are also the only delegation type currently supported in the Linux kernel [39].

2.5.1 Granting a Delegation

An open delegation is granted when a client opens a file with an appropriate flag. However, clients must not assume that a delegation will be granted, because that choice is up to the server. If a delegation is rejected, the server can explain its decision via flags in the open reply (e.g., lock contention, unsupported delegation type). Even if a delegation is granted, the server is free to recall it at any time via the back channel, which is a RPC channel that enables the NFS servers to notify clients. Recalling a delegation may involve multiple clients and multiple messages, which may lead to considerable delay. Thus, the decision to grant the delegation might be complex. However, because Linux currently supports only file-read delegations, it uses a simpler decision model. The delegation is granted if three conditions are met: (1) the back channel is working, (2) the client is opening the file with `O_RDONLY`, and (3) the file is not currently open for write by any client.

During our initial experiments we did not observe any delegations even when all three conditions held. We traced the kernel using `SystemTap` and discovered that the Linux NFS server's implementation of delegations was outdated: it did not recognize new delegation flags introduced by NFSv4.1. The effect was that if an NFS client got the filehandle of a file before the client

Operation	**NFSv3**	**NFSv4.1 deleg. off**	**NFSv4.1 deleg. on**
OPEN	0	10,001	1000
READ	10,000	10,000	1000
CLOSE	0	10,001	1000
ACCESS	10,003	9003	3
GETATTR	19,003	19,002	1
LOCK	10,000	10,000	0
LOCKU	10,000	10,000	0
LOOKUP	1002	2	2
FREE_STATEID	0	10,000	0
TOTAL	**60,008**	**88,009**	**3009**

Table 2.1: NFS operations performed by each client for NFSv3 and NFSv4.1 (delegations on and off). Each NFSv4.1 operation represents a compound procedure. For clarity, we omit trivial operations (e.g., PUTFH) in compounds. NFSv3's LOCK and LOCKU come from the Network Lock Manager (NLM).

opened the file (e.g., using `stat`), no delegation was granted. We fixed the problem with a kernel patch, which has been accepted into the mainline Linux kernel.

2.5.2 Delegation Performance: Locked Reads

We previously showed the benefit of delegations in Figure 2.8, where delegations helped V4.1p read small files 172× faster than V3. This improvement is due to the elimination of cache revalidation traffic; no communication with the server (GETATTRs) is needed to serve reads from cache. Nevertheless, delegations can improve performance even further in workloads with file locking. To quantify the benefits, we repeated the delegation experiment performed by Gulati [54] but scaled it up. We pre-allocated 1000 4KB files in a shared NFS directory and then mounted it on the five clients. Each client repeatedly opened each of the files in the shared NFS directory, locked it, read the entire file, and then unlocked it. (Locking the file is a technique used to ensure an atomic read.) After ten repetitions the client moved to the next file.

Table 2.1 shows the number of operations performed by V3 and by V4.1p with and without delegation. Only V4.1p shows OPENs and CLOSEs because only V4 is stateful. When delegations were on, V4.1p used only 1000 OPENs even though each client opened each file ten times. This is because each client obtained a delegation on the first OPEN; the following nine were performed locally. Note that in Table 2.1, without a delegation (for V3 and V4.1p with delegations off), each application read incurred an expensive NFS READ operation even though the same reads were repeated ten times. Repeated reads were not served from the client-side cache because of file locking, which forces the client to revalidate the data.

Another cause of the difference in the number of READ operations in Table 2.1 is the timestamp granularity on the NFS server. Traditionally, NFS provides close-to-open cache consistency [71]. Timestamps are updated at the server when a file is closed, and any client subsequently opening the same file revalidates its local cache by checking its attributes with the server. If the locally-saved timestamp of the file is out of date, the client's cache of the file is invalidated. Unfortunately, some

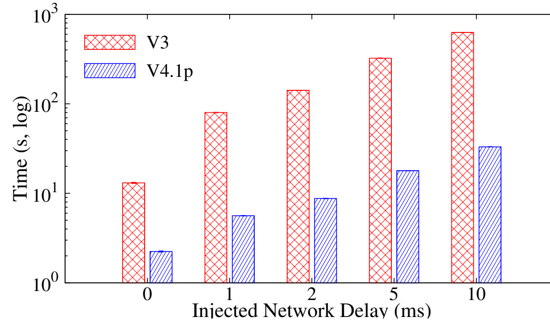


Figure 2.14: Running time of the locked-reads experiment ($\log_{10}$). Lower is better.

NFS servers offer only one-second granularity, which is too coarse for modern systems; clients could miss intermediate changes made by other clients within one second. In this situation, NFS locking provides stronger cache coherency by first checking the server's timestamp granularity. If the granularity is finer than one microsecond, the client revalidates the cache with GETATTR; otherwise, the client invalidates the cache. Since the Linux in-kernel server uses a one-second granularity, each read operation incurs a READ RPC request because the preceding LOCK has invalidated the client's local cache.

Invalidating an entire cached file can be expensive, since NFS is often used to store large files such as virtual disk images [124], media files, etc. The problem is worsened by two factors: (1) invalidation happens even when the client is simply acquiring read (not write) locks, and (2) a file's entire cache contents are invalidated even if the lock only applies to a single byte. In contrast, the NFS client with delegation was able to satisfy nine of the ten repeated READs from the page cache. There was no need to revalidate the cache at all because its validity was guaranteed by the delegation.

Another major difference among the columns in Table 2.1 was the number of GETATTRs. In the absence of delegation, GETATTRs were used for two purposes: to revalidate the cache upon file open, and to update file meta-data upon read. The latter GETATTRs were needed because the locking preceding the read invalidated both the data and meta-data caches for the locked file. A potential optimization for V4.1p would be to have the client append a GETATTR to the LOCK in the same compound, and let the server piggyback file attributes in its reply. This could save 10,000 GETATTR RPCs.

The remaining differences between the experiments with and without delegations were due to locking. A LOCK/LOCKU pair is sent to the server when the client does not have a delegation. Conversely, no NFS communication is needed for locking when a delegation exists. For V4.1p with delegations off, one FREE_STATEID follows each LOCKU to free the resource (stateid) used by the lock at the server. (A potential optimization would be to append the FREE_STATEID operation to the same compound procedure that includes LOCKU; this could save another 10,000 RPCs.)

In total, delegations cut the number of V4.1p operations by over 29× (from 88K to 3K). This enabled the original stateful and "chattier" V4.1p (with extra OPEN, CLOSE, and FREE_STATEID calls) to finish the same workload using only 5% of the requests used by V3. In terms of data volume, V3 sent 3.8MB and received 43.7MB, whereas V4.1p with delegation sent 0.6MB and received 4.5MB. Delegation helped V4.1p reduce the outgoing traffic by 6.3× and the incoming traffic by 9.7×. As seen in Figure 2.14, these reductions translate to a 6–19× speedup in networks

with 0–10ms latency.

2.5.3 Delegation Recall Impact

We have shown that delegations can effectively improve NFS performance when there is no conflict among clients. To evaluate the overhead of conflicting delegations, we created two groups of NFS clients: the Delegation Group (DG) grabs and holds NFS delegations on 1000 files by opening them with the `O_RDONLY` flag, while the Recall Group (RG), recalls those delegations by opening the same files with `O_RDWR`. To test scalability, we varied the number of RG clients from one to four. For n clients in the DG, an RG open generated n recalls because each DG client's delegation had to be recalled separately.

We compared the cases when the DG clients were and were not holding delegations. Each DG client needed two operations to respond to a recall: a DELEGRETURN to return the delegation, and an OPEN to re-open the file (since the delegation was no longer valid).

For the RG client, the presence of a delegation incurred one additional NFS OPEN per file. The first OPEN failed, returning an `NFS4ERR_DELAY` error to tell the client to try again later because the server needed to recall outstanding delegations. The second open was sent as a retry and succeeded.

The running time of the experiment varied dramatically, from 0.2 seconds in the no-delegation case to 100 seconds with delegation. This $500\times$ delay was introduced by the RG client, which failed in the first OPEN and retried it after a timeout. The initial timeout length is hard-coded to 100ms in the client kernel module (`NFS4_POLL_RETRY_MIN` in the Linux source code), and is doubled every time the retry fails. This long timeout was the dominating factor in the experiment's running time.

To test delegation recall in networks with longer latencies, we repeated the experiment after injecting network delays from 1–10ms. Under those conditions, the experiment's running time increased from 100s to 120s. With 10ms of extra network latency, the running time was still dominated by the client's retry timeout. However, when we increased the number of clients in DG from one to four, the total running time did not change. This suggests the delegation recall works well when there are several clients holding conflicting delegations at the same time.

We believe that a long initial timeout of 100ms is questionable considering that most SLAs specify a latency of 10–100ms [4]. Also, because Linux does not support write delegations, Linux NFS clients do not have any dirty data (of delegated files) to write back to the server, and thus should be able to return delegations quickly. We believe it would be better to start with a much shorter timeout; if that turns out to be too small, the client will back-off quickly anyway since the timeout increases exponentially.

Recalling read delegations is relatively simple because the clients holding them have no dirty data to write back to the server. For write delegations, the recall will be more difficult because the amount of dirty data can be substantial—since the clients are free from network communication, they are capable of writing data faster than in normal NFS scenarios. The cost of recalling write delegations would be interesting to study when they become available.

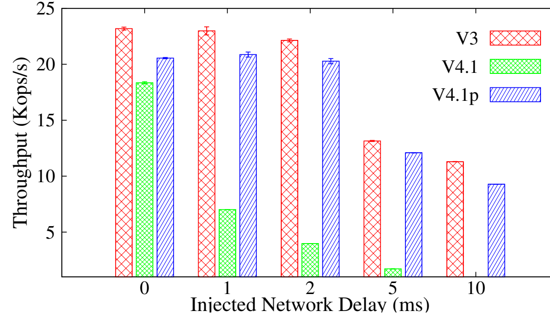


Figure 2.15: File Server throughput (varying network delay)

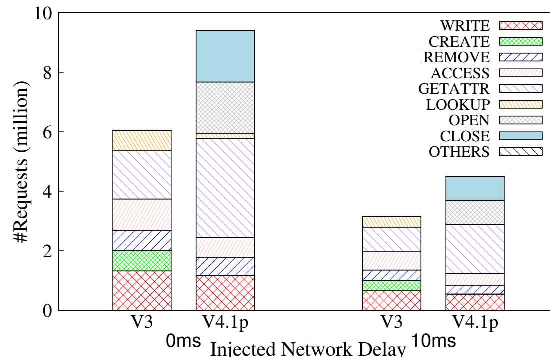


Figure 2.16: Number of NFS requests made by the File Server

2.6 Macro-Workloads

We now turn to macro-workloads that mix data and meta-data operations. These are more complex than micro-workloads, but also more closely match the real world. Our study used Filebench's File Server, Web Server, and Mail Server workloads [68].

2.6.1 The File Server Workload

The File Server workload includes opens, creates, reads, writes, appends, closes, stats, and deletes. All dirty data is written back to the NFS server on close to enforce NFS's close-to-open semantics. We created one Filebench instance for each client and ran each experiment for 5 minutes. We used the File Server workload's default settings: each instance had 50 threads operating on 10,000 files (in a dedicated NFS directory) with an average file size of 128KB, with the sizes chosen using Filebench's gamma function [136].

As shown in Figure 2.15, V4.1p had lower throughput than V3. Without any injected network delay, V4.1p's throughput was 12% lower because V4.1p is stateful and more talkative. To maintain state, V4.1p did 3.5 million OPENs and CLOSEs (Figure 2.16), which was equivalent to 58% of all V3's requests. Note that 0.6 million of the OPENs not only maintained states, but also created files. Without considering OPEN and CLOSE, V4.1p and V3 made roughly the same number of requests: V4.1p sent 106% more GETATTRs than V3 did, but no CREATEs and 78% fewer LOOKUPs.

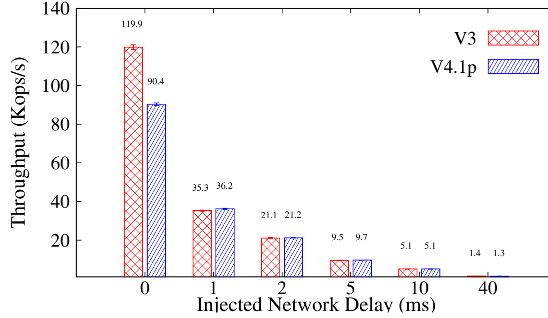


Figure 2.17: Web Server throughput (varying network delay).

V4's verbosity hurts its performance, especially in high-latency networks. We observed the same problems in other workloads such as small-file reading (Section 2.4.1), where V4 was 40% slower than V3 with a single thread and a 10ms-delay network. Verbosity is the result of V4's stateful nature, and the V4 designers were aware of the issue. To reduce verbosity, V4 provides compound procedures, which pack multiple NFS operations into one message. However, compounds have not been implemented effectively in Linux (and other OSes): most contain only 2–4 often-trivial operations (e.g., SEQUENCE, PUTFH, and GETFH); and applications currently have no ability to generate their own compounds. We believe that implementing effective compounds is difficult for two reasons: (1) The POSIX API dictates a synchronous programming model: issue one system call, wait, check the result, and only then issue the next call. (2) Without transaction support, failure handling in compounds with many operations is fairly difficult.

In this File Server workload, even though V4.1p made a total of 56% more requests than V3, V4.1p was only 12% slower because its asynchronous calls allowed 40–95% more outstanding requests (as explained in Section 2.4.2). When we injected delay into the network (Figure 2.15), V4.1p continued to perform slower than V3, by 8–18% depending on the delay. V4.1p's delegation mechanism did not help for the File Server workload because it contains mostly writes, and most reads were cached (also Linux does not currently support write delegations).

Figure 2.15 also includes the unpatched V4.1. As we increased the network delay, V4.1p performed increasingly better than V4.1, eventually reaching a $10.5\times$ throughput improvement. We conclude that our patch helps V4.1's performance in both micro- and macro-workloads, especially as network delays increase.

2.6.2 The Web Server Workload

Filebench's Web Server workload emulates servicing HTTP requests: 100 threads repeatedly operate on 1000 files, in a dedicated directory per client, representing HTML documents with a mean size of 16KB. The workload reads 10 randomly-selected files in their entirety, and then appends 16KB to a log file that is shared among all threads, causing contention. We ran one Web Server instance on each of the five NFS clients.

Figure 2.17 shows the throughput with different network delays. V4.1p was 25% slower than V3 in the zero-delay network. The `mountstats` data showed that the average round-trip time (RTT) of V4.1p's requests was 19% greater than for V3. As the network delay increased, the RPC RTT became overshadowed by the delay, and V4.1p's performance became close to V3's

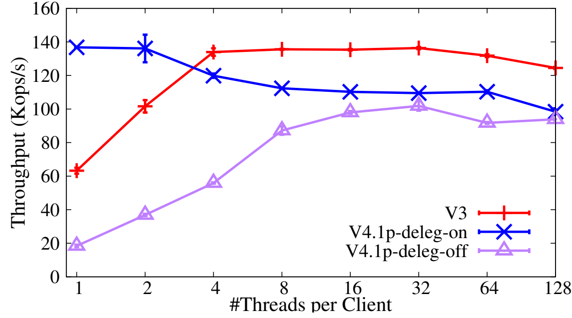


Figure 2.18: Web Server throughput in the zero-delay network (varying thread count per client).

and even slightly better (up to 2.6%). V4.1p's longer RTT was due to its complexity and longer processing time on the server side, as explained in Section 2.4.3. With longer network delays, V4.1p's performance picked up and matched V3's because of its use of asynchronous calls.

To test delegations, we turned on and off the `readonly` flag of the Filebench workload, and confirmed that setting `readonly` enabled delegations. In the zero-delay network, delegations reduced the number of V4.1p's `getattr` requests from over 8.7M to only 11K, and `opens` and `closes` from over 8.8M to about 10K. In summary, delegations cut the total number of all NFS requests by more than 10×. However, the substantial reduction in requests did not bring a corresponding performance boost: the throughput increased by only 3% in the zero-delay network, and actually decreased by 8% in the 1ms-delay situation. We were able to identify the problem as the writes to the log file. With delegations, each Web Server thread finished the first 10 reads from the client-side cache without any network communication, but then was blocked at the last write operation.

To characterize the bottleneck, we varied the number of threads in the workload and repeated the experiments with delegations both on and off. Figure 2.18 shows that delegations improved V4.1p's single-threaded performance by 7.4×, from 18 to 137 Kops/s. As the thread count increased, the log write began to dominate and delegations' benefit decreased, eventually making no difference: and the two curves of V4.1p in Figure 2.18 converged. With delegations, V4.1p was 2.2× faster than V3 when using one thread. However, V4.1p began to slow down with 4 threads, whereas V3 sped up and did not slow down until the thread number increased to 64. The eventual slowdown of both V3 and V4.1p was because the system became overloaded when the log-writing bottleneck was hit. However, V4.1p hit the bottleneck with fewer threads than V3 did because V4.1p, with delegations, only performed repeated WRITEs, whereas V3 performed ten GETATTRs (for cache revalidation) before each WRITE. With more than 32 threads, V4.1p's performance was also hurt by waiting for session slots (see Section 2.4.2).

This Web Server macro-workload demonstrated how the power of V4.1p's delegations can be limited by the absence of write delegations in the current version of Linux. Any real-world application that is not purely read-only might quickly bottleneck on writes even though read delegations can eliminate most NFS read and revalidation operations. However, write delegations will not help if all clients are writing to a single file, such as a common log.

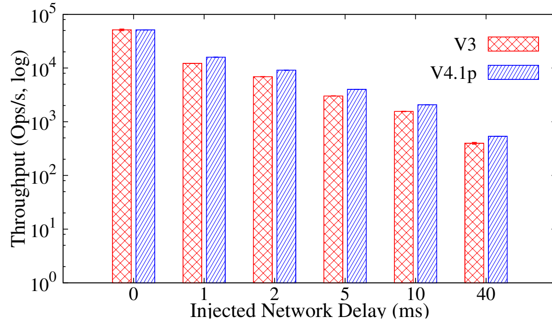
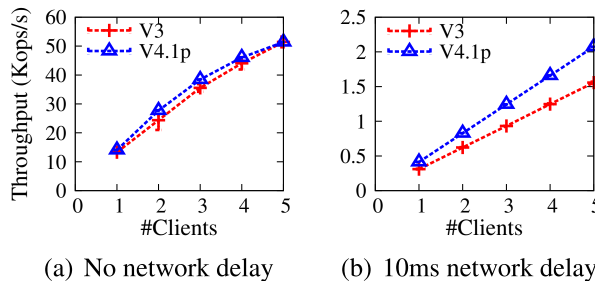


Figure 2.19: Mail Server throughput (varying network delay)



(a) No network delay (b) 10ms network delay

Figure 2.20: Mail Server throughput (varying client count)

2.6.3 The Mail Server Workload

Filebench's Mail Server workload mimics `mbox`-style e-mail activities, including compose, receive, read, and delete. Each Mail Server instance has 16 threads that repeat the following sets of operations on 1000 files in a flat directory: (1) create, write, fsync, and close a file (compose); (2) open, read, append, fsync, and close a file (receive); (3) open, read, and close a file (read); (4) delete a file (delete). The initial average file size was 16KB, but that could increase if appends were performed. We created a dedicated NFS directory for each NFS client, and launched one Mail Server instance per client. We tested different numbers of NFS clients, in addition to different network delays.

