Kurma: Efficient and Secure Multi-Cloud Storage Gateways for Network-Attached Storage

A Dissertation 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-17-01

April 2017

Abstract

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 combines 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 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 gateway (middleware) 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 gateways 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 gateways. Kurma uses multiple clouds as backends for higher availability, and splits data among clouds using secret sharing for higher confidentiality. Kurma can also efficiently detect stale data caused by replay attacks or due to the eventual consistency nature of clouds.

We have thoroughly benchmarked the in-kernel NFSv4.1 implementation and improved its performance by up to 11×. Taking advantage of NFSv4.1 compound procedures, we have designed and implemented a vectorized file-system API and library (called vNFS) that can further boost NFS performance by up to two orders of magnitude. Assuming a public cloud supporting NFSv4, we have designed and implemented an early Kurma prototype (called SeMiNAS) with a performance penalty of less than 18%, while still protecting integrity and confidentiality of files.

Based on SeMiNAS, we developed Kurma which uses real public clouds including AWS S3, Azure Blob Store, Google Cloud Storage, and Rackspace Cloud Files. Kurma reliably stores files in multiple clouds with replication, erasure coding, or secret sharing to tolerate cloud failures. To share files among clients in geo-distributed offices, Kurma maintains a unified file-system namespace across geo-distributed gateways. Kurma keeps file-system metadata on-premises and encrypts data blocks before writing them to clouds. In spite of the eventual consistency of clouds, Kurma ensures data freshness using an efficient scheme that combines versioning and timestamping. Our evaluation showed that Kurma's performance is around 52–91% that of a local NFS server while providing geo-replication, confidentiality, integrity, and high availability.

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

Contents

List of Figures

List of Tables

Acknowledgments

This thesis 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 Tinglan 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 security features in SeMiNAS; Vishnu Vardhan Rajula helped implement SeMiNAS namespace encryption. Garima Gehlot, Henry Nelson, Jasmit Saluja, Bharat Singh, and Ashok Sankar Harihara Subramony helped me with the vNFS implementation and benchmarking; Farhaan Jalia and Geetika Babu Bangera are carrying on the work of improving vNFS; Vishal Sahu helped explore the methods of adding transaction support into vNFS. Praveen Kumar Morampudi helped me with testing and bug-fixing in the Kurma NFS Server; Harshkumar Patel helped implement Kurma conflict resolution and Rushabh Shah helped implement and test Kurma metadata transaction library; Shivanshu Goswami, Rushabh Shah, and Mukul Sharma are carrying on the work of improving Kurma. Their help is very valuable to me. It would be much, much harder for me to finish all these tasks all by myself.

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, Dongju Ok, Justin Seyster, Vasily Tarasov, Leixiang Wu, and Sun (Jason) Zhen.

I am also grateful to external collaborators on this thesis including Prof. Geoff Kuenning from Harvey Mudd College and Dr. Dean Hildebrand from IBM Research—Almaden. They are very knowledgeable and helped this thesis 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 Profs. Anshul Gandhi, Donald Porter, and Scott Stoller for serving on the committee of my Ph.D. thesis and providing invaluable advices. This work was made possible in part thanks to NSF awards CNS-1223239, CNS-1251137, CNS-1302246, CNS-1305360, CNS-1622832, ONR award N00014-16-1-2264, and gifts from EMC, NetApp, and IBM.

Chapter 1

Introduction

Cloud storage has many desirable traits including high accessibility (from multiple devices, at multiple locations), availability, flexibility, scalability, and cost-effectiveness [10, 123, 199]. For instance, clouds eliminate up-front commitment by users and allows pay-as-you-go; clouds also make most infinite computing resources available on demand [10]. However, cloud storage providers need to improve the integrity and confidentiality of customer data. Some customer data got silently corrupted in the cloud [187]. Silent data corruption could be disastrous, especially in health-care 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 [9] and theft of patient records [130]. Data in cloud are lost due to internal bugs [28,40,84,126], and leaked [131], and modified [30]

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 cloud's eventual-consistency semantics [42]. 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 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 [133,134,150] 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 [133, 134, 150] use public clouds as a separate tier for specific workloads such as backup or web-tier hosting. Hybrid clouds, although enjoying strong security and high performance for generic workloads, were less studied than public clouds.

This thesis focuses on efficient and secure hybrid-cloud storage solutions that seamlessly support traditional NAS-based applications. As hybrid-cloud storage systems involve both private and public clouds, this thesis studies both types of clouds. For private clouds, we focus on the Net-

work File System (NFS)—the standard NAS protocol; for public clouds, we focus on cloud storage gateways. We began this work by benchmarking NFSv4.1—the latest version of NFS with new features including compound procedures and delegations. We compared NFSv4.1 performance to NFSv3 in both local- and wide-area networks under a large number of workloads. We found NFSv4.1 has comparable performance to NFSv3: NFSv4.1 is more talkative with a stateful protocol, but NFSv4.1 also enables higher concurrency through asynchronous RPC calls. During the benchmarking, we also found and fixed a number of bugs in Linux's NFS implementation; one of our bug fixes in the Linux NFSv4.1 client improved workload performance by up to 11×.

Our benchmarking study also showed that NFSv4.1 compound procedures have the potential to significantly improve performance but are underused because of the limited POSIX file-system API [32]. To overcome the limitations, we then proposed a vectorized file-system API and implemented the API using a user-space library and an NFS client called *vNFS*. The vectorized API allows many file-system operations to be performed in batch using a single network round trip. This batching amortizes high network latency and is especially important after years of significant network improvement in bandwidth but not in latency. The vNFS API is not only efficient but also convenient to use with higher semantics including automatic file opening, atomic file appending, and file copying. We found it easy to modify several Unix utilities, an HTTP/2 server, and the benchmarking tool Filebench [56] to use vNFS. We evaluated vNFS under a wide range of workloads and network latency conditions, showing that vNFS improves performance even for low-latency networks. On high-latency networks, vNFS can improve performance by up to two orders of magnitude.

After evaluating network storage (NAS) performance through benchmarking and then improving this performance in the private cloud side, we proceeded to the public cloud side and developed two cloud storage gateway systems. The first gateway system is *SeMiNAS*, which is an early prototype of the second system Kurma. SeMiNAS allows files to be securely outsourced to cloud and be shared among geo-distributed clients. SeMiNAS also explores the possibility of using NFSv4.1 for both client-to-gateway and gateway-to-cloud communications. SeMiNAS provides end-to-end data integrity and confidentiality that protects data from not only attacks during data transmission over the Internet, but also from misbehaving clouds. Data stays in encrypted form in the cloud, and is not decrypted until clients retrieve the data from clouds. SeMiNAS leverages advanced NFSv4 features, including compound procedures and Data-Integrity eXtensions (DIX), to minimize network round trips caused by security metadata. SeMiNAS caches remote files locally to reduce accesses to providers over WANs. We designed, implemented, and evaluated SeMiNAS, which demonstrates a small performance penalty of less than 26% and an occasional performance boost of up to 19% for Filebench workloads.