Figure 2.19 (note the log Y scale) presents the Mail Server throughput with different network delays. Without delay, V4.1p and V3 had the same throughput; with 1–40ms delay, V4.1p was 1.3–1.4× faster. Three factors affected V4.1p's performance: (1) V4.1p made more NFS requests for the same amount of work (see Section 2.6.1); and (2) V4.1p's operations were more complex and had longer RPC round-trip times (see Section 2.4.3); but (3) V4.1p made many asynchronous RPC calls and helped the networking algorithms coalesce RPC messages (see Section 2.4.1). Although the first two factors hurt V4.1p's performance, the third more than compensated for them. Increasing the network delay did not change factor (1), but diminished the effect of (2) as the delay gradually came to dominate the RPC RTT. Longer network delays also magnified the benefits of factor (3) because longer round trips were mitigated by coalescing requests. Thus, V4.1p increasingly outperformed V3 (1.3–1.4×) as the delay grew. V4.1p's read delegations did not help in this workload because most of its activities write files (reads are largely cached). This again shows the potential benefit of write delegations, even though Linux does not currently support them.

Figure 2.20 shows the aggregate throughput of the Mail Server workload with different num-

bers of NFS clients in the zero- and 10ms-delay networks. With zero delay, the aggregate throughput increased linearly from 1 to 3 clients, but then slowed because the NFS server became heavily loaded. An injected network delay of 10ms significantly reduced the NFS request rate: the server's load was much lighter, and although the aggregate throughput was lower, it increased linearly with the number of clients.

2.7 Related Work

NFS versions 2 and 3 are popular and have been widely deployed and studied. Stern et al. performed NFS benchmarking for multiple clients using nhfsstone [118]. Wittle and Keith designed the LADDIS NFS workload that overcomes nhfsstone's drawback, and measured NFS response time and throughput under various loads [137]. Based on LADDIS, the SPECsfs suites were designed to benchmark and compare the performance of different NFS server implementations [116]. Martin and Culler [77] studied NFS's behavior on high performance networks. They found that NFS servers were most sensitive to processor overhead, but insensitive to network bandwidth due to the dominant effect of small meta-data operations. Ellard and Seltzer designed a simple sequential-read workload to benchmark and improve NFS's readahead algorithm [36]; they also studied several complex NFS benchmarking issues including the ZCAV effect, disks' I/O reordering, the unfairness of disk scheduling algorithms, and differences between NFS over TCP vs. UDP. Boumenot conducted a detailed study of NFS performance problems [17] in Linux, and found that the low throughput of Linux NFS was caused not by processor, disk, or network performance limits, but by the NFS implementation's sensitivity to network latency and lack of concurrency. Lever et al. introduced a new sequential-write benchmark and used it to measure and improve the write performance of Linux's NFS client [70].

Most prior studies [17, 36, 70, 77, 116, 137] were about V2 and V3. NFS version 4, the latest NFS major version, is dramatically different from previous versions, and is far less studied in the literature. Prior work on V4 focuses almost exclusively on V4.0, which is quite different than V4.1 due to the introduction of sessions, Exactly Once Semantics (EOS), and pNFS. Harrington et al. summarized major NFS contributors' efforts in testing the correctness and performance of Linux's V4.0 [9] implementation. Radkov et al. compared the performance of a prototype version of V4.0 and iSCSI in IP-networked storage [100]. Martin [76] compared the file operation performance between Linux V3 and V4.0; Kustarz [68] evaluated the performance of Solaris's V4.0 implementation and compared it with V3. However, Martin and Kustarz studied only V4.0's basic file operations without exercising unique features such as statefulness and delegations. Hildebrand and Honeyman explored the scalability of storage systems using pNFS, an important part of V4.1. Eshel et al. [74] used V4.1 and pNFS to build Panache, a clustered file system disk cache that shields applications from WAN latency and outages while using shared cloud storage.

Only a handful of authors have studied the delegation mechanisms provided by NFSv4. Batsakis and Burns extended V4.0's delegation model to improve the performance and recoverability of NFS in computing clusters [11]. Gulati et al. built a V4.0 cache proxy, also using delegations, to improve NFS's performance in WANs [54]. However, both of these studies were concerned more with enhancing NFS's delegations to design new systems rather than evaluating the impact of standard delegations on performance. Moreover, they used V4.0 instead of V4.1. Although Panache is based on V4.1, it revalidated its cache using the traditional method of checking timestamps of file

objects instead of using delegations.

As the latest minor version of V4, V4.1's Linux implementation is still evolving [39]. To the best of our knowledge there are no existing, comprehensive performance studies of Linux's NFSv4.1 implementation that cover its advanced features such as statefulness, sessions, and delegations.

NFS's delegations are partly inspired by Andrew File System (AFS). AFS stores and moves files at the unit of whole files [57], and it breaks large files into smaller parts when necessary. AFS clients cache files locally and push dirty data back to the server only when files are closed. AFS clients re-validate cached data when clients use the data for the first time after restart; AFS servers will invalidate clients' cache with update notification when files are changed.

2.8 Conclusions

We have presented a comprehensive benchmarking study of Linux's NFSv4.1 implementation by comparison to NFSv3. Our study found that: **(1)** V4.1's read delegations can effectively avoid cache revalidation and help it perform up to 172× faster than V3. **(2)** Read delegations alone, however, are not enough to significantly improve the overall performance of realistic macro-workloads because V4.1 might still be bottlenecked by write operations. Therefore, we believe that write delegations are needed to maximize the benefits of delegations. **(3)** Moreover, delegations should be avoided in workloads that share data, since conflicts can incur a delay of at least 100ms. **(4)** We found that V4.1's stateful nature makes it more talkative than V3, which hurts V4.1's performance and makes it slower in low-latency networks (e.g., LANs). Also, V4.1's compound procedures, which were designed to help the problem, are not in practice effective. **(5)** However, in high-latency networks (e.g., WANs), V4.1's performed comparably to and even better than V3's since V4.1's statefulness permits higher concurrency through asynchronous RPC calls. For highly threaded workloads, however, V4.1 can be bottlenecked by the number of session slots. **(6)** We also showed that NFS's interactions with the networking and storage subsystems are complex, and system parameters should be tuned carefully to achieve high NFS throughput. **(7)** We identified a Hash-Cast networking problem that causes unfairness among NFS clients, and presented a solution. **(8)** Lastly, we made improvements to Linux's V4.1 implementation that boost its performance by up to 11×.

With this comprehensive benchmarking study, we conclude that NFSv4.1's performance is comparable to NFSv3. Therefore, we plan to support NFSv4.1 in Kurma. We also believe that NFSv4.1's compound procedures, which are currently woefully underutilized, hold much promise for significant performance improvement. We plan to implement more advanced compounds, such as transactional NFS compounds that can coalesce many operations and execute them atomically on the server. With transactional compounds, programmers, instead of waiting and then checking the status of each operation, can perform many operations at once and use exception handlers to deal with failures. Such a design could greatly simplify programming and improve performance at the same time.

2.8.1 Limitations

This benchmarking study has two limitations: **(1)** Most of our workloads did not share files among clients. Because sharing is infrequent in the real world [107], it is critical that any sharing

be representative. One solution would be to replay multi-client NFS traces from real workloads, but this task is challenging in a distributed environment. **(2)** Our WAN emulation using `netem` was simple, and did not consider harsh packet loss, intricate delays, or complete outages in real networks.

Chapter 3

SeMiNAS: Single-Cloud Secure Middlewares

3.1 Introduction

Cloud computing is becoming increasingly popular as utility computing is being gradually realized, but many organizations still cannot enjoy the advantages of public clouds due to security concerns, legacy infrastructure, and high performance requirement (especially low latency). Many researchers tried to secure public clouds, but few studied the unique security problems of hybrid clouds. Kurma is our proposed hybrid cloud solution to the storage aspect of these problems.

In Chapter 2, we decided to use NFSv4.1 as Kurma's storage protocol because of NFSv4.1's advanced features such as sessions, delegations, and compound procedures. We discuss an early prototype of Kurma in this chapter. We name the prototype *SeMiNAS*—Secure Middlewares for cloud-backed Network Attached Storage. SeMiNAS is the first step towards our ultimate development of Kurma; it has the same threat model, an analogous architecture, and similar design goals. Both SeMiNAS and Kurma appear as NFS service providers to clients, and include on-premises proxies for security enhancement and performance improvement. The high-level architectures of SeMiNAS and Kurma are similar: they differ mostly in their proxy architecture and components. SeMiNAS provides the same caching and security features including confidentiality, integrity, and malware detection. SeMiNAS's discussions of complex interactions among those security and caching features is also applicable to Kurma. However, SeMiNAS is limited in several aspects and makes several simplifying assumptions. For example, SeMiNAS uses a single public cloud as back-end and is not secure against replay attacks; and it requires new NFS features not standardized yet or available from cloud providers. We propose our solutions to those limitations later in Chapter 4.

SeMiNAS consists of on-premises proxies that allow clients to outsource data securely to clouds using the same file system API as traditional NAS appliances. As shown in Figure 3.1, SeMiNAS inherits many advantages from the popular middleware architecture, as exemplified by network firewalls. For instance, SeMiNAS minimizes migration cost and can be deployed with only minimal changes to existing clients and servers. Managed by trusted security personnel, SeMiNAS can protect data from not only untrusted cloud servers but also semi-trusted clients, which might have been accidentally compromised by virus infection. Because accesses to cloud storage

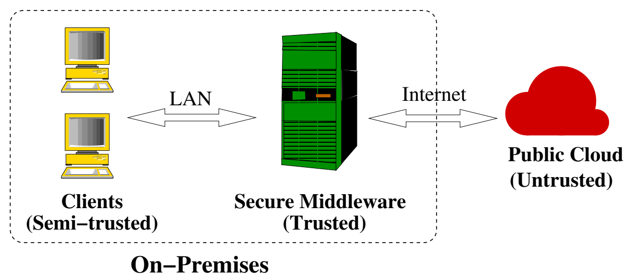


Figure 3.1: SeMiNAS high level architecture

servers incur long latency over WANs, SeMiNAS also caches cloud data to improve performance. SeMiNAS is deployed in the same local-area network with clients and thus can significantly reduce access latency to the cloud.

SeMiNAS provides end-to-end data integrity and confidentiality using authenticated encryption before sending data to cloud. Data stays in encrypted form in the cloud, and is not decrypted until SeMiNAS retrieves the data from clouds on clients' behalf. End-to-end integrity and confidentiality protects data from not only potential attacks during data transmission over the Internet but also misbehaving cloud servers and storage devices. Using a robust cryptographic scheme for key exchange, SeMiNAS can share files securely among multiple geographically distributed proxies. SeMiNAS also performs anti-virus scanning to catch infected clients and to stop the spread of viruses.

SeMiNAS reduces performance overhead using three mechanisms. First, it uses an authenticated encryption scheme that embeds Message Authentication Code (MAC) and other security meta-data into the already existing Data Integrity Field (DIF) of modern storage devices [32]. Thus, SeMiNAS minimizes proxy-to-cloud communication, which goes through the Internet and is usually the link with the highest latency. Second, SeMiNAS facilitates flexible trade-off between performance and security by allowing each security feature (integrity, encryption, and anti-virus) to be enabled and configured separately according to security policies. Some files (e.g., public binary executables), need only integrity and anti-virus, whereas some other files (e.g., media files), need only integrity and encryption. Third, SeMiNAS contains a persistent write-back cache that stores recently used data and coalesces writes to server. This reduces the communication to remote servers and allows many operations to be handled locally. Because interactions between the caching and security modules have significant and complex effects, SeMiNAS carefully integrates caching with the security modules to pursue the right balance between security and performance.

We implemented and evaluated SeMiNAS to ensure it achieves its design goals. To study the trade-off between security and performance, we measured SeMiNAS' performance and security overhead under different security policies. Adding integrity to caching, SeMiNAS introduces an overhead of 9–43% for micro-workloads; when caching, integrity, confidentiality, and anti-virus are all enabled, SeMiNAS introduces a moderate overhead of 5–55% for macro-workloads.

This chapter makes three contributions: **(1)** a security middleware system that allows NAS clients to use cloud storage as back-end in a secure, seamless, and flexible manner; **(2)** a study of the trade-off between performance and security under different security policies and workloads; and **(3)** insights into complex interactions between caching and security features, including integrity, confidentiality, and anti-virus.

The rest of this chapter is organized as follows. Section 3.2 presents the motivation behind SeMiNAS. Section 3.3 details SeMiNAS' design including its threat model, design goals, architecture, and security and caching features. Section 3.4 describes the implementation of our SeMiNAS prototype. Section 3.5 evaluates the performances of SeMiNAS under different security policies and workloads. Section 3.6 discusses related work and Section 3.7 concludes.

3.2 Background and Motivation

We present the background and motivation behind SeMiNAS by asking three questions: (1) Why a security middleware such as SeMiNAS is needed in cloud environments? (2) Why flexible trade-off between security and performance is important? (3) Why a file system API to the cloud is preferable to a RESTful API?

3.2.1 A secure middleware for the cloud

Security concerns about outsourcing data to third-party cloud providers are well warranted for three reasons. First, the surface area of exploitation and vulnerability increases significantly with public communication channel. Second, storage devices are managed by cloud providers in a *cloudy* (opaque) manner which begets uncertainty and mistrust. Third, even if the providers themselves are trustworthy, other cloud tenants, with whom customers may share physical resources, may be malicious.

Besides security concerns, performance and legacy infrastructure are other factors impeding the adoption of public cloud. A solution to this problem is the hybrid cloud model, where a portion of computing and storage goes to public cloud while the rest remains on-premises (private cloud) for stronger security and better performance. SeMiNAS fits this hybrid cloud model and is an on-premises file system proxy that protects data integrity and confidentiality from the public cloud.

Despite being shielded from many outside attacks, on-premises clients are still susceptible to other security threats, with malware being most notorious threat [103]. Moreover, there are usually a large number of clients making it difficult to secure and trust all of them. A consolidated secure proxy is convenient to enforce security policies at both untrusted ends.

3.2.2 Security vs. performance

A key incentive for cloud storage is its low total cost of ownership, but extra measures to secure cloud data do not come for free. Some researchers [24] argue that encryption is too expensive to justify storing encrypted data on the cloud, whereas others [1, 132] claim new hardware acceleration makes encryption viable and cheap for cloud storage. These debates highlight the importance of reducing performance overhead when securing cloud storage.

While performance can frequently be quantified by throughput or latency, measuring security is more difficult and often depends on the threat model and the workload. One widely used indicator of the security level of a system is the set of security features supported (e.g., integrity, confidentiality, authentication, anti-virus, etc.). For example, HTTP has no security feature, whereas HTTPS provides authentication, integrity, and confidentiality. More subsets of security features enable more flexible trade-off between security and performance. Providing only integrity, HTTPI [25,

113] is enough for many applications (e.g., video streaming) and almost as fast as HTTP. To let applications find the right balance between security and performance, SeMiNAS allows security features to be used separately.

3.2.3 File System APIs vs. RESTful APIs

Since Network-Attached Storage (NAS) was the most popular enterprise storage solution [123] before the cloud era, SeMiNAS provides a file system API to clients so that legacy applications based on NAS continue to work unchanged. This minimizes the migration cost while enjoying the advantages of cloud storage. SeMiNAS uses NFS, and supports both NFSv3 and NFSv4, offering compatibility with old applications and new features of NFSv4.

SeMiNAS also uses NFS to talk to cloud servers although many cloud providers currently offer only RESTful GET/PUT APIs [6]. Compared to vendor-specific RESTful APIs, it is more convenient to use a pervasive, open, and standard protocol to seamlessly talk to private and public clouds. As more applications are deployed in clouds, rudimentary RESTful APIs begin to fall short of functionalities to support complex systems [95]. In contrast, file system APIs, with much richer semantics, can significantly simplify application development and provide advanced optimization opportunities such as pNFS [107] and server-side copying [120]. Using a file system API is a growing trend as seen by the recent cloud offering of the NFSv4-based Amazon Elastic File System (EFS) [60].

File system APIs also offer stronger consistency guarantees than RESTful APIs do. Many early cloud storage systems promise only eventual consistency [28], which offloads tough questions such as "what if the loaded data are woefully out-dated" and "what if we read a totally different object that has been deleted and re-created" onto cloud application developers. One big challenge of weak consistency is conflicting changes, the reconciliation of which is difficult and often has to resort to human intervention. As technology advances with innovations, increasingly more cloud storage systems begin to provide strong consistency [19, 26, 126]. These systems make file system cloud APIs not only feasible but also more desirable than RESTful APIs.

Still, the performance of NFS over a WAN will be "bounded by the speed of light" [130] if each file operation incurs multiple round trips in the WAN, but a caching proxy can considerably accelerate performance as was demonstrated in both academia [74] and industry [104]. The performance acceleration of an NFS caching proxy can be particularly significant with the help of NFSv4 delegations—a client caching mechanism that enables local file operations without communication to remote NFS servers. We have showed in Chapter 2 that delegations can reduce the number of NFS messages by almost 30×. Delegations do not compromise NFS's close-to-open consistency [107]; they are effective as long as concurrent and conflicting file sharing among clients is rare, which is often true [69].

3.3 SeMiNAS Design

We present the design of SeMiNAS including its threat model, design goals, architecture, caching, and security features.

3.3.1 Threat Model

Our threat model reflects the settings of an enterprise office or an academic laboratory where clients access cloud data via a NAS proxy (see Figure 3.1). We discuss the trustworthiness of the cloud, clients, and the proxy with regard to security properties such as availability and integrity.

The Cloud. We do not trust the cloud in terms of confidentiality and integrity. It is risky to put any sensible data in plaintext format considering threats both inside and outside the cloud [8]. Since communication to public clouds goes through the Internet, plaintext data is vulnerable to man-in-the-middle attacks. Even if the communication is protected by encryption, storing plaintext data on cloud servers is still dangerous because the storage device may be shared with other malicious tenants. The same is true for data integrity: attackers inside and outside the cloud may covertly tamper with the data. However, we think cloud availability is a smaller concern. High availability is an important trait that makes cloud attractive: major cloud services had availability higher than four nines (99.99%) [138]. SeMiNAS assumes the cloud provider is always available.

Clients. Clients are semi-trusted. Clients are usually operated by employees of the organization, and are generally trustworthy if proper access control is enforced. SeMiNAS supports NFSv4 and thus can enforce access control using traditional mode bits and advanced ACLs [107]. However, clients are not fully trusted because they may be infected by malware and be compromised by intrusions [103]. This requires SeMiNAS to scan the data written by clients in order to detect infected clients.

The Middleware. The proxies of SeMiNAS are fully trusted. SeMiNAS is the source of trust and provides centralized and consolidated security services. Physically, the proxy is a small cluster of computers and appliances, which can fit in a guarded and monitored machine room. Thus, securing the proxy is easier than securing all clients that might scatter over multiple buildings. An organization can also dedicate experienced security personnel to administer the middleware, making it more resilient to security threats. We also trust that only SeMiNAS proxies can authenticate themselves to the cloud NFS servers using RPCSEC_GSS [35]; therefore, adversaries cannot fake a proxy.