After SeMiNAS, we further developed *Kurma*, our final secure cloud storage gateway system. SeMiNAS and Kurma have similar architectures and on-premises caches; they also share a simple key-exchange scheme that allows per-file encryption keys to be exchanged securely among geo-distributed gateways without relying on any trusted third-party. Kurma is our final secure cloud storage gateway system. Similar to SeMiNAS, Kurma also provides end-to-end data integrity and confidentiality while outsourcing data storage to untrusted public clouds. However, Kurma improves SeMiNAS in three unique ways:

1. SeMiNAS assumes that public clouds support NFSv4.1 DIX. Although DIX is a forward-looking feature improving security and performance, it is not standardized yet. In contrast,

Kurma communicates to real clouds including Amazon AWS, Microsoft Azure, Google Cloud, and Rackspace using their native object store APIs.

2. SeMiNAS uses only a single public cloud. Although most public cloud providers have high availability close to five nines (99.999%) [199], availability still remains the largest obstacle for cloud computing [10]. A single cloud provider is itself a single point of failure [10]; once it is out of service, there is little tenants can do but wait for the cloud to come back up. Therefore, Kurma stores data across multiple clouds using replication, erasure coding, or secret sharing. Kurma can thus tolerate failure of a single or more clouds.

3. SeMiNAS is susceptible to stale data when clouds return old versions of data because of eventual consistency or malicious replay attacks. Kurma solves this problem and can detect stale data by maintaining a version number of each data block. Kurma synchronizes the version numbers of data blocks among all geo-distributed gateways so that clients can check freshness of files after they are modified by remote Kurma gateways.

Kurma uses public clouds as block stores. It stores only encrypted and authenticated file data blocks on clouds while keeping all metadata in trusted gateways. The metadata includes file-system namespace, file block mapping, per-file encryption keys, and integrity metadata of file data blocks. Therefore, Kurma is secure against side-channel attacks that exploit information such as file-access patterns. Each Kurma gateway maintains a copy of the whole file-system metadata, so that it can still serve files to local clients after network failures separate it from other gateways. The metadata changes made by a Kurma gateway are asynchronously replicated to all other gateways.

To simplify file sharing among distant clients, Kurma also maintains a unified file-system namespace across geo-distributed gateways. Kurma provides NFS close-to-open consistency among clients connected to the local Kurma gateway, which is the same as traditional network-attached storage (NAS). For clients across geo-distributed gateways, Kurma trades off consistency for higher performance and availability, and provides FIFO consistency [109]. This means that operations in different gateways may be conflicting. Kurma can reliably detect conflicts and contains a framework for resolving them. We have implemented several default resolution policies for common conflicts. We have implemented and evaluated a Kurma prototype. Our evaluation shows that Kurma performance is around 52–91% that of a local NFS server while providing geo-replication, confidentiality, integrity, and high availability.

Our thesis is that cloud storage can be both efficient and secure for many generic workloads, and it seamlessly integrate with traditional NAS-based systems. In summary, this thesis includes five major contributions:

- A comprehensive and in-depth performance analysis of NFSv4.1 and its unique features (statefulness, compounds, sessions, delegations, etc.) by comparison to NFSv3 under low- and high-latency networks, using a wide variety of micro- and macro-workloads.

- An NFSv4.1-compliant client that exposes a vectorized high-level file-system API and leverages NFS compounds to improve performance by up to two orders of magnitude.

- A forward-looking and secure cloud storage gateway system that uses advanced NFSv4.1 features (compound procedures and DIX) in both private and public clouds.

- A realistic and highly-available cloud gateway system that allows geo-distributed clients to store and share data in multiple public clouds in a seamless, secure, and efficient manner.
- An efficient security scheme that ensures data integrity, confidentiality, and data freshness without using traditional Merkle trees [124], which are expensive in cloud environments.

The rest of this thesis is organized as follows. Chapter 2 presents the performance benchmarking of NFSv4.1. Chapter 3 describes our vNFS client that maximizes NFS performance using compounds and vectorized I/Os. Chapter 4 details the design, implementation, and evaluation of SeMiNAS. Chapter 5 describes the design of Kurma—our final multi-cloud geo-distributed secure cloud gateway system. Chapter 6 concludes this thesis and discusses future work.

Chapter 2

Benchmarking Network File System

2.1 NFS Introduction

Before the cloud era, over 90% of enterprise storage capacity was served by network-based storage [204], and Network File System (NFS) represents a significant proportion of that total [179]. NFS has become a highly popular network-storage solution since its introduction more than 30 years ago [159]. 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 gateways 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 [10]. 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 [159]. 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 [159,175], 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 [158]; (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 [78, 158]. NFS's evolution does not stop after NFSv4.1; NFSv4.2 is under development with many new features and optimizations [77] already proposed.

V4.1 became ready for production deployment only a couple of years ago [53, 121]. 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 [121], 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 [32].

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. Chapter 2.2 describes our benchmarking methodology. Chapters 2.3 and 2.4 discuss the results of data- and metadata-intensive workloads, respectively. Chapter 2.5 explores NFSv4's delegations. Chapter 2.6 examines macro-workloads using Filebench. Chapter 2.7 overviews related work. We conclude and discuss limitations in Chapter 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 (Chapter 2.3);
2. Metadata-intensive micro-workloads that exercise NFS's handling of file metadata and small messages (Chapter 2.4);
3. Micro-workloads that evaluate delegations, which are V4's new client-side caching mechanism (Chapter 2.5); and
4. Complex macro-workloads that represent real-world applications (Chapter 2.6).

2.3 Benchmarking 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.

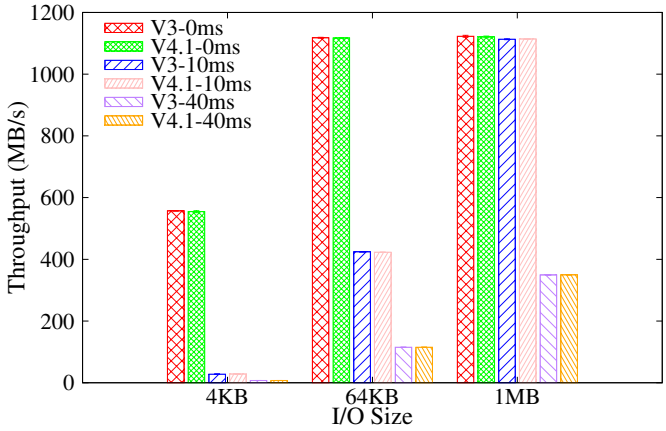


Figure 2.1: Random-read throughput with 16 threads and different network delays (varying I/O size).

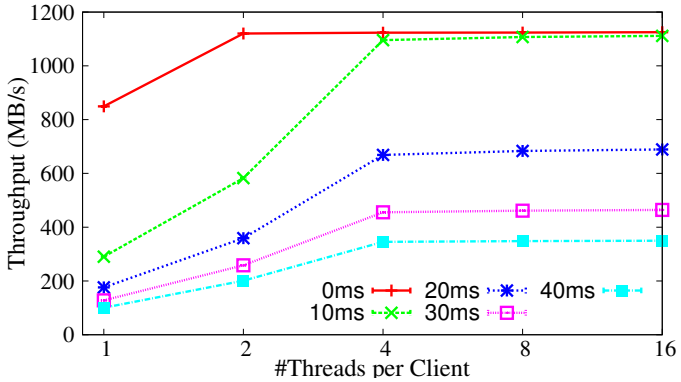


Figure 2.2: Random-read throughput with 1MB I/O size, default 2MB TCP maximum buffer size, and different network delays (varying the number of threads per client).

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

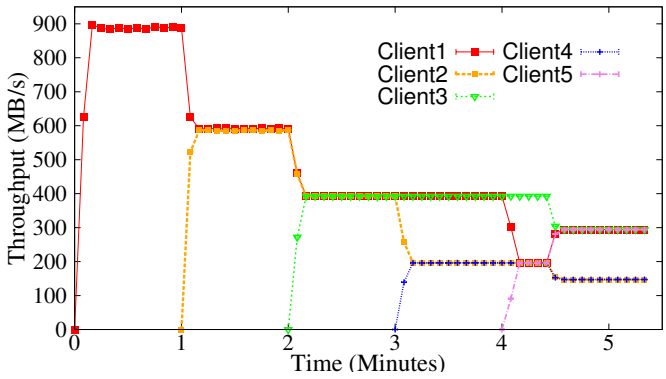


Figure 2.3: Sequential-read throughputs of individual clients when they were launched one after the other at an interval of one minute. Throughput results of one run of experiments; the higher the better. When all five clients are running (after 4 minutes), the throughputs of the five identical clients 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 irrelevant 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.

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 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

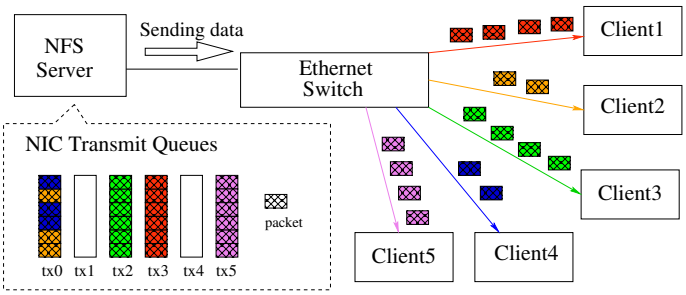


Figure 2.4: Illustration of Hash-Cast. The NIC of the NFS server has six transmit queues (tx). The NFS server is sending data to five clients using one TCP flow for each client. Linux hashes the TCP flows of Client1, Client3, and Client5 into tx3, tx2, and tx5, respectively; and hashes the flows of both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

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 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 [155]. 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 [155]. 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%).

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 [36]. 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 [122]. 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 Chapter 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 Chapter 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 [179], 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 2.5 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 metadata 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 metadata directly needed to retrieve that data are written synchronously; other metadata

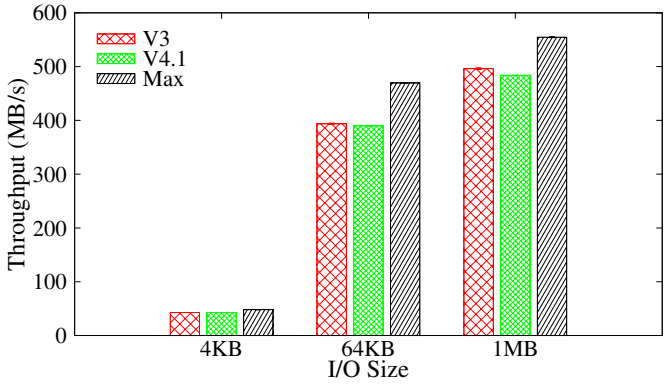


Figure 2.5: Random-write throughput in a zero-delay network with different I/O size. The higher the better. "Max" is the throughput when running the workload directly on the server side without using NFS. The "Max" throughput serves as a baseline when evaluating NFS's overhead.

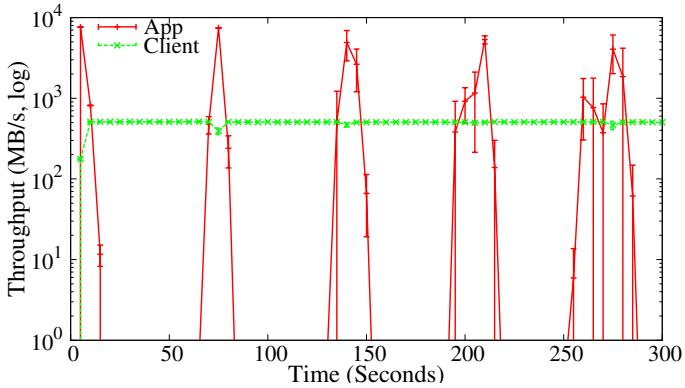


Figure 2.6: Random-write throughput of a single NFS client in a zero-delay network (log_{10}). The higher the better. The "App" curve is the throughput observed by the benchmarking application, i.e., the writing speed of the application to the file; the "Client" curve is the throughput observed by the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

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 metadata. 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 metadata updates in NFS, but not in `ext4`. The overhead of those extra updates was aggravated by `ext4`'s journaling of metadata 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

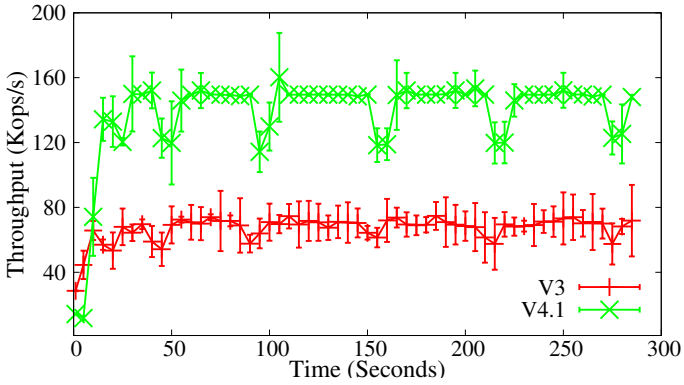


Figure 2.7: Throughput of reading small files with one thread in a zero-delay network.

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.

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 Chapter 2.3.3 had an even stronger effect if the backend storage used HDDs, because the small disk writes generated by the metadata 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 [33].