3.3.2 Design Goals

We designed SeMiNAS to achieve the following four goals, ordered by descending importance:

- **High security**: SeMiNAS should be secure against attacks in its threat model. It should ensure integrity and confidentiality to data stored in the cloud, and be immune to malware from clients.
- **Low overhead**: For each configuration, SeMiNAS should minimize the performance impact of the security features by using a low-overhead security scheme and effectively caching data.
- **Flexibility**: SeMiNAS should be configurable to enforce a wide range of security policies. Security can hence be traded for performance according to security policies.

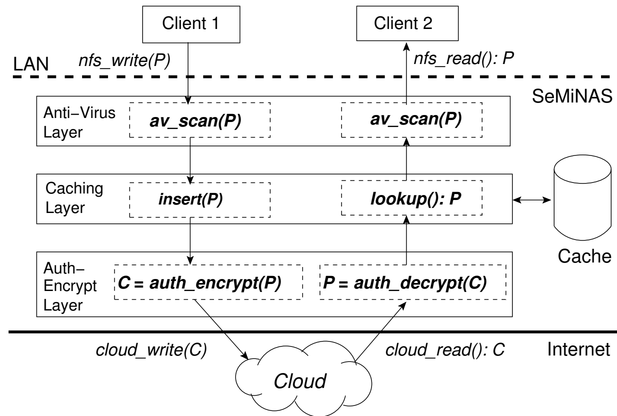


Figure 3.2: SeMiNAS layered architecture

- **Simplicity**: SeMiNAS should have a simple architecture that eases development and maintenance.

3.3.3 Architecture

Figure 3.2 shows the architecture of SeMiNAS, where we use a stackable file system architecture to achieve the design goals. Stackable file systems, such as Linux's UnionFS [139] and OverlayFS [18], are a popular technique to add new functionalities to existing file systems. Stackable file systems are flexible for three reasons: (1) they can intercept all file operations including `ioctls`; (2) they can be stacked on top of any other file systems (e.g., `ext4` and NFS); and (3) the stacking can happen in any order to achieve a wide range of functionalities. Stackable file systems are simpler than standalone file systems because they can use existing unmodified file systems as building blocks. Stackable file systems can also achieve high security as shown in previous studies [55, 62, 85, 140].

A SeMiNAS proxy acts as an NFS server to clients, and as a client to remote cloud servers. Internally, SeMiNAS consists of multiple stackable file system layers. They all sit between the upper NFS layer and the lower cloud layer. The upper NFS layer handles NFS requests from clients; the lower cloud layer performs file system operations to cloud servers. A client writes a file by first sending an NFS request to SeMiNAS. SeMiNAS scans the file (for malware detection), then simultaneously encrypts and authenticates the data to generate ciphertext and Message Authentication Codes (MACs). After that, SeMiNAS sends the ciphertext and MACs to the cloud. File reading happens in reverse. SeMiNAS simultaneously authenticates and decrypts the data from the ciphertext and MACs, but scans the data again only if the database of the anti-virus engine was updated.

In addition to the security layers, SeMiNAS also contains a persistent caching layer. Caching is indispensable to reduce the performance overhead so that security measures are practical when using a cloud back-end. However, the interactions between caching and the security layers are complex and have significant performance ramifications. Putting the caching layer between the anti-virus and authenticated-encryption layer is the result of a trade-off study, which we detail next.

3.3.3.1 Mixing Caching and Security Layers

Mixing the caching and the security layers is more complex than the simplest alternative, which is to put the caching layer all the way below or above all security layers. But both ordering turned out to be suboptimal in terms of security and performance. Stacking the caching layer below the security layers (i.e., closer to the cloud) means that data is cached in ciphertext format. Each cache hit incurs extra decryption and authentication, adding substantial performance overheads considering that intense computation is added to the fast path. Nevertheless, this ordering has the benefit of protecting accidental data corruption (because of, e.g., device driver bug or disk corruptions) that may happen in the persistent cache.

The biggest problem of putting caching above the security layers (i.e., closer to clients) is that the cache may contain virus-infected data. When the anti-virus layer is below cache, viruses cannot be caught until dirty cached data is written back; other clients may read the dirty cache and get infected. Using a write-through cache instead of a write-back cache alleviates this problem but at the expense of worse performance. Still, a write-back cache does not solve the problem, and reading infected data is still possible from the dirty page cache in RAM.

Stacking the caching layer above security layers also suffers from bad performance, especially for anti-virus scanning. When scanning a small write of a large file, the anti-virus engine may need to read other parts of the file and introduces extra I/O to the cloud. Even if the needed data is cached, the security layers cannot access the cached data without violating the layered architecture. Yet another problem is the difficulty of re-checking cached files after the malware database is updated. One plausible solution is to invalidate the cache upon each update, but invalidating cache is expensive as malware databases are updated frequently, nowadays hourly or even faster.

Conversely, "sandwiching" the caching layer in the middle (see Figure 3.2) solves the aforementioned problems. Putting the anti-virus layer closer to clients, and the authenticated-encryption layer closer to the cloud, also meets with our threat model: authentication and encryption protect integrity and confidentiality from the cloud; and anti-virus prevents malware from clients.

Before arriving at the solution seen in Figure 3.2, we constructed detailed flow charts of several possible permutations that mix security and caching layers, and analyzed each thoroughly. The detailed analysis and discussion of those alternatives are presented in a technical report [93].

3.3.4 Integrity and Confidentiality

As shown in Figure 3.2, SeMiNAS uses an authenticated-encryption file system layer to simultaneously provide data integrity, confidentiality, and authenticity [135]. Using one layer for multiple security features slightly complicates SeMiNAS's layered architecture. However, it is desirable for strong security because combining an encryption layer and an authentication may be susceptible to security flaws. There are three ways to combine encryption and authentication: (1) Authenticate then Encrypt (AtE) as used in SSL; (2) Encrypt then Authenticate (EtA) as used in IPSec; and (3) Encrypt and Authenticate (E&A) as used SSH. Despite being used by popular security protocols (SSL and SSH), both AtE and E&A turned out to be "not generically secure" [66]. Only one out of the three combinations (i.e., EtA) is considered to be secure [78]. The security ramifications of these combinations are rather complex [14]: even experts can make mistakes [67]. Therefore, SeMiNAS avoids separating encryption and authentication, and instead uses one of the standard authenticated encryption schemes that perform both operations simultaneously.

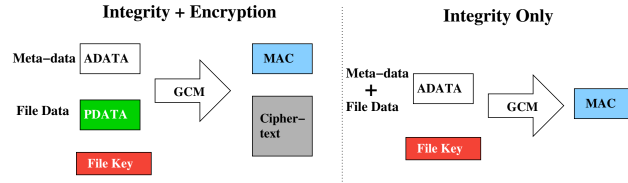


Figure 3.3: GCM for integrity and optional encryption

Out of the ISO-standardized authenticated encryption modes, we chose the Galois/Counter Mode (GCM) because of its superior performance [27] to other modes such as CCM [133] and EAX [13]. As shown in Figure 3.3, GCM accepts three inputs and produces two outputs. The three inputs are the plaintext to be both authenticated and encrypted (PDATA), additional data only to be authenticated (ADATA), and a key; the two outputs are ciphertext and a Message Authentication Code (MAC). Out of the three inputs, either PDATA or ADATA can be absent. This lets SeMiNAS achieve integrity but not encryption by leaving PDATA empty and using the concatenation of data and meta-data as ADATA.

On write operations, GCM uses the data to be written as PDATA and additional security meta-data (discussed in Section 3.3.4.1) as ADATA. GCM outputs the ciphertext and MAC, which are then written to the cloud. On read operations, SeMiNAS retrieves the ciphertext and MAC, and then simultaneously verifies the MAC and decrypts the ciphertext. SeMiNAS thus achieves end-to-end data integrity and confidentiality as the protection covers both the transport channel and the cloud storage stack.

3.3.4.1 Meta-Data Management

SeMiNAS divides a file into fix-sized data blocks (e.g., 16KB) and applies GCM to each block (with padding if necessary). SeMiNAS maintains per-file and per-block meta-data to provide security while enabling file sharing among multiple SeMiNAS instances deployed at multiple sites of an organization. As shown in Figure 3.4, the most important per-file meta-data is the encrypted key pairs which we discuss in Section 3.3.4.2; others are authenticated and encrypted file attributes including real file size, unique file ID, flags. The per-block meta-data includes the 16-byte-long MAC used for authentication, and an 8-byte-long block offset used for defending the attack of swapping blocks. SeMiNAS can also detect inter-file swapping of blocks because each file has a unique key.

Storing the per-file meta-data is simple, and SeMiNAS uses a file header for that. However, storing the per-block meta-data is more complex. One method is to write the concatenation of each encrypted block and its MAC as one file in the cloud (referred as Cloud Block File, *CBF*, thereafter). This method not only burdens file system meta-data management with many small files [50], but it also negates the benefits of using a file system API, including file-level strong consistency provided by file locking. Although locking a file is still possible by locking all underlying CBFs in a deadlock-free order (e.g., increasingly by block offset), it is complex and infeasible for large files considering the limit of open files and the large number of round trips across the Internet.

Another method to store the per-block meta-data is to use an extra cloud file for all per-block meta-data of each file. However, this is suboptimal, especially considering the high latency of cloud accesses, because writing one block incurs two NFS requests to the cloud. With compound

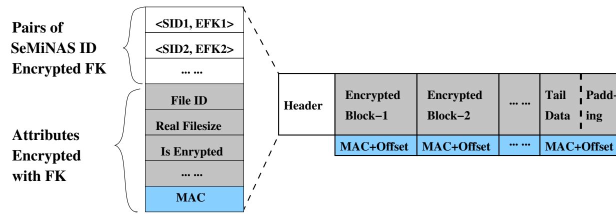


Figure 3.4: SeMiNAS meta-data management

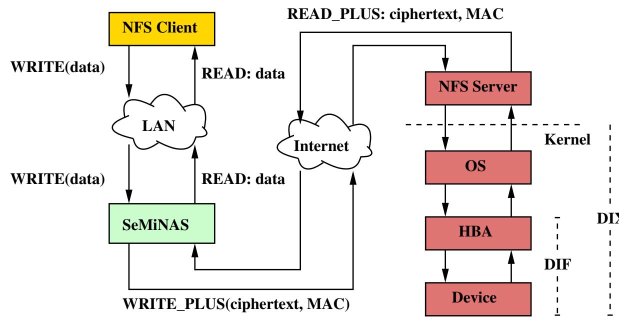


Figure 3.5: NFS end-to-end data integrity using DIX

operations [107], NFSv4 has the potential to avoid this problem by combining the two requests into one compound. Unfortunately, compounding is not effective in practice [21].

Yet another method is to map one block to a larger block in the cloud file. For example, a file with ten 16KB blocks corresponds to a cloud file with ten slightly larger blocks (i.e., 16KB+N where N is the size of the per-block meta-data). This method is free from the problems of the other two, but its performance still suffers from extra read-modify-update operations caused by breaking block alignment. Using larger block sizes (e.g., 256KB instead of 16KB) alleviates this problem by having fewer extra read-modify-update operations, but it has the side effect of making each extra operation more expensive.

SeMiNAS uses a better alternative that does not have any of the aforementioned problems. It leverages the Data Integrity eXtensions (DIX) [31, 32], a trend of making the once-hidden Data Integrity Fields (DIF) of direct access storage devices available to applications (Figure 3.5). DIX effectively extends the disk sector size from 512 bytes to 520 bytes by adding eight DIF bytes. The DIF bytes have long existed inside disks, but were used only internally for checksums. Flash devices also have similar checksum bytes in the out-of-band storage. However, internal checksums cannot protect the software storage stack, which is becoming deeper, more complex, and thus more error-prone [10, 73], especially due to virtualization. Extending DIF to protect the upper layers of the storage stack is a promising solution.

With DIX, OSes are now able to use these protection bytes [16], and SeMiNAS implements an existing proposal to extend DIX further to NFSv4 [92]. SeMiNAS saves the per-block meta-data to the DIF bytes of the block. To accommodate the 24 bytes per-block meta-data, we require the block to be at least 2KB large, because each sector uses at least two DIF bytes by itself for an internal checksum and provides at most six DIF bytes for applications.

3.3.4.2 Key Management

Key management is critical to achieve strong security. SeMiNAS uses a master key pair (MK) per SeMiNAS instance (usually in the form of a NAS appliance), and a per-file symmetric encryption key (FK). MK is used for asymmetric encryption (RSA), and consists of a public key (MuK) and a private key (MrK). The public keys are exchanged among all instances manually by security personnel. This is feasible because one geographic site usually has only one SeMiNAS instance, and key exchange is only needed when opening an office in a new site. This scheme has the advantage of not relying on any third-party for public key distribution. Each file also stores a randomly generated initialization vector (IV) in its header. For each block, SeMiNAS added this IV with the block offset to generate a unique IV.

Because each SeMiNAS instance maintains the MuKs of all other instances, the file keys (FKs) can be shared among all SeMiNAS instances under the protection of MuKs. When creating a file, a SeMiNAS instance (*creator*) generates an FK. Then for each SeMiNAS instance with which the creator is sharing the file (*accessor*), the creator encrypts the FK using the accessor's public key (MuK) with the RSA algorithm, and generates a ⟨SID, EFK⟩ pair where SID is the unique ID of the accessor and EFK is the encrypted FK. All the ⟨SID, EFK⟩ pairs are then stored in the file header. With the upcoming sparse file support [120], the file header can reserve sufficiently large space with a hole following the header. Therefore, adding a new SeMiNAS instance need only add its ⟨SID, EFK⟩ in the header by filling the hole without shifting the file data. When opening a file, a SeMiNAS instance, which needs to be an accessor of the file, first finds its ⟨SID, EFK⟩ pair in the header, and then it retrieves the file key FK by decrypting the EFK using its private key (MrK).

3.3.5 Malware Detection

SeMiNAS uses an anti-virus file system layer for malware detection. Malware files can be identified in two ways. One is to search malware patterns in file contents; the other way is to match the hash values (e.g., MD5) of files against the hash values of known malware. Searching patterns has lower overhead than matching hash values because only the new content (and its adjacent old content in case of multi-part patterns) have to be scanned upon writes [85], instead of the whole file. However, partial file scanning is not secure because the vast majority of malware are identified by hash values. For example, hash-based signatures account for more than 99% of all the signatures of the open source ClamAV [65] malware database. Thus, SeMiNAS scans whole files, and detects malware using both patterns and hash values.

Because SeMiNAS uses file hash values for malware detection, it needs to compute the hash values every time files are changed. This is expensive because the whole file has to be read again. One optimization is to defer the scanning until file close so that a sequence of writes following one file open incur only one scanning instead of multiple scanning. If the OS flushes the infected dirty data before file close, this optimization may allow the malware sneak in before catching it. However, this is not a problem for SeMiNAS because the write-back cache does not write dirty data back to cloud until file close. Unfortunately, other clients in the same SeMiNAS instance may still read the infected dirty data from the memory or the local cache. To avoid this, SeMiNAS immediately scans a dirty file and makes sure the file is clean before letting any read operation proceed.

Scanning only small files for malware detection is a common practice. For example, Google

Drive only scans files smaller than 25MB, and almost all online malware detection services set file size limits from 1–40MB [109]. Therefore, SeMiNAS uses a configurable parameter as the file size threshold, and only scans files smaller than the threshold. When a file is found to be infected, SeMiNAS, depending on the security policy configuration, either returns an error to the client and purges the cache, or returns success to the client and secretly quarantines the infected file as forensic evidence without really writing it to the cloud.

Because SeMiNAS ensures file integrity, the cloud could not inject malware without being caught. Therefore, anti-virus scanning is normally not needed when reading files. However, it becomes necessary when the malware database is updated. Otherwise, new malware that may have snuck in before the malware database was updated may spread further (e.g., "zero-day" malware). SeMiNAS stores the database version of the latest scanning in the file header, and incrementally re-scans the file on reads as well if the database is out-of-date.

3.3.6 Caching

SeMiNAS's caching file system layer maintains a cache of recently used file data blocks, so that hot data can be read in the low-latency on-premises network without communicating with the cloud. The caching layer is designed to be a write-back cache, to minimize writes to the cloud as well. Being write-back, the cache is persistent because some NFS requests—WRITEs with the stable flag, and COMMITs—require dirty data be flushed to "stable storage" [110] before replying. Because the NFS protocol demands stable writes to survive server crashes, the cache layer also maintains additional meta-data in stable storage to ensures correct crash recovery. The meta-data includes a list of dirty files and a per-block dirty flag to distinguish dirty blocks from clean blocks.

To maintain NFS's close-to-open semantics, Kurma revalidates the cache when opening a file. To tell whether the cache is still valid, Kurma compares the timestamp of the cached content with the timestamp of the remote file. If Kurma detects the cache content have changed, it invalidates the cache. Kurma also flushes dirty cache of a file back to the cloud NFS server when closing the file.

For each cached file, SeMiNAS maintains a sparse file of the same size in the proxy's local file system. Insertion of file blocks are performed by writing to the corresponding blocks of the sparse files. Evictions are done by punching holes at the corresponding locations using Linux's `fallocate` system call. This design delegates file block management to the local file system, and thus significantly simplifies the caching layer. SeMiNAS also stores the crash recovery meta-data of each file in a local file. The caching layer does not explicitly keep hot data blocks in memory, but implicitly does so by relying on the OS's page cache.

When holding a write delegation of a file, a SeMiNAS instance does not have to write cached dirty blocks of the file back to the cloud until the delegation is recalled. Without a write delegation, SeMiNAS has to write dirty data backs to the cloud upon file close to maintain NFS's close-to-open consistency. To avoid bursty I/O requests and long latency upon delegation recall or file close, SeMiNAS also allows dirty data to be written back periodically at a configurable interval.

3.4 Implementation

We have implemented a prototype of SeMiNAS in C and C++ on Linux. We have tested our implementation thoroughly using functional unit tests and ensured our prototype passed all `xfstests` [141] cases that are applicable to NFS. We present the technical background and the implementation of the security and caching features.

3.4.1 NFS-Ganesha

Our prototype is based on NFS-Ganesha [29,30,49], an open-source user-land NFS server that supports NFS v3, v4, and v4.1. NFS-Ganesha provides a generic interface to file system implementations with a *File System Abstraction Layer* (FSAL), which is similar to a Virtual File System (VFS) in Linux. With different FSAL implementations, NFS-Ganesha can provide NFS services to clients using different back-ends such as local and distributed file systems. NFS-Ganesha's FSAL implementations include FSAL_VFS that uses a local file system as back-end, and FSAL_PROXY that uses another NFS server as back-end. We use FSAL_VFS for the cloud NFS server, and FSAL_PROXY for our secure proxy.

Like their stackable counterparts in Linux [144], FSALs can also be stacked to add features in a modular manner. For example, an FSAL for encryption can be stacked on top of FSAL_PROXY. NFS-Ganesha originally allowed only one stackable layer; we added the support of multiple stackable layers. We have also improved FSAL_PROXY by fixing bugs and optimizing performance. NFS-Ganesha configures each exported directory and its backing FSAL separately in a configuration file, allowing SeMiNAS to specify security policies for each exported directory.