2.4 Benchmarking 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 metadata-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 metadata 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

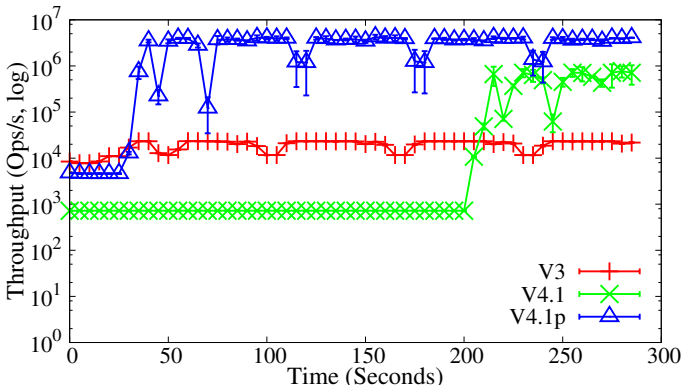


Figure 2.8: Aggregate throughput of reading small files with 16 threads in a 10ms-delay network ($\log_{10}$). The "V4.1" curve is the throughput of vanilla NFSv4.1; the "V4.1p" curve is the throughput of the patched NFSv4.1 with our fix.

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 to be used without revalidation. We discuss and evaluate V4's delegations in greater detail in Chapter 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 [158]; it was used by V4.0 clients to notify the 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 [158] 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 metadata requests when reading small files. To exercise NFS's metadata 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), 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 Chapter 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.

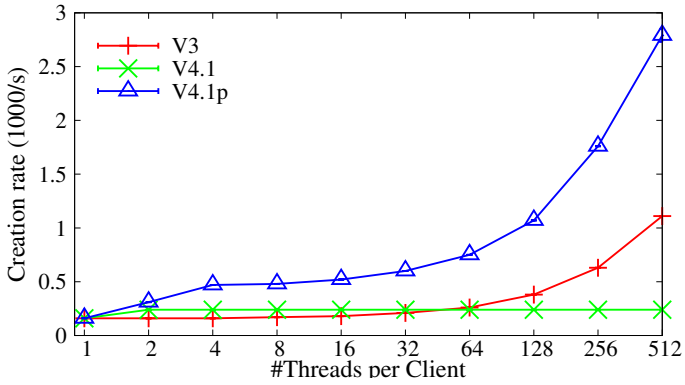


Figure 2.9: Aggregate throughput of creating empty files in a 10ms-delay network with different numbers of threats per client.

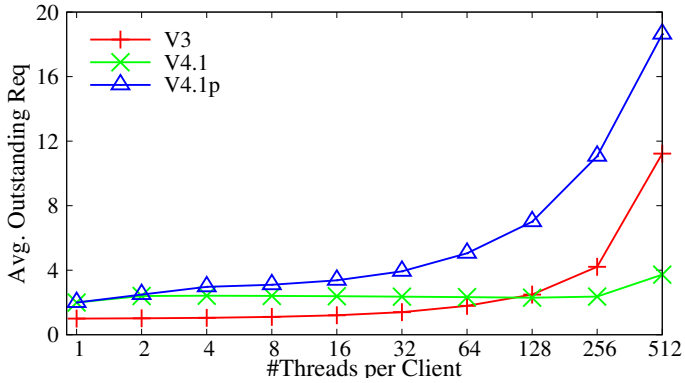


Figure 2.10: Average number of outstanding requests when creating empty files in a 10ms-delay network.

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

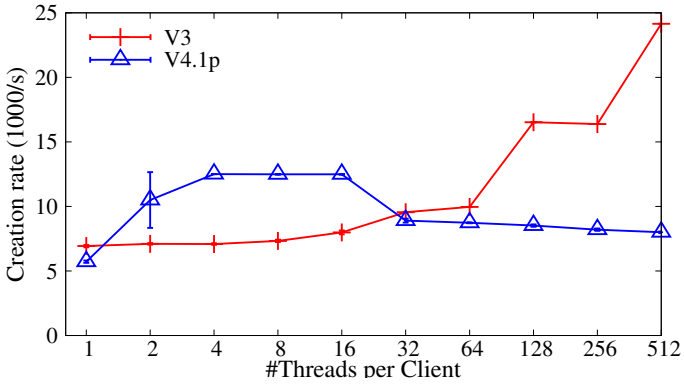


Figure 2.11: Rate of creating empty files in a zero-delay network.

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 [132].

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 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× 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 [176]. In addition to V4.1p's higher wait time in this figure, the high standard deviation means that it 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

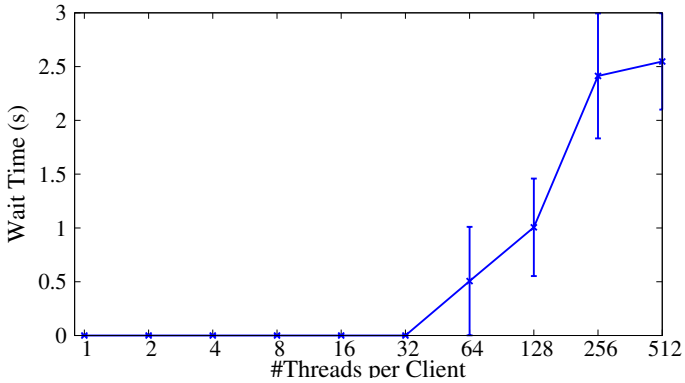


Figure 2.12: Average waiting time for V4.1p's session slots of ten experimental runs. Error bars are standard deviations.

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 Chapter 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 metadata-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 (Chapter 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.

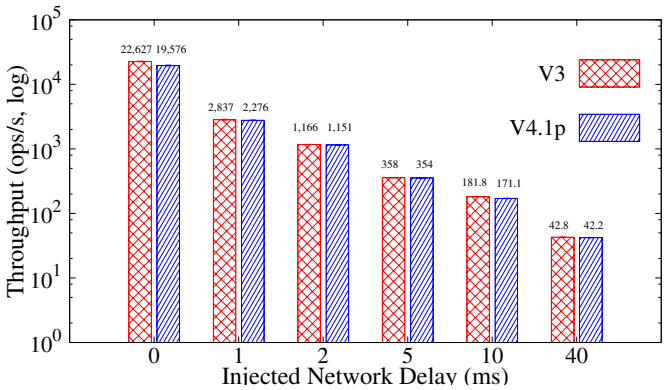


Figure 2.13: Directory listing throughput ($\log_{10}$).

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 Benchmarking 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 [31, 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 [158] and rarely concurrent [101]. 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 [55].

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

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).

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 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 [73] 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 [103]. 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 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 [179], 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 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 metadata upon read. The latter GETATTRs were needed because the locking preceding the read invalidated both the data and metadata 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.

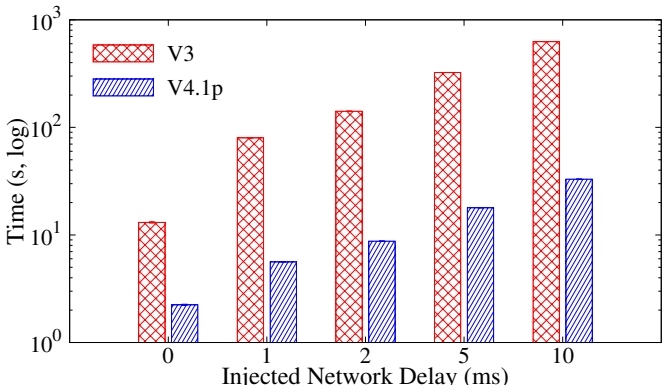


Figure 2.14: Running time of the locked-reads experiment ($\log_{10}$). Lower is better.

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 [5]. 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

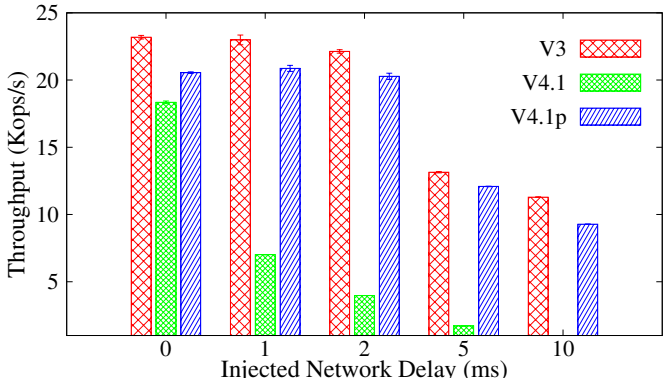


Figure 2.15: File Server throughput (varying network delay)

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.

2.6 Benchmarking Macro-Workloads

We now turn to macro-workloads that mix data and metadata 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 [100].

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 [196].

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.

V4's verbosity hurts its performance, especially in high-latency networks. We observed the same problems in other workloads such as small-file reading (Chapter 2.4.1), where V4 was 40%

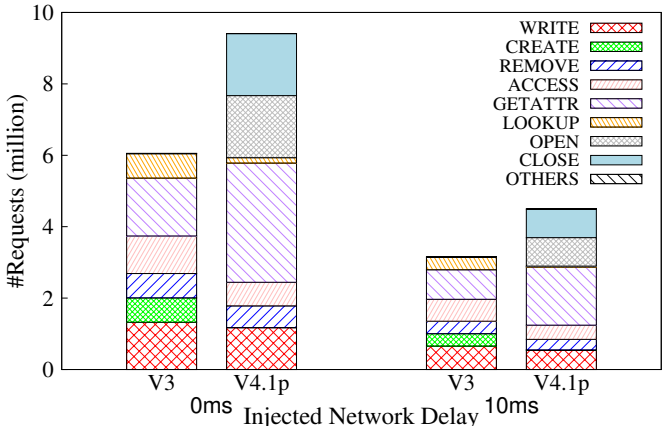


Figure 2.16: Number of NFS requests made by the File Server

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 Chapter 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

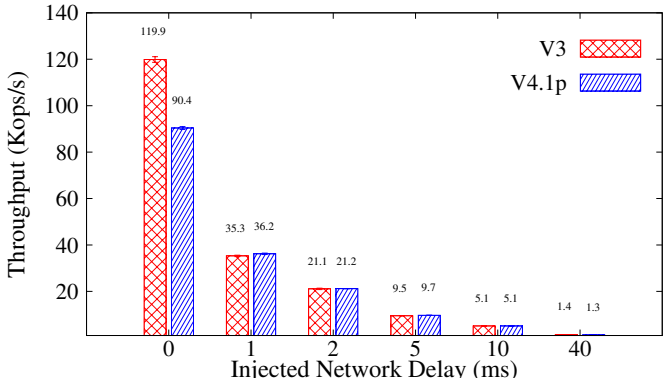


Figure 2.17: Web Server throughput (varying network delay).

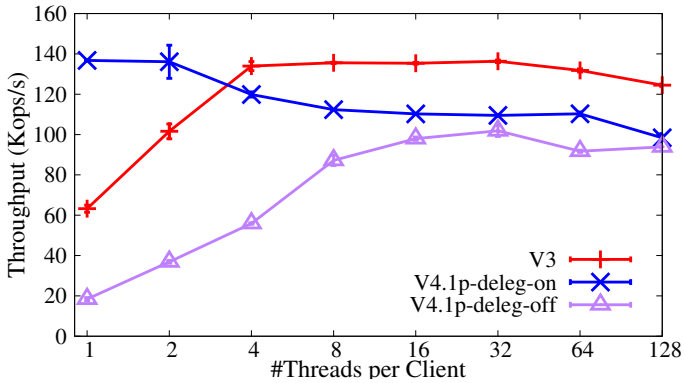


Figure 2.18: Web Server throughput in the zero-delay network (varying thread count per client).

RPC RTT became overshadowed by the delay, and V4.1p's performance became close to V3's 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 Chapter 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 `open`s and `close`s 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

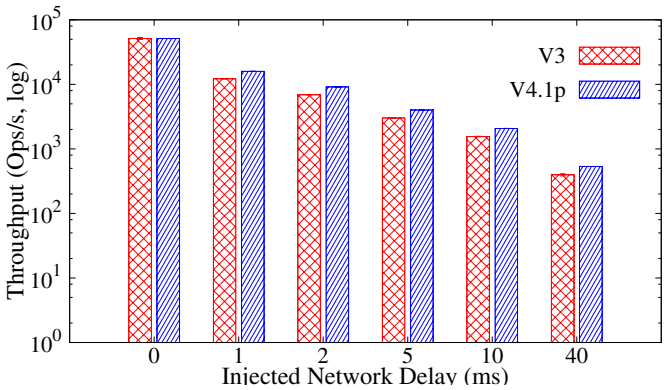


Figure 2.19: Mail Server throughput (varying network delay)

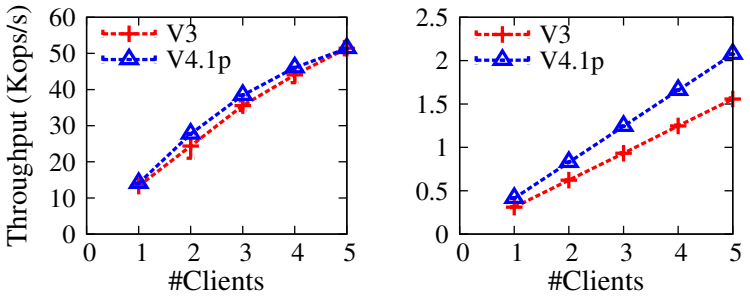
2.2× faster than V3 when using one thread. However, V4.1p began to slow down with 4 threads, whereas V3 sped up and did not slow down until the thread number increased to 64. The eventual slowdown of both V3 and V4.1p was because the system became overloaded when the log-writing bottleneck was hit. However, 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 Chapter 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.