3.4.2 Authenticated Encryption

We implemented SeMiNAS's authenticated encryption in an auth-encrypt FSAL layer. We used `cryptopp` as our cryptographic library because it supports a wide range of cryptographic schemes such as AES, GCM, and VMAC [125]. We used AES as the symmetric key cryptographic block cipher for GCM. We implemented the NFS end-to-end data integrity extension [92] in NFS-Ganesha so that ciphertext and the security meta-data (MAC, etc.) can be transmitted together between the proxy and the cloud. First, we implemented the READ_PLUS and WRITE_PLUS operations of NFSv4.2 [120] in NFS-Ganesha (which does not support NFSv4.2 as of this writing) because the extension [92] are based on these two new operations. Then, at the proxy side, we changed FSAL_PROXY to use these two operations for communications with the cloud NFS server (Figure 3.5). A WRITE_PLUS (READ_PLUS) operation writes (reads) the ciphertext and security meta-data in one request without extra round trips over the Internet. Lastly, at the cloud side (running NFS-Ganesha FSAL_VFS), we changed FSAL_VFS to use WRITE_PLUS and READ_PLUS, and to write the ciphertext and security meta-data together to storage devices. Currently, Linux does not have system calls to pass file data and their DIF bytes from user space to kernel; so we used a DIX kernel patchset [97] after we fixed a few bugs.

We implemented two performance optimizations in the auth-encrypt FSAL. First, we cache the file key (FK) and the $\langle$SID, EFK$\rangle$ pairs in memory to reduce the frequency of expensive RSA decryption of FKs. This is safe because the auth-encrypted FSAL runs in the trusted proxy. Second,

Language	**Files**	**Comment**	**Code**
C	14	841	3,209
C++	48	589	6,435
C/C++ Header	50	1,574	2,976
CMake	9	15	230
Total	**121**	**3,019**	**12,850**

Table 3.1: Lines of code of the SeMiNAS prototype

we use the faster ($3.2\times$ on our testbed) VMAC [27, 125] instead of GCM when only integrity (but not encryption) was required.

3.4.3 Malware Detection

We implemented SeMiNAS's malware detection using ClamAV [65]. Vanilla ClamAV accepts a file path or a file descriptor as input and scans the entire file. Considering the common practice of scanning only small files for malware [109], we modified ClamAV and added a scanning function to accept a memory buffer, which contains the data of a whole file. This function has the advantage of not having to read a whole file again when the file (or a part of it) is already loaded in memory (e.g., when a file is being overwritten). As an optimization discussed in Section 3.3.5, malware detection can be deferred to file close, when the dirty file is flushed to the cloud. However, a dirty file can also be flushed periodically before file close in order to avoid bursty I/O requests. This required us to scan the dirty buffer periodically as well; otherwise, malware files could sneak into the cloud. Therefore, SeMiNAS performs malware detection before write-back no matter if it is before or at file close. When a malware database is updated, one typically has to rescan all files before reading them (part of our future work).

3.4.4 Caching

Because the caching layer needs to manage write-back threads, we implement the caching layer as an outside library to avoid complicating NFS-Ganesha's threading model. The caching library provides caching (lookup, insert, invalidate, etc.) and write-back APIs for NFS-Ganesha. When inserting dirty blocks of a file using the library, SeMiNAS registers a write-back callback function along with the dirty buffer. The callbacks are invoked periodically as long as the file remains dirty. When closing a file, SeMiNAS calls the write-back function directly, and deregisters the callback.

3.4.5 Lines of Code

The implementation took more than 4000 man-hours. Table 3.1 shows the lines of code of our prototype excluding existing NFS-Ganesha code. In addition, we have fixed bugs and added the multi-layer stacking feature in NFS-Ganesha; our patches have been merged into the mainstream NFS-Ganesha. We plan to release all code as open source in the near future.

3.5 Evaluation

We now present the evaluation of SeMiNAS under different workloads, security polices, and network settings.

3.5.1 Experimental Setup

Our testbed consists of seven identical Dell PowerEdge R710 machines running CentOS 7.0 with a 3.14 Linux kernel. Each machine has a six-core Intel Xeon X5650 CPU, 64GB of RAM, a Broadcom 1GbE NIC, and an Intel 10GbE NIC. Five machines ran as NFS clients, one as a secure proxy, and one as a cloud NFS server. Clients communicated to the proxy using the 10GbE NIC, whereas the proxy communicated to the server using the 1GbE NIC. The average RTT between the clients and the proxy is 0.2ms. The proxy runs our SeMiNAS prototype, and uses an Intel DC S3700 200GB SSD for the persistent cache. We emptied the cache before each experiment to observe the system's behavior when an initial empty cache is gradually filled. We used 4KB as the block size of SeMiNAS.

To emulate the connection between the proxy and the cloud, we injected 10–30ms delay in the outbound link of the server using `netem`; 10ms and 30ms are the average network latencies we measured from our machines to in-state data centers and the closest Amazon data center, respectively. We patched the server's kernel with the DIX support [97] (with our bug fixes) that allows DIF bytes to be passed from user space to kernel.

Physical storage devices that support DIX are still rare, so we had to set up a 10GB DIX-capable virtual SCSI block device backed by RAM using `targetcli` [91]. Using RAM, instead of a disk- or flash-backed loop device, allowed us to emulate the large storage bandwidth provided by distributed storage systems in the cloud. Although using RAM fails to account for the server-side storage latency, the effect is minor because the Internet latency (typically 10–100ms) usually dwarfs the storage latency (typically 1–10ms), especially considering the popularity of cloud in-memory caching systems such as RAMcloud [94] and Memcached [41]. If storage latency in the cloud was counted, the extra latency introduced by SeMiNAS's security features would actually be smaller relative to the overall latency; hence the results we report here are more conservative. The DIX-capable device was formatted with `ext4`, and exported by NFS-Ganesha using FSAL_VFS.

We verified that all SeMiNAS's security features work correctly. To test the correctness of integrity check, we created files on a client, changed different parts of the files on the cloud server, and verified that SeMiNAS detected all the changes. To test the correctness of encryption, we manually confirmed that file data was unreadable ciphertext when reading from the server, but was plaintext identical to what was written when reading from clients. We also verified that malware were detected by SeMiNAS when clients attempted to write malware files. Our malware detection experiments also showed that scanning malware files took about the same amount of time (less than 1% difference) as scanning equal-sized good files. As the absence of malware files had negligible influence on SeMiNAS's performance, we used only good files in our following experiments.

We benchmarked seven combinations of the caching and security features as listed in Table 3.2. P and C are baselines, and the remaining five correspond to common security policies. We benchmarked a set of synthetic micro-workloads with pre-configured read-write ratios, and Filebench [40] macro-workloads including File Server and Web Server. The micro-workloads help us understand SeMiNAS's behavior in a controlled environment; the macro-workloads reflect

Configs	Proxy	Integrity	Encryption	Caching	Anti-virus
P	✔	✘	✘	✘	✘
C	✔	✘	✘	✔	✘
I	✔	✔	✘	✘	✘
IE	✔	✔	✔	✘	✘
IC	✔	✔	✘	✔	✘
ICE	✔	✔	✔	✔	✘
ICEA	✔	✔	✔	✔	✔

Table 3.2: Combinations of security and caching features. The names of the configurations are combinations of the first letter of enabled features. For example, IC means the configuration when both Integrity and Caching are enabled.

its performance in popular and realistic scenarios.

3.5.2 Micro-Workloads

We evaluated SeMiNAS using micro-benchmark workloads that help us understand the performance impact of SeMiNAS's features. We pre-allocated 100 files for each of the five NFS clients, and then repeated the following operations for two minutes: randomly pick one file, open it with `O_SYNC`, perform n reads and m writes with a fixed I/O size at random offsets, and close it. We varied n and m to control the read-write ratio, the file size to control the anti-virus overhead, the I/O size to control the authenticated-encryption overhead, and the network latency between the proxy and the cloud to control the effectiveness of caching.

3.5.2.1 Security and Caching Features

SeMiNAS's security and caching features have different performance impact: caching generally helps performance, whereas integrity, encryption, and anti-virus hurt performance. The combined performance of these features depends heavily on workload characteristics, especially the read-write ratio. For example, a read-only workload is not influenced by anti-virus and benefits a lot from caching. Conversely, a write-heavy workload does not benefit much from caching, and also incurs frequent anti-virus scanning. We studied read-write ratios (n:m) from write-intensive 1:16 to read-intensive 16:1 to cover common ratios in real workloads [69, 106].

Figure 3.6(a) shows the results of 1:1 read-write ratio. Overall, the configurations with caching (i.e., C, IC and ICE) outperform their counterparts without caching (i.e., P, I, IE) by 2–3×. Adding caching to integrity (I→IC) improves throughput by more than 2×. The further addition of encryption does not reduce the throughput because encryption/decryption is relatively fast and reading cached (cleartext) data does not incur decryption at all. Therefore, the performance of ICE is almost the same as IC. Further, adding anti-virus to ICE has only negligible performance penalty because scanning a 1MB file is fast compared to the 30ms network latency.

Caching boosts performance even more for workloads with a higher read-write ratio of 16:1, as shown in Figure 3.6(b). The configurations with caching outperform their counterparts without

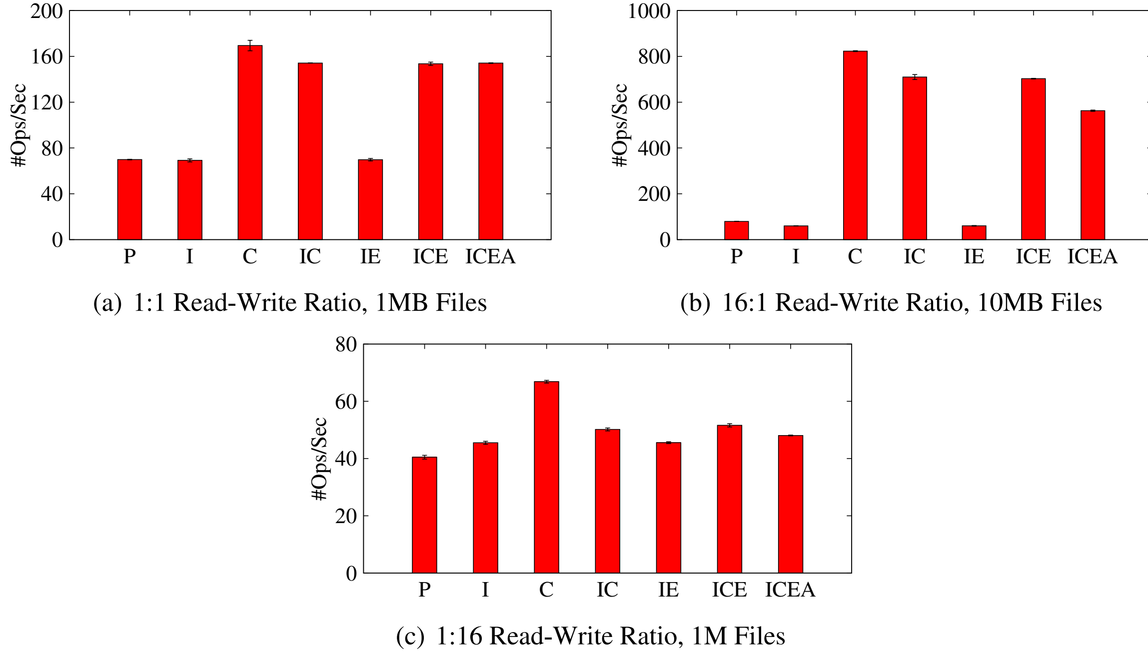


Figure 3.6: Throughput of different combinations of security and caching features with 4KB I/O size and 30ms network latency

caching by more than $10\times$. Figure 3.6(b) reinforces the observation that ICE's performance is almost the same as IC's. However, unlike in Figure 3.6(a), adding anti-virus to ICE incurs an performance penalty of 20% in Figure 3.6(b). This is because the anti-virus layer performs whole file scanning and the file size is larger (10MB vs. 1MB) in Figure 3.6(b).

However, the performance benefit of caching is only 10–65% with a write-intensive read-write ratio of 1:16 (Figure 3.6(c)). Caching is less helpful to writes because the dirty data, although cached, still has to be flushed back to the server when the file is closed. This is required by NFS's close-to-open consistency, which guarantees that when a client opens an NFS file, it can observe the changes made by all clients that have closed the file before. The small improvement of 10–65% in Figure 3.6(c) comes from caching read operations and coalescing multiple small writes to fewer larger writes.

3.5.2.2 Integrity Tests

Data integrity ensures that what we read from the cloud is indeed what we stored, and is the most important security feature of SeMiNAS. Without integrity, encryption only prevents sensitive data from leaking to public, but does not prevent the sensitive data from corruption or tampering; without integrity, anti-virus alone does not prevent the cloud from infecting files. Therefore, we evaluated integrity first, and we always enable integrity in subsequent evaluations of encryption and anti-virus.

The performance overhead of data integrity depending on whether the cache is enabled or not, so we studied both cases. Figure 3.7 shows the results without cache, under different read-write ratios and network latencies. Figure 3.7 shows that with a 30ms network latency and the write-intensive workloads, where $m > n$, the I configuration has higher throughput than P, despite

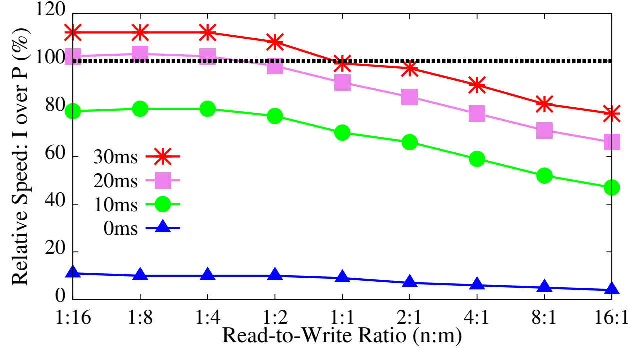


Figure 3.7: Relative speed of I over P with 4KB I/O size and 1MB file size. 100% (dashed line) is the baseline P.

the fact that adding integrity to proxy (P→I) incurs extra read operations for file keys. This is because SeMiNAS is more efficient than NFS-Ganesha's FSAL_PROXY for stable writes to the cloud. FSAL_PROXY uses one WRITE request followed by one COMMIT request to achieve one stable write operation, whereas SeMiNAS uses only one WRITE_PLUS request with the stable flag. For write-intensive workloads, the benefit of WRITE_PLUS is more than the overhead of reading file keys; the net effect of adding integrity (P→I) therefore boosts performance by up to 16%. However, as we increase the read-write ratio, the benefit of WRITE_PLUS diminishes, and thus the integrity overhead grows.

The performance overhead of integrity also depends on the network latency between the proxy and the cloud. In the extreme case of no (zero) latency, adding integrity (P→I) causes the performance to drop by more than 90%. This is caused by the expensive RSA decryption to recover the file key, which dominates in the I/O latency. However, as the network latency grows, the integrity overhead decreases because the RSA decryption latency no longer dominates. With a more realistic latency of 10–30ms, the throughput of integrity (I) is between 16% higher and 53% lower than that of proxy (P).

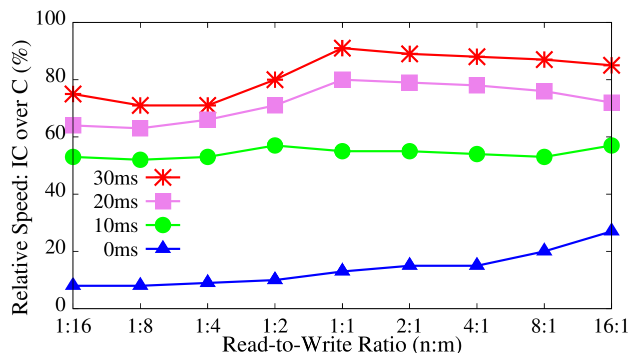


Figure 3.8: Relative speed of IC over C with 4KB I/O size and 1MB file size. 100% is the baseline with only caching (C).

Figure 3.8 shows the integrity performance overhead (C→IC) when the cache is present. The throughput of IC is always lower than C, unlike in Figure 3.7 where adding integrity sometimes improves performance due to the elimination of COMMITs by WRITE_PLUSs. With a write-back

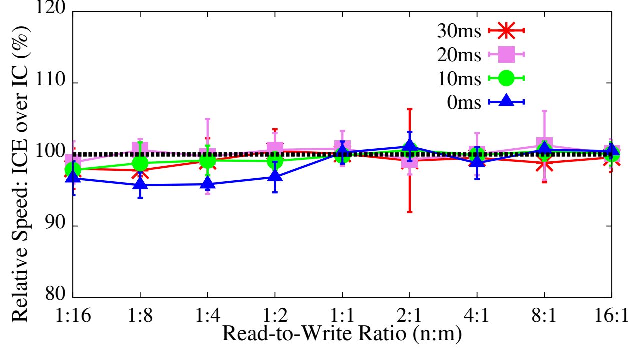


Figure 3.9: Relative speed of ICE over IC with 64KB I/O size and 1MB file size. 100% (dashed line) is the baseline with integrity and caching (IC). Note the Y-axis starts from 80%.

cache, C needs only one COMMIT after writing back all dirty data, rather than one COMMIT following each write. Thus, using WRITE_PLUS requests in IC saves only one COMMIT request regardless of the number of writes (i.e., m) during a period when a file is open. This also explains the initial rise of the curves in Figure 3.8: for the n reads and m writes after opening a file, the number of requests C sends to the cloud is $m + 1$ (i.e., m WRITEs and one COMMIT, and reads are cached); saving one COMMIT means saving $\frac{1}{(m+1)}$ of the requests. Consequently, as m decreases, the performance benefit of WRITE_PLUSs increases. In sum, the performance overhead of adding integrity to caching (C→IC) is between 9–43% with the 10–30ms network latency.

In all micro-workloads, the caching configurations always had higher absolute throughputs than their non-caching counterparts, regardless of the read-write ratio and the network latency. Therefore, we only describe configurations with caching in subsequent evaluations.

3.5.2.3 Encryption Tests

In our experiments of adding encryption to integrity (IC→ICE), we did not notice significant performance difference between IC and ICE. The speed of ICE is almost the same as the baseline IC configuration regardless of the I/O size (4KB–1MB), the read-write ratio (1:16–16:1), and the network delay (0–30ms). Further adding encryption does not incur significant overhead because symmetric encryption is fast relative to other SeMiNAS latencies. Other than the long latency between the proxy and the cloud, the slowest component of SeMiNAS is the RSA asymmetric decryption for retrieving of file keys, which are needed only once at file open. Adding file data encryption on top of integrity, however, does not incur extra RSA operations. Therefore, its performance overhead is negligible.

Only the extreme case of no (zero) network delay shows a noticeable difference (4%) under write-intensive workloads, but not under read-intensive workloads. Both reading from and writing to the cloud incur extra cryptographic operations. However, with caching, many read operations are fulfilled from the proxy cache. The caching layer is stacked above the encryption layer and stores data in cleartext form. Therefore, read operations that are cached do not incur decryption at all. In sum, adding encryption on top of integrity is almost free in SeMiNAS.