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 Chapter 2.6.1); and (2) V4.1p's operations were more complex and had longer RPC round-trip times (see Chapter 2.4.3); but (3) V4.1p made many asynchronous RPC calls and helped the networking algorithms coalesce RPC messages (see Chapter 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



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

Figure 2.20: Mail Server throughput (varying client count)

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 numbers 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 of NFS Performance Benchmarking

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 [174]. Wittle and Keith designed the LADDIS NFS workload that overcomes nhfsstone's drawback, and measured NFS response time and throughput under various loads [197]. Based on LADDIS, the SPECsfs suites were designed to benchmark and compare the performance of different NFS server implementations [171]. Martin and Culler [118] 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 metadata operations. Ellard and Seltzer designed a simple sequential-read workload to benchmark and improve NFS's readahead algorithm [52]; 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 [26] 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 [102].

Most prior studies [26, 52, 102, 118, 171, 197] 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 [13] implementation. Radkov et al. compared the performance of a prototype version of V4.0 and iSCSI in IP-networked storage [149]. Martin [117] compared the file operation performance between Linux V3 and V4.0; Kustarz [100] 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. [112] 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 [16]. Gulati et al. built a V4.0 cache proxy, also using delegations, to improve NFS's performance in WANs [73]. 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 [55]. 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 [79], 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 Benchmarking 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 [158], 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

vNFS: Maximizing NFS Performance with Compounds and Vectorized I/O

Modern systems use networks extensively, accessing both services and storage across local and remote networks. Latency is a key performance challenge, and packing multiple small operations into fewer large ones is an effective way to amortize that cost, especially after years of significant improvement in bandwidth but not latency. To this end, the NFSv4 protocol supports a *compounding* feature to combine multiple operations. However, as shown by our benchmarking study in Chapter 2, compounding has been underused because the synchronous POSIX file-system API issues only one (small) request at a time.

In this chapter, we propose *vNFS*, an NFSv4.1-compliant client that exposes a vectorized high-level API and leverages NFS *compound procedures* to maximize performance. We designed and implemented vNFS as a user-space RPC library that supports an assortment of bulk operations on multiple files and directories.

3.1 vNFS Introduction and Background

Modern computer hardware supports high parallelism: a smartphone can have eight cores and a NIC can have 256 queues. Although parallelism can improve throughput, many standard software protocols and interfaces are unable to leverage it and are becoming bottlenecks due to serialization of calls [32, 75]. Two notable examples are HTTP/1.x and the POSIX file-system API, both of which support only one synchronous request at a time (per TCP connection or per call). As Moore's Law fades [190] and the focus shifts to adding cores, it is increasingly important to make these protocols and interfaces parallelism-friendly. For example, HTTP/2 [21] added support for sending multiple requests per connection. However, to the best of our knowledge little progress has been made on the file-system API.

In this chapter we similarly propose to batch multiple file-system operations per call. We focus particularly on the Network File System (NFS), and study how much performance can be improved by using a file-system API friendly to NFSv4 [158, 164]; this latest version of NFS supports *compound procedures* that pack multiple operations into a single RPC so that only one round trip is needed to process them. Unfortunately, although NFS compounds have been designed, standardized, and implemented in most NFS clients and servers, they are severely underutilized—

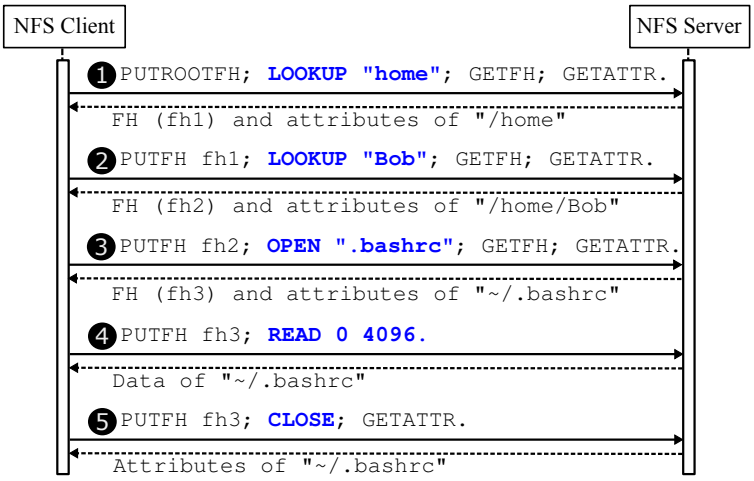


Figure 3.1: NFS compounds used by the in-kernel NFS client to read a small file. Each numbered request is one compound, with its operations separated by semicolons. The operations use an NFSv4 server-side state, the current filehandle (CFH). PUTROOTFH sets the CFH to the FH of the root directory; PUTFH and GETFH set or retrieve the CFH; LOOKUP and OPEN assume that the CFH is a directory, find or open the specified name inside, and set it as the CFH; GETATTR, READ, and CLOSE all operate on the file indicated by the CFH.

mainly because of the limitations of the low-level POSIX file-system interface [32].

To explain the operations and premise of NFS4's compound procedures, we discuss them using several instructive figures. We start with Figure 3.1, which shows how reading a small file is limited by the POSIX API. This simple task involves four syscalls (`stat`, `open`, `read`, and `close`) that generate five compounds, each incurring a round trip to the server. Because compounds are initiated by low-level POSIX calls, each compound contains only one significant operation (in bold blue), with the rest being trivial operations such as PUTFH and GETFH. Compounds reduced the number of round trips a bit by combining the syscall operations (LOOKUP, OPEN, READ) with NFSv4 state-management operations (PUTFH, GETFH) and attribute retrieval (GETATTR), but the syscall operations themselves could not be combined due to the serialized nature of the POSIX file-system API.

Ideally, a small file should be read using only one NFS compound (and one round trip), as shown in Figure 3.2. This would reduce the read latency by 80% by removing four of the five round trips. We can even read multiple files using a single compound, as shown in Figure 3.3. All these examples use the standard (unmodified) NFSv4 protocol. SAVEFH and RESTOREFH operate on the *saved filehandle* (SFH), an NFSv4 state similar to the *current filehandle* (CFH). SAVEFH copies the CFH to the SFH; RESTOREFH restores the CFH from the SFH.

For compounds to reach their full potential, we need a file-system API that can convey high-level semantics and batch multiple operations. We designed and developed *vNFS*, an NFSv4 client that exposes a high-level vectorized API. vNFS complies with the NFSv4.1 standard, requiring no changes to NFS servers. Its API is easy to use and flexible enough to serve as a building block for new higher-level functions. vNFS is implemented entirely in user space, and thus easy to extend.

vNFS is especially efficient and convenient for applications that manipulate large amounts of metadata or do small I/Os. For example, vNFS lets `tar` read many small files using a single RPC

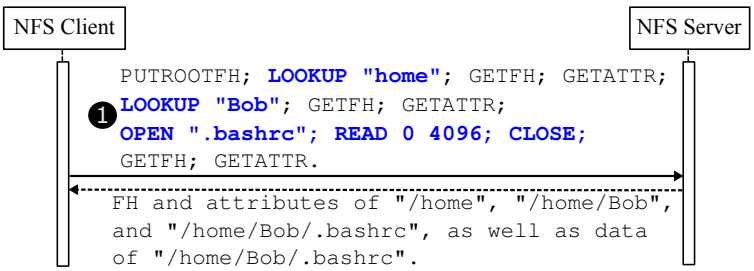


Figure 3.2: Reading `/home/Bob/.bashrc` using only one compound. This single compound is functionally the same as the five in Figure 3.1, but uses only one network round trip.

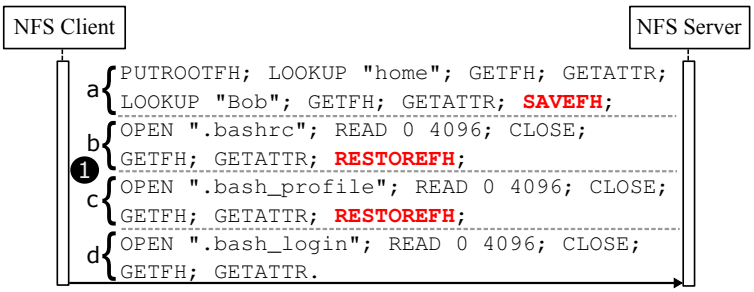


Figure 3.3: One NFS compound that reads three files. The operations can be divided into four groups: (a) sets the current and saved filehandle to `/home/Bob`; (b), (c), and (d) read the files `.bashrc`, `.bash_profile`, and `.bash_login`, respectively. SAVEFH and RESTOREFH (in red) ensure that the CFH is `/home/Bob` when opening files. The reply is omitted.

instead of using multiple RPCs for each; it also lets `untar` set the attributes of many extracted files at once instead of making separate system calls for each attribute type (owner, time, etc.).

We implemented vNFS using the standard NFSv4.1 protocol, and added two small protocol extensions to support file appending and copying. We ported GNU's Coreutils package (`ls`, `cp`, and `rm`), `tar`/`untar`, `nghttp2` (an HTTP/2 server), and Filebench [56, 178] to vNFS. In general, we found it easy to modify applications to use vNFS. We ran a range of micro- and macro-benchmarks on networks with varying latencies, showing that vNFS can speed such applications by 3–133× with small network latencies (≤5.2ms), and by up to 263× with a 30.2ms latency. This vNFS study has been published in the 15^{th} USENIX Conference on File and Storage Technologies (FAST 2017) [34].

The rest of this chapter is organized as follows. Chapter 3.2 summarizes vNFS's design. Chapter 3.3 details the vectorized high-level API. Chapter 3.4 describes the implementation of our prototype. Chapter 3.5 evaluates the performance and usability of vNFS by benchmarking applications we ported. Chapter 3.6 discusses related work and Chapter 3.7 concludes.

3.2 vNFS Design Overview

In this section we summarize vNFS's design, including our design goals, choices we made, and the vNFS architecture.

3.2.1 Design Goals

Our design has four goals, in order of importance:

- **High performance**: vNFS should considerably outperform existing NFS clients and improve both latency and throughput, especially for workloads that emphasize metadata and small I/Os. Performance for other workloads should be comparable.
- **Standards compliance**: vNFS should be fully compliant with the NFSv4.1 protocol so that it can be used with any compliant NFS server.
- **Easy adoption**: vNFS should provide a general API that is easy for programmers to use. It should be familiar to developers of POSIX-compliant code to enable smooth and incremental adoption.
- **Extensibility**: vNFS should make it easy to add functions to support new features and performance improvements. For example, it should be simple to add support for Server Side Copy (a feature in the current NFSv4.2 draft [77]) or create new application-specific high-level APIs.

3.2.2 Design Choices

The core idea of vNFS is to improve performance by using the compounding feature of standard NFS. We discuss the choices we faced and justify those we selected to meet the goals listed in Chapter 3.2.1.

3.2.2.1 Overt vs. covert coalescing

To leverage NFS compounds, vNFS uses a high-level API to overtly express the intention of compound operations. An alternative would be to covertly coalesce operations under the hood while still using the POSIX API. Covert coalescing is a common technique in storage and networking; for example, disk I/O schedulers combine many small requests into a few larger ones to minimize seeks [12]; and Nagle's TCP algorithm coalesces small outbound packets to amortize overhead for better network utilization [87].

Although overt compounding changes the API, we feel it is superior to covert coalescing in four important respects: **(1)** By using a high-level API, overt compounding can batch dependent operations, which are impossible to coalesce covertly. For example, using the POSIX API, we cannot issue a `read` until we receive the reply from the preceding `open`. **(2)** Overt compounding can use a new API to express high-level semantics that cannot be efficiently conveyed in low-level primitives. NFSv4.2's Server Side Copy is one such example [77]. **(3)** Overt compounding improves both throughput and latency, whereas covert coalescing improves throughput at the cost of latency, since accumulating calls to batch together inherently requires waiting. Covert coalescing is thus detrimental to metadata operations and small I/Os that are limited by latency. This is important in modern systems with faster SSDs and 40GbE NICs, where latency has been improving much slower than raw network and storage bandwidth [157]. **(4)** Overt compounding allows implementations to use all possible information to maximize performance; covert coalescing depends

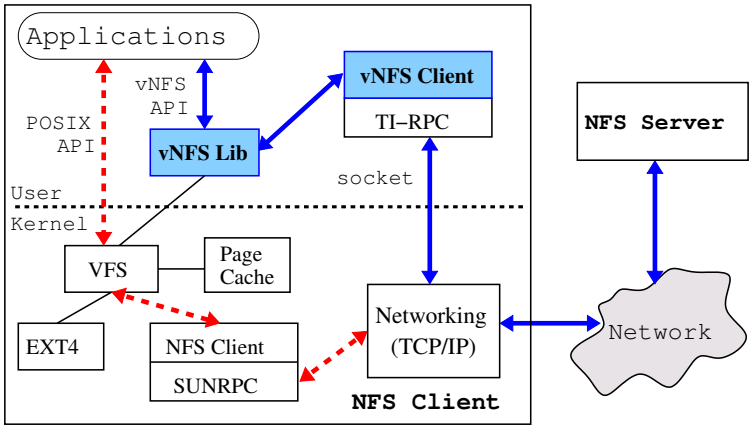


Figure 3.4: vNFS Architecture. The blue arrows show vNFS's data path, and the dashed red arrows show the in-kernel NFS client's data path. The vNFS library and client (blue shaded boxes) are new components we added; the rest existed before.

on heuristics, such as timing and I/O sizes, that can be sub-optimal or wrong. For example, Nagle's algorithm can interact badly with Delayed ACK [38].