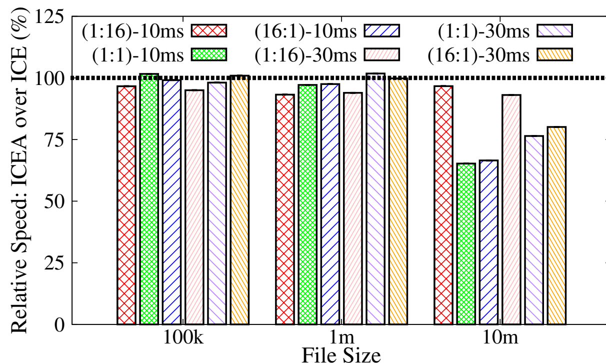


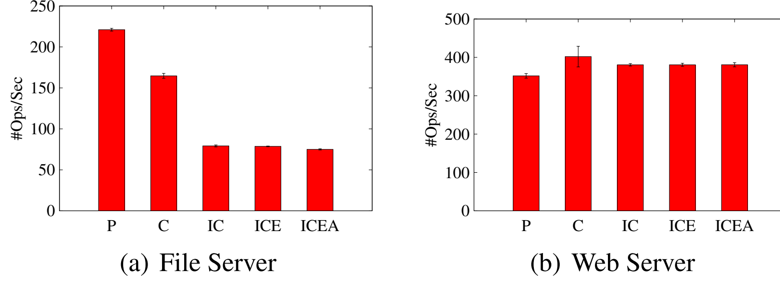
Figure 3.10: Relative speed of ICEA over ICE with 4KB I/O size under different file sizes, read-write (n:m) ratios, and network latencies. 100% (dashed line) is the baseline ICE.

3.5.2.4 Anti-virus Tests

We present the performance overhead of adding anti-virus on top of caching, integrity, and encryption (ICE→ICEA). We studied anti-virus's overhead under different file sizes, read-write ratios and network latencies because the relative speed of ICEA over ICE depends on all the three factors. Specifically, we show in Figure 3.10 the results of 100KB, 1MB, and 10MB files under read-write ratios of 1:16, 1:1, and 16:1, and network latencies of 10ms and 30ms. Compared with the ICE baseline, ICEA has almost the same throughput for 100KB and 1MB files regardless of read-write ratios and network latencies. This is because the time to scan small files is negligible in the presence of longer network latency between the proxy and the cloud.

For 10MB files (rightmost cluster of bins in Figure 3.10), adding anti-virus to ICE incurs a performance overhead of up to 35% because the scanning time is more substantial. Under 10ms network latency, the overhead is about the same and around 35% for read-write ratios of 1:1 and 16:1, but negligible for 1:16. Because reads are cached, for a read-write ratio of n:m, ICE's throughput is approximately $\frac{n+m}{cL+mL}$ where c is the constant number of requests for opening and closing a file, and L is the network latency between the proxy and the cloud. Although writes are also cached, m write requests are still needed upon write-back if the writes are not overlapping and thus not coalesced. Adding anti-virus, ICEA's throughput is $\frac{n+m}{cL+mL+S}$ where S is the anti-virus scanning time for one 10MB file. The performance overhead of anti-virus (ICE→ICEA) is thus $\frac{S}{cL+mL+S}$. This explains why the overhead is about the same for the 1:1 and 16:1 ratios as reads are cached and n is irrelevant. The formula also explains the small overhead of the 1:16 ratio: the denominator becomes much larger than the numerator when m is as large as 16. The case of 30ms latency is similar: the 1:1 and 16:1 ratios have lower (than the 1:16 ratio) and close performance overheads. The overheads of the 1:1 and 16:1 ratios are also lower than their 10ms-latency counterparts because a larger L makes the denominator larger and the end result smaller.

In sum, the overhead of adding anti-virus on top of ICE is negligible for 100K and 1MB files, and 2–35% for 10MB files. The anti-virus overhead (ICE→ICEA) is lower for more write-intensive workloads with longer network latencies.



(a) File Server

(b) Web Server

Figure 3.11: Filebench results with 30ms network latency

3.5.3 Macro-Workloads

We evaluated SeMiNAS using the Filebench File Server and Web Server workloads, which are popular macro-workloads where integrity, encryption, and anti-virus may be desired. Figure 3.11(a) shows the results of the File Server workload, which is metadata-intensive and has more writes than reads. Adding caching to proxy (P→C) causes a 26% performance drop for two reasons: (1) the proxy cache does not help reads because all reads are fulfilled by clients' page caches, which are pre-populated during Filebench's file pre-allocation stage; and (2) because this workload wraps each write with a pair of file open and close calls, the write-back cache cannot coalesce writes, and incurs extra "stable" writes for no benefit. Even when we configured the client's RAM (1GB) to be smaller than the working set size (1.28GB), most of reads were stilled cached and the results were similar. In Figure 3.11(a), adding integrity (C→IC) halves the throughput because of the overhead of maintaining file headers. Each write in this workload is actually an append and thus incurs two operations on the file header: one for retrieving the header from the cloud, and another for updating the file size in the header. However, further adding encryption and anti-virus does not incur significant performance overhead because the files are relatively small (128KB in average) in this workload. In sum, adding integrity, encryption, and anti-virus has a performance overhead of 66% and 54% with respect to P and C, respectively.

Figure 3.11(b) shows the results of the Web Server workload, which is read-intensive. Unlike the File Server workload, here, adding caching (P→C) improves performance—although only by 14% because most reads are also fulfilled by clients' page cache. Adding integrity to caching (C→IC) has an overhead of 5% because of file headers and RSA decryption; and further adding encryption and anti-virus does not add extra overheads because of small file sizes. In sum, the throughput of ICEA is 8% higher and 5% lower than P and C, respectively.

3.6 Related Work

As a secure NAS system, SeMiNAS is related to secure distributed storage systems, and storage-based intrusion detection systems. As a proxy using the cloud as back-end, SeMiNAS is related to cloud storage gateways and cloud NAS. We discuss each next.

3.6.1 Secure Distributed Storage Systems

SiRiUS [52] and Plutus [64] provide end-to-end file integrity and confidentiality with minimal trust on the server; but neither of them is secure against compromised clients. SeMiNAS' sharing of file keys (FK) is similar to SiRiUS [52]. However, because access control is enforced by the trusted proxy, SeMiNAS needs only one key per file instead of two (one for reading and the other for writing) in SiRiUS. NASD [51] and SNAD [84] add strong security to distributed storage systems using secure distributed disks. In both NASD's and SNAD's threat models, disks are trusted; these are fundamentally different from threat models in the cloud where storage hosts are physically inaccessible by clients and thus hard to be trusted.

3.6.2 Storage-Based Intrusion Detection Systems

Avfs [85] is an anti-virus stackable file system that scans file against pattern-based virus signatures. Avfs also use ClamAV (c. 2004), which has evolved considerably over the last decade and the original pattern based signatures are less than 1% of the entire database [65]. Most signatures are now based on hash values of file contents. FileWall [114] combines the idea of firewall with network file systems, and provides file access control based on both network context (e.g., IP address) and file system context (e.g., file owner). Molina et al. presented a study of detecting compromised clients by searching suspicious patterns in their file system activities [86]. All these studies differ from SeMiNAS because they focus only on clients, and do not consider security threats from storage servers or the cloud.

3.6.3 Cloud NAS

Panache [74] is a parallel file system cache that enables efficient global file access over a WAN without the fluctuations and latencies of WANs. It uses pNFS to read data from remote cloud servers over WANs and caches them locally in a cache cluster. Using NFS, Panache enjoys the strong consistency of file system API. However, its main focus is high performance with parallel caching, instead of security. Cloud NAS services are provided by companies such as Amazon [60], SoftNAS [115] and Zadara Storage [143]. These services focus on providing file system services in public clouds. As back-ends of SeMiNAS, cloud NAS service providers control and trust the ultimate storage devices, whereas SeMiNAS cannot control or trust the devices.

3.6.4 Cloud Storage Gateways

Using the cloud as back-end, a cloud gateway gives a SAN or NAS interface to local clients, and can provide features such as replication, security, and caching. There are several cloud gateway technologies, in both industry and academia. In academia, Hybris [33], BlueSky [132], and Iris [117] are examples of cloud storage gateway systems that provide integrity. Hybris additionally gives fault tolerance by using multiple cloud providers, whereas BlueSky also provides encryption. BlueSky and Iris have a file system interface on the client side, and Hybris provides a key-value store. However, none of them uses a file system API for cloud communication, and thus they offer only a weaker model—the eventual consistency model that usually uses a RESTful API. In the storage industry, NetApp SteelStore [89] is a cloud integrated storage for backup. Riverbed

SteelFusion [104] provides a hyper-converged infrastructure with WAN optimization, data consolidation, and cloud back-ends. The exact security mechanisms of SteelStore and SteelFusion are not publicly known although they claim to support encryption.

3.7 Conclusions

We presented the design, implementation, and evaluation of SeMiNAS, a secure NAS proxy using the cloud as back-end. SeMiNAS provides end-to-end data integrity and confidentiality using efficient authenticated encryption, which encrypts data and generates a MAC at the same time. SeMiNAS stores its cryptographic meta-data efficiently in the Data Integrity Field (DIF) of modern storage devices without incurring extra network round trips. In addition, SeMiNAS performs on-access malware detection to prevent malware from spreading across clients. SeMiNAS uses a persistent cache to alleviate the performance overhead of these security features. We designed and implemented these features in a layered architecture so that the features can be combined flexibly according to security policies. We carefully designed the interactions among these layers. We have evaluated the performance of SeMiNAS under different security policies to study the trade-off between security and performance. Our evaluation showed that adding integrity has a moderate overhead of 5–66% depending on workloads; adding confidentiality further comes almost for free; and adding malware detection is cheap for small files (smaller than 1MB) but moderately expensive (2–35%) for large files (10MB).

3.7.1 Limitations

SeMiNAS currently does not encrypt file system meta-data, and a malicious cloud may still get sensitive information from directory and file names. SeMiNAS is also vulnerable to replay attacks, which usually requires building a Merkle tree [82] for the entire file system and is thus expensive for cloud environments. Moreover, SeMiNAS uses only one public cloud as the back-end and assumes the cloud providers can support NFS end-to-end integrity [92], which is not standardized yet. We propose to resolve all these limitations in Chapter 4.

Chapter 4

Kurma: Multi-Cloud Secure Middlewares

4.1 Introduction

Kurma is the final design of a cloud middleware system we propose to build. Kurma is based on SeMiNAS, and they have common design goals such as strong security, high performance, and flexible trade-off between security and performance. Their threat models are the same: public clouds are not trusted, clients are semi-trusted, and only the secure proxies are fully trusted. They also both provide the same security features: integrity, confidentiality, and malware detection. Both Kurma and SeMiNAS have a middleware architecture where clients access cloud storage using NFS indirectly via secure cloud proxies. They both have an on-premises persistent cache that is nearly identical.

Nevertheless, Kurma is better than SeMiNAS in four important aspects: robustness, security, performance, and feasibility.

First, Kurma is more robust than SeMiNAS by eliminating single points of failure, and thus enjoys higher availability. Although most public cloud providers have high availability close to five nines (99.999%) [138], availability and business continuity remain the largest obstacles for cloud computing [8]. A single cloud provider is itself a single point of failure [8]; once it is out of service, there is not much tenants can do but wait for it to come up. By using multiple clouds, Kurma solves this problem. In SeMiNAS, another single point of failure is the proxy server; Kurma eliminates this by storing meta-data in a highly available distributed service (ZooKeeper), and by partitioning file data among a cluster of on-premises NFS servers.

Second, Kurma is more secure than SeMiNAS by protecting file system meta-data and detecting replay attacks. SeMiNAS encrypts only file data but not file system meta-data such as file names and directory tree structure. This makes SeMiNAS susceptible to in-cloud side-channel attacks that might extract secret information from the meta-data. In contrast, Kurma saves on the cloud only encrypted data blocks, but not any file system meta-data whatsoever. Moreover, SeMiNAS's vulnerability to replay attacks is fixed by Kurma. Kurma keeps file system meta-data on premises and replicates the meta-data across the proxies (regions) via secure communication channels. A part of the replicated meta-data is a per-block version number, which can detect replay attacks that covertly replace fresh data with overwritten stale data.

Third, Kurma has higher performance than SeMiNAS by relaxing the (unnecessarily strong) consistency requirement, and optimizing NFS's compound procedures. We argue in Chapter 3 that

SeMiNAS's NFS file system consistency among geo-distributed proxies is feasible and desirable for many applications. However, the cost of global NFS consistency is high even in the presence of a persistent cache, as was demonstrated in the evaluation of SeMiNAS (Section 3.5), especially in the Filebench file-server macro-workload. To achieve the global NFS close-to-open consistency of NFS, SeMiNAS has to talk to the cloud NFS server synchronously upon each file open and close operation. This incurs a long latency because of round trips in WANs. Conversely, Kurma is willing to trade global NFS consistency for high performance. Instead of pursuing global NFS close-to-open consistency among all geo-distributed proxies, Kurma maintains NFS's close-to-open consistency at only the proxy (or regional) level. That is, NFS operations are synchronized with operations to the same proxy instance (i.e., within one common region), but not with operations to other instances (i.e., in other regions). This consistency model is the same as provided by traditional NAS appliances, and thus is enough for legacy applications. Kurma's geo-distributed proxies still share a common namespace by asynchronously replicating proxy-level changes to other proxies. Without overall consistency, however, the asynchronous replication may cause conflicts, which Kurma has to resolve automatically or with end users' intervention.

Kurma further improves performance by supporting large NFS compounds comprising many operations. This is inspired by our findings in the NFS benchmarking study that NFSv4's compound procedures are not effectively used (see Section 2.8). Kurma provides a customized library for clients to initiate large compound requests that are not limited by the POSIX file system API. Large compound requests can significantly boost performance by enabling more I/O coalescing and reducing the round trips between clients and Kurma proxies. To simplify error handling in case of partial failure in a large compound, Kurma also supports an execution of all operations in a single compound as one transaction.

Fourth, Kurma is more feasible than SeMiNAS with more realistic assumptions of what cloud providers support. SeMiNAS assumes the cloud NFS server supports NFS's end-to-end integrity [92], which simplifies management of security meta-data (see Section 3.3.4.1), but is non-standard part of the NFSv4.2 protocol proposal. As it stands today, SeMiNAS could not be deployed at a cloud scale and be objectively evaluated. On the other hand, Kurma uses existing cloud APIs and is thus more practical.

The rest of this chapter is organized as follows. We first introduce the background of Kurma in Section 4.2. We present our detailed design of Kurma in Section 4.3. We discuss related work in Section 4.4. And we propose the implementation and evaluation of Kurma in Chapter 5.

4.2 Background

In addition to NFS-Ganesha introduced in Section 3.4, Kurma also depends on several open-source distributed systems. These systems are important components of Kurma; understanding these systems is helpful in understanding Kurma. We discuss them here before we turn to Kurma's design.

4.2.1 ZooKeeper: A Distributed Coordination Service

Apache ZooKeeper [59] is a distributed coordination service. ZooKeeper achieves consensus among distributed systems using an algorithm called ZAB, short for ZooKeeper Atomic Broad-

cast [63]. ZooKeeper is popular and regarded as "The King of Coordination" [12]. It is also widely used for leader selection, configuration management, distributed synchronization, and namespace management. ZooKeeper provides strong consistency and has been used in cloud service as "consistency anchor" [1].

In ZooKeeper, distributed systems coordinate with each other through a shared tree-structured namespace. Each node in the namespace is called *znode*, and the path from the tree root to a znode is called *zpath*. Each znode can store a small amount (typically less than 1MB) of data, and have children znodes. ZooKeeper keeps all data (including the namespace metadata and znode data) in memory to achieve high throughput and low latency. Kurma achieves durability by maintaining replicas among its servers, and saving transaction logs and snapshots in a persistent store. ZooKeeper is transactional and has a global ordering of all transactions. Therefore, it guarantees a consistent view of the tree-structured namespace. ZooKeeper supports a rich set of attributes for each znode, including a unique ID, ACL, number of children, as well as version numbers and timestamps for data changes, ACL changes, and children member changes. ZooKeeper allows clients to register watchers to znodes, and will notify interested clients upon changes on watched znodes.

ZooKeeper is stable and has been successfully used in many industrial applications [46]. Although ZooKeeper is implemented in Java, it provides both C and Java APIs to clients. ZooKeeper also has a helper library called Apache Curator [43]. Curator includes a higher level ZooKeeper API, recipes for common usage of ZooKeeper, a testing framework, and other utilities.

Kurma uses ZooKeeper for three purposes: (1) storing the namespace data (file attributes, directory structure, and block mapping) of the Kurma file system; (2) coordinating multiple NFS servers in the same region; and (3) transaction execution of large NFS compounds. Kurma uses Apache Curator [43] to simplify the programming of ZooKeeper.

4.2.2 Hedwig: A Publish-Subscribe System

Apache Hedwig is an open-source "publish-subscribe system designed to carry large amounts of data across the Internet in a guaranteed-delivery fashion" [44]. Clients to Hedwig are either *publishers* (sending data) or *subscribers* (receiving data). Hedwig is topic based: a publisher posts messages to a topic, and Hedwig delivers the messages in the published order to all subscribers that are interested in that topic.

Hedwig is designed for inter-data-center communication; it consists of geo-distributed *regions* spread across the Internet. A message published in one region is delivered to subscribers in all regions. Hedwig achieves guaranteed delivery by saving messages in a persistent store, replicating messages in all interested regions, and then sending messages to all subscribers until they acknowledge the delivery. To achieve high availability, Hedwig uses ZooKeeper for meta-data management, and uses BookKeeper [42], a highly available replicating log service, for persistent store. Hedwig supports both synchronous and asynchronous publishing; it also supports message filters in subscriptions.

Kurma uses Hedwig for distributed state replication to maintain a global namespace. Using Hedwig, a proxy asynchronously propagates changes to the namespace to other proxies in remote regions. A per-block version number is a part of the namespace data, and is used for detecting replay attacks. Hedwig's filters can be used for advanced access control, for example when a certain file should not be visible by clients in a region.

4.2.3 Thrift: A Cross-Language RPC Framework

Apache Thrift is cross-language RPC framework for scalable cross-language services development [45]. Thrift allows seamless integration of services built in different languages, including C, C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, and many others. Thrift includes a code generator to generate messages (structs or types), RPC stubs (services), and data marshaling routines; it is similar to Sun's `rpcgen`, but not limited to support only the C language. In addition to a general RPC framework, Thrift also has concrete building blocks of high performance RPC services, such as scalable multi-threaded servers. Thrift has been used by large companies such as Facebook, Siemens, and Uber [47].

Kurma uses Thrift for two purposes: (1) defining messages stored in ZooKeeper and replicated among proxies, and (2) implementing RPC communication between the Kurma services running in a proxy. For example, the Kurma NFS server talks to Kurma's file system server using RPC implemented using Thrift. Thrift supports data compression when encoding messages, and considerably cut the memory footprint of compressible data such as block version numbers. This is particularly helpful when storing compressible data in ZooKeeper, which keeps all its data in memory.

4.3 Kurma Design

We present the design of Kurma including its threat model, design goals, architecture, consistency model, caching, file system partition, and security features.

4.3.1 Design Goals

Kurma's design goals are similar to those of SeMiNAS, except that Kurma strives for higher availability, stronger security, and better performance. We list Kurma's five design goals by descending importance:

- **High availability**: Kurma should have no single point of failure; it should be available when a part of proxy machines are down and when one or two public clouds are down.
- **Strong security**: Kurma should be secure against attacks in its threat model, including advanced replay attacks. It should ensure integrity and confidentiality to both file data and meta-data stored in public clouds, and be immune to malware from clients.
- **High performance**: Kurma should minimize the performance overhead of its security features, and should optimize performance for low latency and high throughput.
- **Flexibility**: Kurma should be configurable in many aspects to support a wide range of security policies and performance requirements. Security, consistency, and compatibility can be traded for performance according to workloads.
- **Simplicity**: Kurma's architecture should be as simple as possible to ease development and maintenance.

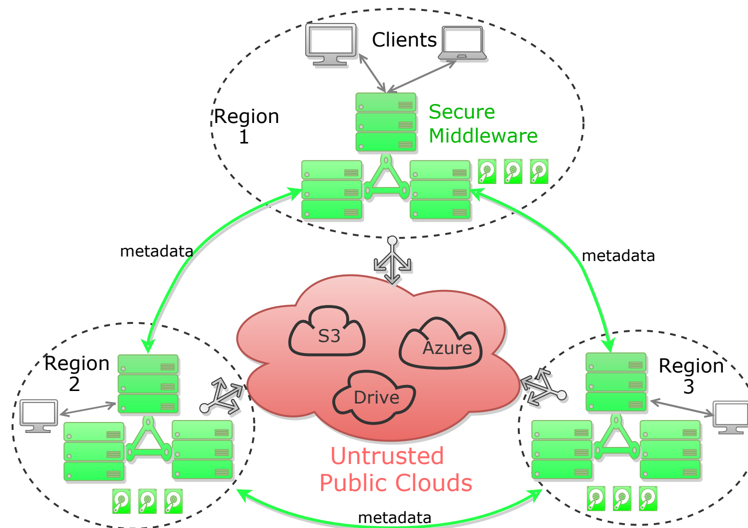


Figure 4.1: Architecture of Kurma. Kurma consists of geo-distributed proxies in regions where end-users reside. Each region has one single proxy. Although the single proxy may be physically distributed among multiple machines, Kurma coordinates these machines using ZooKeeper to ensure they behave as a single logical proxy. All proxies are inter-connected by a trusted communication link; Kurma uses the links to replicating meta-data among proxies.

4.3.2 Architecture

Kurma's architecture is analogous to SeMiNAS: on-premises proxies (trusted) acts as security bridges between semi-trusted clients and untrusted public clouds. However, Kurma has three major architectural differences, as illustrated in Figure 4.1. First, each geographic region has a distributed proxy instead of a centralized one. A conceptual Kurma proxy consists of a cluster of machines which are properly coordinated using ZooKeeper. This distributed proxy avoids any single point of failure and enjoys better scalability and availability.

Second, Kurma proxies in geo-distributed regions are interconnected. In SeMiNAS, proxies communicate with each other only indirectly through the cloud NFS server. This significantly simplifies the SeMiNAS's architecture, but sharing secrets through the untrusted public cloud makes replay attacks difficult to detect. With a trusted direct communication channel between each pair of proxies, secret file system meta-data can be easily shared, and replay attacks can be efficiently detected using version numbers of data blocks.

Third, Kurma uses multiple public clouds, instead of a single one, as back-ends. Kurma uses clouds as block stores other than file servers. For a data block, Kurma stores in each cloud either a replica, or a part of the erasure coding results of the block. In case of cloud outage, Kurma can continue its service by accessing other clouds that are still available; in case of data corruption in cloud, Kurma can restore the data from other replicas or other erasure coding parts.

Kurma stored data in clouds and meta-data in on-premises ZooKeeper. Kurma replicate data among multiple clouds using simple replication or erasure code; Kurma replicates meta-data across proxies asynchronously. Note that each Kurma proxy runs one instance of ZooKeeper, and Kurma uses Hedwig to replicate meta-data in geo-distributed ZooKeeper instances by replaying changes made by each proxy.

Each Kurma proxy deploys three services on its cluster of machines: an NFS service facing clients (NFS service), a persistent file cache service (PCache service), and a Kurma file system service (FS service) communicating to public clouds and other proxies. Kurma also deploys three services it depends on: ZooKeeper, BookKeeper, and Hedwig. These services are loosely coupled and use RPC or network sockets for mutual communication. Therefore, they can be deployed in different machines, and enjoy the flexibility of adjusting the number of specific servers according to load.

The relationships among Kurma's NFS service, PCache service, FS service, their supporting services, and the remaining components of the proxy are illustrated in Figure 4.2. A Kurma NFS server processes NFS requests initiated by clients. A Kurma proxy may contain multiple NFS servers each serving a part of the overall namespace. When an NFS server encounters an operation to a file system object that should be handled by a different NFS server, it can redirect the client to the correct server using NFSv4's referral method [111]. The distribution of file system objects among NFS servers is determined by the Kurma file system and saved in ZooKeeper.

Kurma's NFS servers run NFS-Ganesha (introduced in Section 3.4.1) with two FSAL layers in the NFS server: FSAL_PCACHE and FSAL_KURMA. FSAL_PCACHE talks to the persistent write-back cache service (PCache), which is the same as SeMiNAS's cache (see Section 3.3.6) but differ only in the way of write-back. In SeMiNAS, write-back to cloud happens synchronously upon file close to achieve close-to-open consistency; Kurma does not guarantee overall consistency and write-back happens asynchronously. When necessary, FSAL_PCACHE also scans dirty data for viruses before inserting it into the cache.

The lower FSAL layer is FSAL_KURMA, which talks to the Kurma file system (FS) service. Kurma FS stores file system meta-data in ZooKeeper and data blocks on public clouds. To maintain a global namespace, Kurma FS publishes local file system changes to, and receives remote changes from remote proxies using Hedwig. When the message of a remote change is received, Kurma FS tries to replay the change locally. Lacking global synchronization, the replay may fail due to conflicts, and Kurma FS will then resolve file system conflicts using a traditional solution [102]. Before writing block data to the clouds, Kurma FS also performs authentication and encryption, as well as optional erasure coding if configured to do so.

4.3.3 Meta-Data Management

Kurma's file system service (Kurma FS) maintains hierarchical namespaces by storing file system meta-data in ZooKeeper. A Kurma file system instance is called a volume and is identified by a volume ID (VID). VID is a variable-length opaque binary that is unique among all proxies. Each volume can choose its backing clouds (e.g., Amazon S3 and Google Cloud Storage), replication or erasure coding settings with a configuration file. For each back-end cloud, all data blocks of a volume are stored as key-value objects in one container identified by the SHA256 value of the volume ID.

Within a volume, each file system object (file, directory, symbolic link, etc.) is identified by a tuple of $\langle$*OID*, *OCREATOR*, *OTYPE*, *OFLAGS*$\rangle$, where OID is a 128-bit integer of object ID, OCREATOR is a 8-bit-long ID of the proxy that creates the object, OTYPE is a 8-bit value of the object type (e.g., file or directory), and OFLAGS is a 8-bit value of flags (e.g., a flag indicating a private file not shared to other proxies). Kurma uses 128-bit OIDs to ensure OIDs are never reused and always stay unique: assuming there are one million machines, each of which can create a file

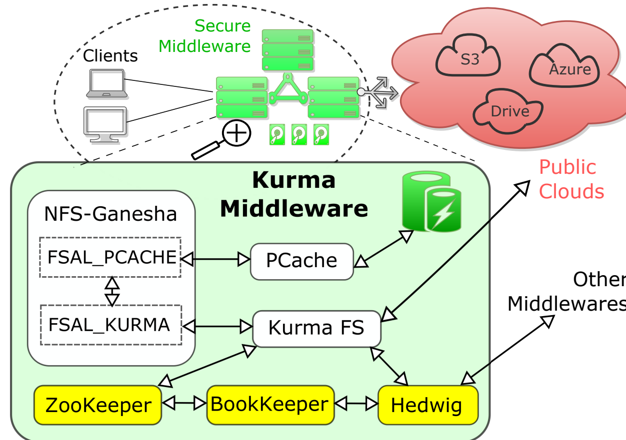


Figure 4.2: Components of a Kurma Proxy. The arrows indicate interaction among the components. A proxy comprises of three services: NFS, PCache, and Kurma FS. The NFS server runs NFS-Ganesha and exports cached and cloud-backed files to clients. The PCache service manages the persistent write-back cache. Kurma FS serves as the NFS server's secure proxy to the cloud back-end, and maintains a global namespace by replicating meta-data across all proxies. Kurma FS depends ZooKeeper, BookKeeper, and Hedwig for meta-data storage, distributed coordination, and meta-data replication.

system object in one nanosecond, it will take more than 10^{16} years to use up the 128-bit ID. Kurma stores the largest used OID as a 128-bit integer in ZooKeeper. Allocating OIDs is a ZooKeeper transaction that atomically reads and increments the integer. To avoid frequent allocations, each transaction allocates a configurable number (100 by default) of OIDs at a time. Since each proxy allocates its own OIDs, the OCREATOR is needed to differentiate among proxies. The concatenated hex representation of VID, OID, and OCREATOR forms the zpath (ZooKeeper path, see Section 4.2.1) to the znode (ZooKeeper node) that stores the meta-data of the file system object.

For a directory, its meta-data content is similar to normal file systems, and comprises attributes and ⟨name, OID⟩ pairs of its children objects. For a regular file, its meta-data includes traditional attributes as well as a list of ⟨*MID*, *EFK*⟩ pairs, where MID is the ID of a proxy, and EFK is the file key (FK) encrypted by the public key of the proxy. The list of ⟨MID, EFK⟩ pair is used for exchanging FK among proxies as discussed in Section 3.3.4.2.

Another important part of a file's meta-data is about block mapping that identified the data blocks in the file's address space. In Kurma, a data block is 64KB by default, and is identified by a tuple of ⟨*OFFSET*, *VERSION*, *MODIFIER*⟩, where OFFSET is the block offset, VERSION is the block version number, and MODIFIER is the MID of the proxy performed the latest change on the block. Each data block is stored as a key-value ⟨*BK*, *BV*⟩ object on clouds. The block key (BK) is the SHA256 value of the concatenation of VID (volume ID), OID (object ID of the file), OFFSET, VERSION, and MODIFIER; the block value (BV) is the concatenation of MAC, and auth-encrypted block data and security meta-data (including the version number, the offset, and a timestamp). When stored in ZooKeeper, the block mappings are simply an array of VERSION, and an array of MODIFIER both indexed by the offset. The two arrays, if not inlined in case of small files, are stored in children znodes of the file's znode. To reduce the memory footprint of the all-in-memory ZooKeeper, Kurma compresses the arrays before storing them there. We

expect the array of version numbers to be highly compressible because it usually contains only small numbers (reflecting small number of changes) and many consecutive runs of equal numbers (reflecting locality of changes). The block mapping arrays may be split and stored in multiple znodes due to the 1MB capacity limit of a znode.

4.3.4 Security

Kurma provides integrity, confidentiality, and malware detection as SeMiNAS does. Kurma's malware detection is as described in Section 3.3.5, and Kurma also use authenticated-encryption to provide encryption and authentication together. However, Kurma is more secure with the following two differences: (1) Kurma detects replay attacks, and (2) Kurma protects not only file data but also file meta-data including file names, file system tree structure, and file access patterns.

To detect replay attacks, Kurma saves a per-block version number in its meta-data, which is replicated among all proxies. When reading a block (a key-value ⟨BK, BV⟩ object) from public clouds, Kurma generates BK using its VID (volume ID), file OID, OFFSET, VERSION, and MODIFIER as inputs to SHA256. Therefore, each version of the block has a unique BK. Without knowing VID, OID, the offset, or the version number, malicious clouds could not tell if two ⟨BK, BV⟩ objects are two different versions of the same block. Moreover, each block value (BV) also contains additional security meta-data including the version number, the offset, and a timestamp. Even if malicious clouds obtained the OID and VID somehow, they cannot replay BV with an old version, which should contains the wrong block version number or offset. Therefore, malicious clouds are unable to carry out replay attacks without being caught.

Because Kurma stores the meta-data of the file system in on-premises ZooKeeper, the namespace information is unknown to clouds. This prevents malicious clouds from guessing sensitive information out of directory and file names. This also stops side-channel attacks that extract sensitive information by correlating operations on related file system objects. Kurma protects the per-file key (FK) with the public keys of proxies using the approach described in Section 3.3.4.2. Kurma stores the encrypted FKs (EFKs) of a file as part of its meta-data in ZooKeeper and shares them among proxies using Hedwig.

4.3.5 Consistency Model

Kurma relaxes the global NFS close-to-open consistency guarantee of SeMiNAS by using RESTful APIs instead of NFS when talking to public clouds. This relaxation allows file meta-data operations being processed within the on-premises network. When file data is also cached in the proxy's on-premises persistent cache, operations can be processed without talking to public clouds at all. This cuts round trips in the WANs and significantly lowers operation latency. In other words, Kurma trades unnecessarily strong consistency for higher performance.

This degree of trade-off is acceptable because the relaxed consistency is still the same as provided by traditional NAS appliances. That is, clients in a region have a consistent view of the file system. For NFS, the consistent view means close-to-open consistency [71], which guarantees a client sees all changes to a file that are made before the moment when the client opens the file. In Kurma's consistency model, a client in a region will see all previous changes to the file that are made by other clients in the same region, but not necessarily the changes made by clients in other regions. However, the client will see remote changes as soon as the changes propagate to the local

region through Hedwig. Therefore, stale data is only observable in the small time window after remote changes and before the propagation. Since concurrent file updates are rare [69], conflicts should be rare as well.

The eventual consistency of cloud storage means what the value of a key we read from clouds may not be the latest value, but an old value. Kurma uses two mechanisms to prevent reading stale value. First, each version of every data block has an unique key that is generated by SHA256 using the block version as one of the inputs. Kurma does not update any value in the clouds; instead, it creates a new key-value pair when modifying the data of a block. Therefore, a key has only one read-only value and thus does not have stale value. Second, the value of each block also contains the version number of the block. Before storing block value on clouds, Kurma also authenticates the version number using the cryptographic key of the parent file. When reading the block value, Kurma can thus verify whether the version number is authentic, and whether this version number retrieved from clouds matches the version number of the block Kurma intends to read.

The close-to-open consistency requires every client to (1) write back all dirty data of a file before it closes the file, and (2) revalidate its local cache of the file before it opens the file. However, these two requirements are no longer applicable to Kurma's persistent cache for two reasons. First, a file is served only by one Kurma NFS server and one PCache server thanks to the coordination of ZooKeeper, and there is no inconsistency issue with only one copy of data in one region. Second, Kurma provides only region-level consistency, and inconsistency in copies from multiple regions does not compromise its regional consistency guarantee.

Although not required, PCache also revalidates its cache upon file open by consulting Kurma FS, in order to limit the time window of cross-region inconsistency. If Kurma FS has received changes by remote proxies to the file, PCache will know. Then, depending on whether the cache is dirty or not, PCache will invalidate its cache or immediately write dirty data back to Kurma FS. There might be conflicts during the write-back, which Kurma FS has to resolve.

4.3.6 Partition over Multiple NFS Servers

To make Kurma proxies robust, Kurma partitioned file system objects (files and directories) of Kurma volumes across multiple NFS servers. Each Kurma proxy maintains a list of the NFS servers running in that proxy, and designates a primary NFS server for each file system object. Each file system object has only one primary NFS server at a time, and the primary NFS server processes all operations to that object. Kurma prefers the least loaded running NFS server when making designation decision. The list of running NFS servers and the designation records of primary NFS servers are stored in ZooKeeper.

To simplify the designation of primary NFS server, Kurma does that only for directories whose directory depths are lower than a configurable level (3 by default). For a file system object without a directly designated NFS server, it inherits the primary NFS server from the lowest ancestor in the directory tree.

When the primary NFS server of a file system object is down, Kurma designates a still-running NFS server as the new primary. We propose to achieve the fail over between NFS servers by setting all NFS servers as an NFS cluster [96]. When the failed NFS server recovers, it will read designation information from ZooKeeper, and know it is no longer the primary NFS server for file system objects designated before the outage. When the recovered NFS server still receives requests on the old file system objects, it will redirect those request to their new primary NFS servers.

Each Kurma proxy stores file system meta-data in one single instance of ZooKeeper, and uses ZooKeeper to coordinate multiple NFS servers. Therefore, NFS requests operating on two file system objects in two NFS servers, such as RENAME, is not a problem because the single ZooKeeper will process changes of the meta-data backing both NFS servers.

4.3.7 Multiple Clouds

The availability of most cloud providers is much higher than the availability of private computing infrastructure [138]. However, cloud outage did happen, and can be quite bad sometimes [127]. Researchers from University of California at Berkeley considered availability as the top one obstacle to growth of cloud computing, and they pointed out the solution is to use multiple cloud providers [8]. By saving data redundantly on multiple clouds, Kurma can achieve high availability and ensure business continuity in the presence of cloud failures (corruption and outage). Suppose failures of clouds are independent and the failure rate of each cloud is λ, then the availability of using two clouds would be $1 - \lambda^2$. Therefore, Kurma can achieve six-nine availability (99.9999%) when using two clouds, each of which has an availability of 99.9%.

Depending on the configuration, Kurma uses either replication or erasure coding to store data redundantly on multiple clouds. These two methods represent different trade-offs among storage and computation overhead.

To tolerate the failure of f clouds using replication, we need to replicate over $f + 1$ clouds. A write operation (i.e., PUT) finishes only when we have put a replica in each of the $f + 1$ clouds, whereas a read operation (i.e., GET) finishes as soon as one valid replica is read. The storage overhead and write amplification are both $(f + 1)\times$. The read amplification is zero in the best case (no failure), but $(f + 1)\times$ in the worst case (f failures). Replication requires no extra computation.

An erasure code transforms a message of $k > 1$ symbols into a longer message with $k + m$ symbols, and it can recover the original message with any k symbols of the longer message. Put differently, the erasure code can tolerate m failures. Therefore, to tolerate the failure of f clouds using erasure coding, Kurma write to $k + f$ clouds and read from at least k clouds. In the best case, a read operations has to read from k clouds but each read size is $\frac{1}{k}$ of the original block size; in the worst case, it has to read from $k + f$ clouds. The storage overhead and write amplification are both $\frac{f+k}{k}\times$. Unlike replication, erasure coding requires extra computation.

In sum, replication uses more extra space, but reads from fewer clouds, and does not cost any computation; conversely, erasure coding uses less extra space, but reads from more clouds, and costs extra computation. Kurma leaves the choice to end user by supporting both, and uses the method as specified in its configuration file.

4.3.8 NFS Transactional Compounds

In addition to trading consistency for performance, Kurma optimizes performance further by taking full advantage of NFSv4's compound procedures, which we found to be woefully under-utilized (see Section 2.6.1). The key reason of the under-utilization is the synchronous and low-level nature of the POSIX file system API. To overcome this limitation, we propose a customized client library of high-level file system functions. For example, a function that opens, reads, and closes a file all at once will be efficient and convenient for manipulating small files. Another example is a function that copies one or more files without moving data back and forth between the client and the server.

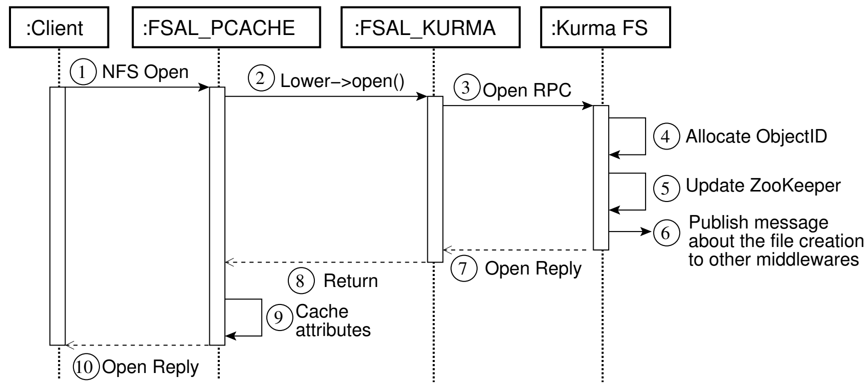


Figure 4.3: Sequence Diagram of File Creation. The sequences of other operations follow a similar direction of moving from client to FSAL_PCACHE, FSAL_KURMA, Kurma FS, and then gradually return back to the client.

This library will allow applications to initiate large NFS compounds with many operations. Large NFS compounds can considerably cut round trips between client and server, and enable larger degree of parallelism and I/O coalescing on the server side.

Another obstacle to large NFS compounds is the difficulty of error handling in case of failures. Because failed operations may leave the file system in an inconsistent state, it is crucial to properly handle them. With many operations in a large compound, failures can happen in many different scenarios. Distinguishing and fixing each scenario is inconvenient and error-prone. Kurma's solution to this problem is transactional executions of large compounds. When an application choose transactional execution of a compound, Kurma will ensure the compound be processed atomically: either all operations succeed, or none of the them takes place. The application will also benefit from other transaction properties, such as consistency and durability.

Because Kurma FS uses ZooKeeper as meta-data store, Kurma can leverage ZooKeeper's transaction support to implement transactional NFS compounds.

4.3.9 Kurma Operation Examples

To further clarify the design of Kurma, we work through the steps Kurma takes to process typical file system operations.

4.3.9.1 Create a New File

The sequence diagram of file creation is illustrated in Figure 4.3. When a Kurma client initiates a file creation, the client sends an NFS OPEN request with the OPEN4_CREATE flag (similar to O_CREATE) to the Kurma NFS server backing the mounted NFS volume. The Kurma NFS server, running NFS-Ganesha, decodes the OPEN request and passes it to the FSAL layers. The upper FSAL_PCACHE layer immediately redirects the request down to the lower FSAL_KURMA layer. Then, FSAL_KURMA sends an OPEN RPC request with proper flags to a Kurma FS server; and will simply return results back to the upper FSAL_PCACHE layer once the RPC reply is received. The Kurma FS server, upon receiving the RPC request, performs the following four tasks in order: (1) allocates a new object ID (OID); (2) atomically creates a znode of the new file and update the

znode of its parent directory in ZooKeeper; (3) asynchronously publishes a message about this file creation to other proxies; and (4) replies the RPC request with file creation results and the file's attributes back to FSAL_KURMA. When the control goes back to FSAL_PCACHE, it caches the file attributes from the RPC reply if the creation succeeded, and returns control to NFS-Ganesha. Finally, NFS-Ganesha sends an NFS reply to the original NFS OPEN request.

4.3.9.2 Open an Existing File

Opening an existing file goes through the same steps as creating a file, but differs in two aspects. First, the OPEN4_CREAT flag is not set when opening an existing file. Second, FSAL_PCACHE revalidates (and invalidates if necessary) its cache using the latest file attributes it receives from FSAL_KURMA. The cache revalidation happens in the same manner as the NFS client revalidates its own client-side cache [21]. That is, FSAL_PCACHE compares current remote-modify-time (i.e., the timestamp of the last remote modify) with the remote-modify-time saved previously when the cache is created. If the two times are the same, this means the file has not been modified by other proxies and the cache is still valid; if the times differ, then FSAL_PCACHE invalidates its cache of the file.

4.3.9.3 Write to a File

Writing a file is simple. A write operation reaches only FSAL_PCACHE, and then returns back to NFS-Ganesha without talking to the underlying FSAL_KURMA. This is true even when the write request is stable (i.e., requesting dirty data be flushed by setting DATA_SYNC4 or FILE_SYNC4 flag). A stable write incurs only an additional flush to the persistent cache. To improve performance, writing back to the clouds is deferred after file close.

4.3.9.4 Read From a File

When a READ NFS request reaches the FSAL_PCACHE layer of a Kurma NFS server, FSAL_PCACHE first tries to serve the requested data from its cache. Upon a cache miss, FSAL_PCACHE then loads the data (rounded up and aligned to the to enclosed blocks) from the lower FSAL_KURMA layer, which sends a READ RPC to a Kurma server. The Kurma server retrieves the corresponding data blocks from the clouds, authenticates and decrypts them, and sends the data to FSAL_KURMA and then to FSAL_PCACHE. FSAL_PCACHE serves the pending NFS read request and insert the data into the persistent cache. When inserting data fetched from the clouds into cache, FSAL_PCACHE should take care to avoid overwriting any overlapping dirty cache data.

4.3.9.5 Close a File

A CLOSE NFS request triggers the FSAL_PCACHE layer to write dirty data back to public clouds, and optionally to scan for viruses. FSAL_PCACHE first flushes dirty data and meta-data to the local storage of the persistent cache, then registers asynchronous write-back of dirty data, and finally it returns. There are dedicated threads to handle asynchronous write-backs. These threads form a consumer-producer relationship with FSAL_PCACHE. The write-back threads perform read-modify-update first in case of unaligned writes, and then write back dirty data asynchronously by sending WRITE RPCs to Kurma FS servers. After that, Kurma FS servers authenticate and encrypt

the data blocks, writes the blocks to the clouds (after erasure coding if necessary), increments the version numbers of the blocks, and publishes a message about the change to remote proxies through Hedwig.

4.4 Related Work

Kurma is inspired by many previous studies in related areas, and we have already studied some of those in the discussion of SeMiNAS (Section 3.6). Here, we focus on studies that are related to Kurma features not available in SeMiNAS. Specifically, we compare Kurma to other file and storage systems that (1) use multiple clouds (i.e., a cloud-of-clouds) as storage back-end, (2) guarantee data or meta-data freshness in the face of replay attacks, and (3) support compound operations.

4.4.1 A Cloud-of-Clouds

Using multiple cloud providers is an effective way to ensure high availability and business continuity in case of cloud failures [8]. There are several studies of multi-cloud systems [1,2,15,33,58], and all of them store data redundantly on different clouds using either replication or erasure coding.

In addition to higher availability, multiple clouds can also be used to enhance security. Because compromises or collusion across multiple cloud providers is less likely, dispersing secrets among them is more secure than storing all of them on a single cloud [5]. For example, DepSky [15] and SCFS [1] ensure security by secretly storing keys on multiple clouds and preventing any single cloud alone from accessing the keys.

However, most of these multi-clouds storage systems [2, 15, 33, 58], including Hybris [33], provide only key-value stores, instead of file systems as Kurma does. Hybris is also a middleware system, but it resides in a single geographic location and does not support geo-distributed proxies. SCFS provides a POSIX-like file system, but it is for personal users instead of enterprises. SCFS favors the scenario when all of a client's files can fit in the client's local storage. SCFS is not a cloud middleware, and needs a trusted in-cloud meta-data service as a consistency anchor.

4.4.2 Freshness Guarantees

Not many cryptographic file systems guarantee data or meta-data freshness [117] because replay attacks are difficult to handle. SiRiUS [52] ensures partial meta-data freshness but not data freshness. SUNDR [72], SPORC [38], and Depot [75] all guarantee *fork consistency* that can detect freshness violations with out-of-band inter-client communication.

Among the few file systems that guarantee freshness of both data and meta-data, most of them [34, 48, 117] use Merkle trees [82] or its variants to detect replay attacks. Iris [117] uses a *balanced Merkle-tree* that supports parallel updates from multiple clients. Athos [53] does not use Merkle trees, and guarantees freshness after replacing the hierarchical structure of the directory tree with an equivalent but different structure based on skip lists. SCFS [1] also provides freshness without using Merkle trees, but it does so by relying on a trusted and centralized meta-data service running on cloud.

Other cloud systems provide data freshness guarantees for key-value stores, instead of file system services. CloudProof [98] provides a mechanism for clients to verify freshness; Venus [112]

and Hybris [33] guarantee freshness by providing strong global consistency out of the eventual consistency model of cloud key-value stores.

Kurma's approach to freshness guarantees is significantly different from all aforementioned systems. Kurma is free from the performance overhead of Merkle trees, and instead uses a reliable publish-subscribe service (Hedwig) to replicate meta-data and block version numbers among geo-distributed cloud proxies. Unlike SCFS [1], Kurma does not rely on any trusted third party for meta-data management or key distribution.

4.4.3 Compound Operations

Improving performance by coalescing and compounding operations is a common technique widely used in storage and networking subsystems. The idea has been exemplified by disk I/O schedulers [121] and Nagle's TCP algorithm [134]. These traditional coalescing and compounding techniques are agnostic to file system and networking APIs (i.e., system calls), and happen under the hood inside the kernel.

However, as storage devices and networks grow increasingly faster, these implicit compounding techniques become inadequate. Pursuing the best system performance, researchers started to propose explicit compounding or vector-based APIs for high speed system calls [99], storage devices [128, 129], and networks [61]. It is demonstrated in their studies that explicit compounding can greatly improve performance. As a network-based storage protocol, NFS compounding holds a promising potential if supplied with an explicit compounding API. In addition, Kurma's transactional execution of compounds will make NFS not only faster but also much more convenient to use.

Chapter 5

Proposed Work

Our work can be improved and extended in many aspects. In this chapter, we discuss further work that we propose to accomplish in this thesis. We will discuss future work beyond this thesis in Chapter 6. The proposed work covers three aspects: (1) the implementation and thorough evaluation of the current design of Kurma; (2) the development of new Kurma features including private namespace and snapshotting; and (3) the performance optimization using NFS transactional compounds with many operations.

5.1 Kurma Implementation and Evaluation

Currently, we have finished all details of Kurma's design (except transactional compounds) discussed in Chapter 4, such as the format of Kurma FS's meta-data stored in ZooKeeper, the RPC protocol between FSAL_KURMA and Kurma FS, the format of messages exchanged cross proxies using Hedwig, the RESTful cloud storage clients to several popular cloud providers, etc. As to the implementation, we have also finished FSAL_PCACHE, FSAL_KURMA, and Kurma FS; and we propose to complete the following implementation tasks that are not finished yet:

1. meta-data replication among proxies and the associated conflicts resolution;
2. partition of file system objects among multiple NFS servers of a proxy;
3. support of erasure coding across multiple public clouds. and
4. error handling upon detection of replay attacks and cloud outage;

Evaluation of Kurma is crucial to judge whether Kurma has achieved the design goals we set. We propose to evaluate Kurma with the following correctness and performance tests:

1. `xfstests` to ensure Kurma has correctly implemented the NFS semantics;
2. security tests to make sure Kurma withstands all attacks it defends against;
3. global namespace tests to ensure Kurma provides a global namespace to all proxies and handles conflicts properly;

4. micro-benchmarking that either checks our design assumptions (e.g., the compressibility of the array of version numbers discussed in Section 4.3.3), or guides our optimizations (e.g., the prioritizing of public clouds based on their performance); and

5. performance tests to evaluate Kurma's competitiveness to other comparable systems.

5.2 Development of New Kurma Features

We also propose to develop two new Kurma features we think are interesting:

1. Private Namespaces. The capability to share files among geo-distributed regions is desirable, but it is not always needed. In fact, file sharing is infrequent and rarely concurrent in realistic workloads [69]. This makes private namespaces, files within which are not shared across proxies, meaningful. We propose to design and implement private namespaces in Kurma, and evaluate its performance benefits compared to normal namespaces.

2. Snapshotting. Snapshotting is highly desirable nowadays due to the popularity of virtualization, and NFS is frequently used to host VM disk images [124]. This makes snapshotting an appealing feature to have in Kurma, and we propose to develop it in this thesis as well. Adding snapshotting should not be difficult considering that Kurma already maintains a version number of each data block, and ZooKeeper, the meta-data store, also supports versioning.

5.3 NFS Transactional Compounds

NFS transactional compounds comprising many operations, discussed in Section 4.3.8, is a major performance optimization we also propose to develop in this thesis. We have begun to investigate this work, but the following remains to be finished:

1. The design and implementation of a high-level NFS client library that allows applications to initiate NFS compounds with large number of operations.

2. The support of transactional execution in Kurma's NFS and FS servers under the help of ZooKeeper.

3. The evaluation of the performance boost of large NFS compounds with and without transactional execution.

Chapter 6

Conclusions

Because of its advantages in availability, cost efficiency, flexibility, and scalability, cloud computing becomes more and more popular. However, to realize the dream of computing as a utility, cloud computing still faces many obstacles, with the top three being availability (ironically), vendor lock-in, and data confidentiality [8]. In this thesis, we proposed Kurma, a cloud middleware system, to address the storage aspect of these obstacles: Kurma uses multiple cloud providers for high availability, uses standard NFS protocol to alleviate vendor lock-in and improve application portability, and deploys encryption and authentication to provide data confidentiality and integrity. Kurma acts as a proxy that provides legacy NAS-based clients with seamless, fast, secure, and highly available cloud storage service without requiring any changes to either clients or cloud providers.

Many NAS-based applications demand high performance (especially low latency), which can be difficult to achieve in clouds due to long physical distances and high network latency. To provide high performance in Kurma, we started from analyzing the performance of different versions of NFS (NFSv3 and NFSv4.1), the storage protocol between clients and Kurma proxies. We conducted a comprehensive and in-depth benchmarking study using a wide range of workloads in different network settings. We fixed a severe performance problem of NFSv4.1 and improved its performance by up to 11×. We found that NFSv4.1 has comparable performance to NFSv3, and holds the potential of much better performance than NFSv3 when its advanced features, such as delegations, are effective. We also identified NFSv4.1's compound procedures, which are currently under-utilized, as an opportunity to significantly boost NFSv4.1's performance.

Then, as a first step to develop Kurma, we designed, implemented, and evaluated a simplified prototype called SeMiNAS, which uses only a single cloud as back-end and is susceptible to replay attacks. SeMiNAS is a secure cloud proxy that provides data integrity, confidentiality, and malware detection. It enjoys flexible trade-off between security and performance by allowing each security feature to be enabled and configured separately. To improve performance, SeMiNAS uses a persistent cache to serve most I/O requests in the faster on-premises network. SeMiNAS embeds security meta-data into file data using a novel method that does not incur extra round trips between the proxy and the cloud. However, the novel meta-data management assumes the cloud provider supports an NFS extension [92] that is not standardized yet.

Based on the experience with SeMiNAS, we presented the design of Kurma, which is more robust, secure, and efficient. Kurma does not have single points of failure: it uses multiple public clouds as back-end, and is built on top of fault-tolerant distributed services such as ZooKeeper and Hedwig. Kurma protects the integrity and confidentiality of not only file data but also meta-

data such as file names and directory structures. Kurma has an efficient solution to replay attacks: it maintains a version number for each data block, and replicates the version numbers across all geo-distributed proxies. Kurma is also faster and more realistic than SeMiNAS by trading off cross-region consistency for performance and using only existing RESTful cloud APIs.

Finally, we proposed the implementation and evaluation of Kurma, as well as two directions of improvement. One is to add new Kurma FS features including private namespaces and snapshotting; the other is to optimize performance further with large and optionally transactional NFS compounds with many file system operations.

We hope Kurma exemplifies an inspiring exploration of, and an elegant solution to the unique problem of simultaneously achieving high availability, security, and performance in hybrid cloud systems that combine traditional on-premises storage and public cloud storage. We also plan to open source Kurma in the hope to benefit both research and engineering communities.

6.1 Future Work

This work can be extended further beyond the scope of the thesis. We see at least four interesting and closely related future directions:

1. More features in Kurma FS. Kurma FS is a good platform for adding more file system features such as compression, deduplication, online file system checking, and pNFS support.

2. Optional strong global consistency. Kurma trades off strong global consistency for better performance; however, it may still be desirable to have optional global consistency for certain files. One plausible way is to use file mastership. By designating a master proxy of a file (e.g., the creator proxy), global consistency can be achieved by synchronizing changes to the file in all proxies with the master.

3. Cost awareness and optimization. Cloud operations have different cost and performance. For example, read data is more expensive than writing [1], and one cloud provider may be slower but cheaper than another. Lowering Kurma's cost under a reasonable performance is an interesting topic worth exploring.

4. Transactional NFS compounds API in general languages. We proposed a customized client file system library to initiate large NFS. However, it would be more convenient and elegant to support that in general programming languages. Some modern languages, such as the proposed C++17 [83], are showing potential to do that. The following listings is an example that may initiate an compound comprising three file operations (i.e., open, read, and close).

```
namespace fs = std::experimental::filesystem;

auto async_open = [](name, flags) { return std::async(fs:open, name, flags); };
auto async_read = [](fd, buf, len) { return std::async(fs:read, fd, buf, len); };
auto async_close = [](fd) { return std::async(fs:close, fd); }

async_open("foo", O_RDONLY)
    .next(async_read, buf, len).unwrap()
        .next(async_close).unwrap().get();
```

Bibliography

[1] A. Bessani and R. Mendes and T. Oliveira and N. Neves and M. Correia and M. Pasin and P. Verissimo. SCFS: A Shared Cloud-backed File System. In *USENIX ATC 14*, pages 169–180. USENIX, 2014.

[2] Abu-Libdeh, Hussam and Princehouse, Lonnie and Weatherspoon, Hakim. RACS: a case for cloud storage diversity. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 229–240. ACM, 2010.

[3] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigraphy. Design tradeoffs for SSD performance. In *Proceedings of the Annual USENIX Technical Conference*, pages 57–70, Boston, MA, June 2008. USENIX Association.

[4] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP. In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[5] Mohammed A. AlZain, Ben Soh, and Eric Pardede. A Survey on Data Security Issues in Cloud Computing: From Single to Multi-Clouds. *Journal of Software*, 8(5):1068–1078, 2013.

[6] Amazon. *Amazon Simple Storage Service Developer Guide API Version2006-03-01*, 2015. `http://docs.aws.amazon.com/AmazonS3/latest/dev/s3-dg.pdf`.

[7] CBS SF Bay Area. Nude celebrity photos flood 4chan after apple icloud hacked, 2014. `http://goo.gl/p5a49Y`.

[8] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010.

[9] B. Harrington, A. Charbon, T. Reix, V. Roqueta, J. B. Fields, T. Myklebust, and S. Jayaraman. NFSv4 test project. In *Linux Simposium*, pages 115–134, 2006.

[10] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-dusseau, and R. H. Arpaci-dusseau. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the Sixth USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose, CA, February 2008. USENIX Association.

[11] A. Batsakis and R. Burns. Cluster delegation: High-performance, fault-tolerant data sharing in NFS. In *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing*. IEEE, July 2005.

[12] Konrad Beiske. Zookeeper—the king of coordination, 2014.

[13] M. Bellare, P. Rogaway, and D. Wagner. EAX: A Conventional Authenticated-Encryption Mode. Cryptology ePrint Archive, Report 2003/069, 2003. `http://eprint.iacr.org/`.

[14] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology – ASIACRYPT 2000*, pages 531–545. Springer, 2000.

[15] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage (TOS)*, 9(4):12, 2013.

[16] Linux bio integrity, 2015. `https://www.kernel.org/doc/Documentation/block/data-integrity.txt`.

[17] Christopher M. Boumenot. The performance of a Linux NFS implementation. Master's thesis, Worcester Polytechnic Institute, May 2002.

[18] Neil Brown. Overlay filesystem, 2015.

[19] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. Haq, M. I. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011.

[20] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification. Technical Report RFC 1813, Network Working Group, June 1995.

[21] M. Chen, D. Hildebrand, G. Kuenning, S. Shankaranarayana, B. Singh, and E. Zadok. Newer is sometimes better: An evaluation of nfsv4.1. In *Proceedings of the 2015 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2015)*, Portland, OR, June 2015. ACM.

[22] M. Chen, D. Hildebrand, G. Kuenning, S. Shankaranarayana, V. Tarasov, A. Vasudevan, E. Zadok, and K. Zakirova. Linux NFSv4.1 Performance Under a Microscope. Technical Report FSL-14-02, Stony Brook University, August 2014.

[23] Ming Chen, John Fastabend, and Eric Dumazet. [BUG?] ixgbe: only num_online_cpus() of the tx queues are enabled, 2014. `http://comments.gmane.org/gmane.linux.network/307532`.

[24] Yao Chen and Radu Sion. To cloud or not to cloud?: musings on costs and viability. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 29. ACM, 2011.

[25] Taehwan Choi and Mohamed G Gouda. Httpi: An http with integrity. In *Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on*, pages 1–6. IEEE, 2011.

[26] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012.

[27] Wei Dai. Crypto++ 5.6.0 Benchmarks. `http://www.cryptopp.com/benchmarks.html`, March 2009.

[28] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007.

[29] P. Deniel. GANESHA, a multi-usage with large cache NFSv4 server. `www.usenix.org/events/fast07/wips/deniel.pdf`, 2007.

[30] Philippe Deniel, Thomas Leibovici, and Jacques-Charles Lafoucrière. GANESHA, a multi-usage with large cache NFSv4 server. In *Linux Symposium*, page 113, 2007.

[31] Data Integrity Extension. `ftp://www.t10.org/t10/document.03/03-111r0.pdf`.

[32] I/O Controller Data Integrity Extensions. `https://oss.oracle.com/~mkp/docs/dix.pdf`.

[33] Dan Dobre, Paolo Viotti, and Marko Vukolić. Hybris: Robust hybrid cloud storage. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[34] J. R. Douceur and J. Howell. EnsemBlue: Integrating Distributed Storage and Consumer Electronics. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 321–334, Seattle, WA, November 2006. ACM SIGOPS.

[35] M. Eisler, A. Chiu, and L. Ling. RPCSEC_GSS protocol specification. Technical Report RFC 2203, Network Working Group, September 1997.

[36] D. Ellard and M. Seltzer. NFS tricks and benchmarking traps. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 101–114, San Antonio, TX, June 2003. USENIX Association.

[37] Sorin Faibish. NFSv4.1 and pNFS ready for prime time deployment, 2011.

[38] Feldman, Ariel J and Zeller, William P and Freedman, Michael J and Felten, Edward W. SPORC: Group Collaboration using Untrusted Cloud Resources. In *OSDI*, pages 337–350, 2010.

[39] Bruce Fields. NFSv4.1 server implementation. `http://goo.gl/vAqR0M`.

[40] Filebench. `http://filebench.sf.net`.

[41] B. Fitzpatrick. Memcached. `http://memcached.org`, January 2010.

[42] The Apache Software Foundation. Apache BookKeeper, 2015.

[43] The Apache Software Foundation. Apache Curator, 2015.

[44] The Apache Software Foundation. Apache Hedwig, 2015.

[45] The Apache Software Foundation. Apache Thrift, 2015.

[46] The Apache Software Foundation. Apache ZooKeeper, 2015.

[47] The Apache Software Foundation. Powered By Apache Thrift, 2015.

[48] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. In *Proceedings of the 4th Usenix Symposium on Operating System Design and Implementation (OSDI '00)*, pages 181–196, San Diego, CA, October 2000. USENIX Association.

[49] NFS-GANESHA. `http://sourceforge.net/apps/trac/nfs-ganesha/`.

[50] G. R. Ganger and M. F. Kaashoek. Embedded inodes and explicit grouping: exploiting disk bandwidth for small files. In *Proceedings of the Annual USENIX Technical Conference*, Anaheim, CA, January 1997. USENIX Association.

[51] G. A. Gibson, D. F. Nagle, W. Courtright II, N. Lanza, P. Mazaitis, M. Unangst, and J. Zelenka. NASD Scalable Storage Systems. In *Proceedings of the 1999 USENIX Extreme Linux Workshop*, Monterey, CA, June 1999.

[52] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. Sirius: Securing remote untrusted storage. In *Proceedings of the Tenth Network and Distributed System Security (NDSS) Symposium*, pages 131–145, San Diego, CA, February 2003. Internet Society (ISOC).

[53] Goodrich, Michael T and Papamanthou, Charalampos and Tamassia, Roberto and Triandopoulos, Nikos. Athos: Efficient authentication of outsourced file systems. *Lecture Notes in Computer Science*, 5222:80–96, 2008.

[54] A. Gulati, M. Naik, and R. Tewari. Nache: Design and Implementation of a Caching Proxy for NFSv4. In *Proceedings of the Fifth USENIX Conference on File and Storage Technologies (FAST '07)*, pages 199–214, San Jose, CA, February 2007. USENIX Association.

[55] M. Halcrow. eCryptfs: a stacked cryptographic filesystem. *Linux Journal*, 2007(156):54–58, April 2007.

[56] D. Hildebrand and P. Honeyman. Exporting storage systems in a scalable manner with pNFS. In *Proceedings of MSST*, Monterey, CA, 2005. IEEE.

[57] J. H. Howard. An Overview of the Andrew File System. In *Proceedings of the Winter USENIX Technical Conference*, February 1988.

[58] Yuchong Hu, Henry C. H. Chen, Patrick P. C. Lee, and Yang Tang. NCCloud: Applying Network Coding for the Storage Repair in a Cloud-of-Clouds. In *Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST '12)*, San Jose, CA, February 2012. USENIX Association.

[59] P. Hunt, M. Konar, J. Mahadev, F. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, 2010.

[60] Amazon Inc. Amazon elastic file system. `https://aws.amazon.com/efs/`, September 2015.

[61] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014.

[62] N. Joukov and J. Sipek. GreenFS: Making Enterprise Computers Greener by Protecting Them Better. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (EuroSys 2008)*, Glasgow, Scotland, April 2008. ACM.

[63] F. Junqueira, B. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 245–256, 2011.

[64] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 29–42, San Francisco, CA, March 2003. USENIX Association.

[65] T. Kojm. ClamAV. `www.clamav.net`, 2015.

[66] H. Krawczyk. The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?). In *Proceedings of CRYPTO'01*. Springer-Verlag, 2001.

[67] H. Krawczyk. Encrypt-then-MAC for TLS and DTLS. `http://www.ietf.org/mail-archive/web/tls/current/msg12766.html`, June 2014.

[68] Eric Kustarz. Using Filebench to evaluate Solaris NFSv4, 2005. NAS conference talk.

[69] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *Proceedings of the Annual USENIX Technical Conference*, pages 213–226, Boston, MA, June 2008. USENIX Association.

[70] C. Lever and P. Honeyman. Linux NFS Client Write Performance. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 29–40, Monterey, CA, June 2002. USENIX Association.

[71] Chuck Lever. Close-to-open cache consistency in the linux NFS client. `http://goo.gl/o9i0MM`.

[72] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 121–136, San Francisco, CA, December 2004. ACM SIGOPS.

[73] L. Lu, A. C. Arpaci-dusseau, R. H. Arpaci-dusseau, and S. Lu. A Study of Linux File System Evolution. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, February 2013. USENIX Association.

[74] M. Eshel and R. Haskin and D. Hildebrand and M. Naik and F. Schmuck and R. Tewari. Panache: A Parallel File System Cache for Global File Access. In *FAST*, pages 155–168. USENIX, 2010.

[75] Mahajan, Prince and Setty, Srinath and Lee, Sangmin and Clement, Allen and Alvisi, Lorenzo and Dahlin, Mike and Walfish, Michael. Depot: Cloud Storage with Minimal Trust. *ACM Trans. Comput. Syst.*, 29(4):12:1–12:38, December 2011.

[76] Ben Martin. Benchmarking NFSv3 vs. NFSv4 file operation performance, 2008. Linux.com.

[77] R. Martin and D. Culler. NFS Sensitivity to High Performance Networks. In *Proceedings of SIGMETRICS*. ACM, 1999.

[78] U. Maurer and B. Tackmann. On the soundness of authenticate-then-encrypt: Formalizing the malleability of symmetric encryption. In *Proceedings of CCS'10*. ACM, 2010.

[79] Alex McDonald. The background to NFSv4.1. *;login: The USENIX Magazine*, 37(1):28–35, February 2012.

[80] Paul E. McKenney. Stochastic fairness queueing. In *INFOCOM'90*, pages 733–740. IEEE, 1990.

[81] Peter Mell and Tim Grance. The NIST definition of cloud computing. Technical report, Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, 2011.

[82] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO'87, pages 369–378, London, UK, 1988. Springer-Verlag.

[83] Bartosz Milewski. C++17: I See a Monad in Your Future!, 2015.

[84] E. Miller, W. Freeman, D. Long, and B. Reed. Strong security for network-attached storage. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST '02)*, pages 1–13, Monterey, CA, January 2002. USENIX Association.

[85] Y. Miretskiy, A. Das, C. P. Wright, and E. Zadok. Avfs: An On-Access Anti-Virus File System. In *Proceedings of the 13th USENIX Security Symposium (Security 2004)*, pages 73–88, San Diego, CA, August 2004. USENIX Association.

[86] J. Molina, J. Gordon, X. Chorin, and M. Cukier. An empirical study of filesystem activity following a SSH compromise. In *Information, Communications Signal Processing, 6th International Conference on*, pages 1–5, 2007.

[87] CNN Money. Hospital network hacked, 2014. `http://money.cnn.com/2014/08/18/technology/security/hospital-chs-hack/`.

[88] Trond Myklebust. File creation speedups for NFSv4.2, 2010.

[89] NetApp. NetApp SteelStore Cloud Integrated Storage Appliance. `http://www.netapp.com/us/products/protection-software/steelstore/`, 2014.

[90] NetApp. Netapp altavault cloud-integrated storage, 2015. `http://www.netapp.com/us/products/protection-software/altavault/index.aspx`.

[91] Linux-IO Target, 2015. `https://lwn.net/Articles/592093/`.

[92] End-to-end Data Integrity For NFSv4, 2014. `http://tools.ietf.org/html/draft-cel-nfsv4-end2end-data-protection-01`.

[93] Arun Olappamanna Vasudevan. Finding the Right Balance: Security vs. Performance with Network Storage Systems. Master's thesis, Stony Brook University, May 2015. Technical Report FSL-15-01, `http://www.fsl.cs.sunysb.edu/docs/arun-msthesis/arun-msthesis.pdf`.

[94] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. Rumble, E. Stratmann, and R. Stutsman. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, 2010.

[95] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. big web services: Making the right architectural decision. In *Proceedings of the 17th International conference on World Wide Web*, pages 805–814, New York, NY, April 2008. ACM.

[96] Bob Peterson. The red hat cluster suite nfs cookbook. Technical report, Red Hat, Inc., July 2010.

[97] Data integrity user-space interfaces, 2014. `https://lwn.net/Articles/592093/`.

[98] Popa, Raluca Ada and Lorch, Jacob R and Molnar, David and Wang, Helen J and Zhuang, Li. Enabling Security in Cloud Storage SLAs with CloudProof. In *USENIX Annual Technical Conference*, 2011.

[99] A. Purohit, C. Wright, J. Spadavecchia, and E. Zadok. Cosy: Develop in User-Land, Run in Kernel Mode. In *Proceedings of the 2003 ACM Workshop on Hot Topics in Operating Systems (HotOS IX)*, pages 109–114, Lihue, Hawaii, May 2003. USENIX Association.

[100] P. Radkov, L. Yin, P. Goyal, P. Sarkar, and P. Shenoy. A performance comparison of NFS and iSCSI for IP-networked storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 101–114, San Francisco, CA, March/April 2004. USENIX Association.

[101] Prabu Rambadran. Announcing Nutanix cloud connect, 2014. `http://www.nutanix.com/2014/08/19/announcing-nutanix-cloud-connect/`.

[102] Peter Reiher, John Heidemann, David Ratner, Greg Skinner, and Gerald Popek. Resolving File Conflicts in the Ficus File System. In *Proceedings of the Summer USENIX Conference*, pages 183–195, 1994.

[103] R. Richardson. CSI Computer Crime and Security Survey. *Computer Security Institute*, 15(1):17, 2010/2011. `http://gatton.uky.edu/FACULTY/PAYNE/ACC324/CSISurvey2010.pdf`.

[104] Riverbed. SteelFusion. `http://www.riverbed.com/products/branch-office-data/`, 2012.

[105] Scott Rixner. Network virtualization: Breaking the performance barrier. *Queue*, 6(1):37:36–37:ff, Jan 2008.

[106] D. Roselli, J. R. Lorch, and T. E. Anderson. A comparison of file system workloads. In *Proceedings of the Annual USENIX Technical Conference*, pages 41–54, San Diego, CA, June 2000. USENIX Association.

[107] S. Shepler and M. Eisler and D. Noveck. NFS Version 4 Minor Version 1 Protocol. Technical Report RFC 5661, Network Working Group, January 2010.

[108] R. Sandberg. The Sun network file system: Design, implementation and experience. Technical report, Sun Microsystems, 1985.

[109] Select Real Security. Scan files for malicious software, 2015.

[110] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS Version 4 Protocol. Technical Report RFC 3530, Network Working Group, April 2003.

[111] S. Shepler, M. Eisler, and D. Noveck. NFS Version 4 Minor Version 1. Technical Report RFC 5661, Network Working Group, January 2010.

[112] Shraer, Alexander and Cachin, Christian and Cidon, Asaf and Keidar, Idit and Michalevsky, Yan and Shaket, Dani. Venus: Verification for Untrusted Cloud Storage. In *Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop*, CCSW '10, pages 19–30. ACM, 2010.

[113] Kapil Singh, H Wang, Alexander Moshchuk, Collin Jackson, and Wenke Lee. Httpi for practical end-to-end web content integrity. In *Microsoft technical report*. Microsoft, 2011.

[114] S. Smaldone, A. Bohra, and L. Iftode. FileWall: A Firewall for Network File Systems. In *Dependable, Autonomic and Secure Computing, Third IEEE International Symposium on*, pages 153–162, 2007.

[115] SoftNAS. SoftNAS Cloud. `https://www.softnas.com/wp/`.

[116] SPEC. SPECsfs2008. `www.spec.org/sfs2008`, 2008.

[117] Emil Stefanov, Marten van Dijk, Ari Juels, and Alina Oprea. Iris: A scalable cloud file system with efficient integrity checks. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 229–238. ACM, 2012.

[118] H. L. Stern and B. L. Wong. NFS performance and network loading. In *Proceedings of the Sixth USENIX Systems Administration Conference (LISA VI)*, pages 33–38, Long Beach, CA, October 1992. USENIX Association.

[119] Sun Microsystems. NFS: Network file system protocol specification. Technical Report RFC 1094, Network Working Group, March 1989.

[120] T. Haynes. NFS Version 4 Minor Version 2 Protocol. Technical report, Network Working Group, September 2015. `https://tools.ietf.org/html/draft-ietf-nfsv4-minorversion2-39`.

[121] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[122] V. Tarasov, S. Bhanage, E. Zadok, and M. Seltzer. Benchmarking File System Benchmarking: It *IS* Rocket Science. In *Proceedings of HotOS XIII:The 13th USENIX Workshop on Hot Topics in Operating Systems*, Napa, CA, May 2011.

[123] Vasily Tarasov, Dean Hildebrand, Geoff Kuenning, and Erez Zadok. Virtual machine workloads: The case for new benchmarks for nas. Technical Report FSL-12-06, Stony Brook University Computer Science, September 2012.

[124] Vasily Tarasov, Dean Hildebrand, Geoff Kuenning, and Erez Zadok. Virtual machine workloads: The case for new benchmarks for NAS. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, February 2013. USENIX Association.

[125] Ted Krovetz and Wei Dai. VMAC: Message Authentication Code using Universal Hashing. Technical report, CFRG Working Group, April 2007. `http://www.fastcrypto.org/vmac/draft-krovetz-vmac-01.txt`.

[126] Alexander Thomson and Daniel J. Abadi. CalvinFS: Consistent WAN Replication and Scalable Metadata Management for Distributed File Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, 2015.

[127] Joseph Tsidulko. The 10 Biggest Cloud Outages Of 2015 (So Far), 2015.

[128] Vijay Vasudevan, David G. Andersen, and Michael Kaminsky. Os support for high-performance nvms using vector interfaces. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, February 2011. USENIX Association.

[129] Vijay Vasudevan, Michael Kaminsky, and David G. Andersen. Using vector interfaces to deliver millions of iops from a networked key-value storage server. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*, SoCC '12, 2012.

[130] Kode Vicious. Bound by the speed of light. *Queue*, 8(12), December 2010.

[131] vinf.net. Silent data corruption in the cloud and building in data integrity, 2011. `http://goo.gl/IbMyu7`.

[132] Michael Vrable, Stefan Savage, and Geoffrey M Voelker. Bluesky: a cloud-backed file system for the enterprise. In *FAST*, page 19, 2012.

[133] Doug Whiting, Russ Housley, and Niels Ferguson. Counter with CBC-MAC (CCM). Technical Report RFC 3610, Network Working Group, September 2003.

[134] Wikipedia. Nagle's algorithm. `http://en.wikipedia.org/wiki/Nagle's_algorithm`.

[135] Wikipedia. Authenticated Encryption, 2015. `https://en.wikipedia.org/wiki/Authenticated_encryption`.

[136] A. W. Wilson. Operation and implementation of random variables in Filebench.

[137] M. Wittle and B. E. Keith. LADDIS: The next generation in NFS file server benchmarking. In *Proceedings of the Summer USENIX Technical Conference*, pages 111–128, Cincinnati, OH, June 1993. USENIX Association.

[138] Network World. Which cloud providers had the best uptime last year?, 2014. `http://goo.gl/SZOKUT`.

[139] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok, and M. N. Zubair. Versatility and unix semantics in namespace unification. *ACM Transactions on Storage (TOS)*, 2(1):1–32, February 2006.

[140] C. P. Wright, M. Martino, and E. Zadok. NCryptfs: A secure and convenient cryptographic file system. In *Proceedings of the Annual USENIX Technical Conference*, pages 197–210, San Antonio, TX, June 2003. USENIX Association.

[141] SGI XFS. xfstests. `http://xfs.org/index.php/Getting_the_latest_source_code`.

[142] N. Yezhkova, L. Conner, R. Villars, and B. Woo. *Worldwide enterprise storage systems 2010–2014 forecast: recovery, efficiency, and digitization shaping customer requirements for storage systems*. IDC, May 2010. IDC #223234.

[143] Zadara Storage. Virtual Private Storage Array. https://www.zadarastorage.com/.

[144] E. Zadok, I. Bădulescu, and A. Shender. Extending file systems using stackable templates. In *Proceedings of the Annual USENIX Technical Conference*, pages 57–70, Monterey, CA, June 1999. USENIX Association.