3.2.2.2 Vectorized vs. start/end-based API

Two types of APIs can express overt compounding: a vectorized one that compounds many desired low-level NFS operations into a single high-level call, or an API that uses calls like `start_compound` and `end_compound` to combine all low-level calls in between [147]. We chose the vectorized API for two reasons: **(1)** A vectorized API is easier to implement than a start/end-based one. Users of a start/end-based API might mix I/Os with other code (such as looping and testing of file-system states), which NFS compounds cannot support. **(2)** A vectorized API logically resides at a high level and is more convenient to use, whereas using a low-level start/end-based API is more tedious for high-level tasks (e.g., C++ programming vs. assembly).

3.2.2.3 User-space vs. in-kernel implementation

A kernel-space implementation of vNFS would allow it to take advantage of the kernel's page and metadata caches and use the existing NFS code base. However, we chose to design and implement vNFS in user space for two reasons: **(1)** Adding a user-space API is much easier than adding system calls to the kernel, and simplifies future extensions. **(2)** User-space development and debugging is faster and easier. Although an in-kernel implementation might be faster, prior work indicates that the performance impact can be minimal [177], and the results in this chapter demonstrate substantial performance improvements even with our user-space approach.

3.2.3 Architecture

Figure 3.4 shows the architecture of vNFS, which consists of a library and a client. Instead of using the POSIX API, applications call the high-level vectorized API provided by the vNFS library, which talks directly to the vNFS client. To provide generic support and encourage incremental

Function	Description
vopen/vclose	Open/close many files.
vread/vwrite	Read/write/create/append files with automatic file opening and closing.
vgetattrs/vsetattrs	Get/set multiple attributes of objects.
vsscopy/vcopy	Copy files in whole or in part with/out Server Side Copy.
vmkdir	Create directories.
vlistdir	List (recursively) objects and their attributes in directories.
vsymlink	Create many symbolic links.
vreadlink	Read many symbolic links.
vhardlink	Create many hard links.
vremove	Remove many objects.
vrename	Rename many objects.

Table 3.1: vNFS vectorized API functions. Each function has two return values: an error code and a count of successful operations; errors halt processing at the server. To facilitate gradual adoption, vNFS also provides POSIX-like scalar API functions, omitted here for brevity. Each vNFS function has a version that does not follow symbolic links, also omitted.

adoption, the library detects when compound operations are unsupported, and instead converts vNFS operations into standard POSIX primitives.

A vNFS client accepts vectorized operations from the library, puts as many of them into each compound as possible, sends them to the NFS server using TI-RPC, and finally processes the reply. Note that existing NFSv4 servers already support compounds and can be used with vNFS without change. TI-RPC is a generic RPC library without the limitations (e.g., allowing only a single data buffer per call) of Linux's in-kernel SUNRPC; TI-RPC can also run on top of TCP, UDP, and RDMA. Like the in-kernel NFS client, the vNFS client also manages NFSv4's client-side states such as sessions, etc.

3.3 vNFS API

This section details vNFS's vectorized API (listed in Table 3.1). Each API function expands on its POSIX counterparts to operate on a vector of file-system objects (e.g., files, directories, symbolic links). Figure 3.5 demonstrates the use of the vectorized API to read three small files in one NFS compound. To simplify programming, vNFS also provides utility functions for common tasks such as recursively removing a whole directory, etc.

3.3.1 vread/vwrite

These functions can read or write multiple files using a single compound, with automatic on-demand file opening and closing. These calls boost throughput, reduce latency, and simplify programming. Both accept a vector of I/O structures, each containing a `vfile` structure (Figure 3.5), offset, length, buffer, and flags. Our vectorized operations are more flexible than the `readv` and `writev` system calls, and can operate at many (discontinuous) offsets of multiple files in one

```
struct vfile {
  enum VFILETYPE type;    // PATH or DESCRIPTOR
  union {
    const char *path;  // When "type" is PATH,
    int fd;       // or (vNFS file) DESCRIPTOR.
  };
};
// The "vio" I/O structure contains a vfile.
struct vio ios[3] = {
  { .vfile = { .type = PATH,
               .path = "/home/Bob/.bashrc" },
    .offset = 0,
    .length = 64 * 1024,
    .data = buf1, // pre-allocated 64KB buffer
    .flags = 0,  // contains an output EOF bit
  }, ...   // two other I/O structures omitted
};
struct vres r = vread(ios, 3); // read 3 files
```

Figure 3.5: A simplified C code sample of reading three files at once using the vectorized API.

call. When generating compound requests, vNFS adds OPENs and CLOSEs for files represented by paths; files represented by descriptors do not need that as they are already open. OPENs and CLOSEs are coalesced when possible, e.g., when reading twice from one file.

The length field in the I/O structure also serves as an output, returning the number of bytes read or written. The structure has several flags that map to NFS's internal Boolean arguments and replies. For example, the flag `is_creation` corresponds to the NFS `OPEN4_CREATE` flag, telling `vwrite` to create the target file if necessary. `is_write_stable` corresponds to NFS's WRITE `DATA_SYNC4` flag, causing the server to save the data to stable storage, avoiding a separate NFS COMMIT. Thus, a single `vwrite` can achieve the effect of multiple writes and a following `fsync`, which is a common I/O pattern (e.g., in logging or journaling).

■ **State management** NFSv4 is stateful, and OPEN is a state-mutating operation. The NFSv4 protocol requires a client to open a file before reading or writing it. Moreover, READ and WRITE must provide the *stateid* (an ID uniquely identifying a server's state [158]) returned by the preceding OPEN. Thus, state management is a key challenge when `vread` or `vwrite` adds OPEN and READ/WRITE calls into a single compound. vNFS solves this by using the NFS *current stateid*, which is a server-side state similar to the current filehandle. To ensure that the NFS server always uses the correct state, `vread` and `vwrite` take advantage of NFSv4's special support for using the current stateid [158, Section 8.2.3].

■ **Appending** `vwrite` also adds an optional small extension to the NFSv4.1 protocol to better support appends. As noted in the Linux manual page for `open(2)` [116], "O_APPEND may lead to corrupted files on NFS file systems if more than one process appends data to a file at once." The base NFSv4 protocol does not support appending, so the kernel NFS client appends by writing to an offset equal to the current known file size. This behavior is inefficient (the file size must

first be read) and is vulnerable to TOCTTOU races. Our extension uses a special offset value (UINT64_MAX) in the I/O structure to indicate appending, making appending reliable with a tiny (5 LoC) change to the NFS server.

3.3.2 vopen/vclose

Using vread and vwrite, applications can access files without explicit opens and closes. Our API still supports vopen and vclose operations, which add efficiency for large files that are read or written many times. vopen and vclose are also important for maintaining NFS's close-to-open cache consistency [103]. vopen opens multiple files (specified by paths) in one RPC, including LOOKUPs needed to locate their parent directories, as shown in Figure 3.3. Each file has its own open flags (read, write, create, etc.), which is useful when reading and writing are intermixed, such as external merge sorting. We also offer vopen_simple, which uses a common set of flags and mode (in case of creation) for all files. Once opened, a file is represented by a file descriptor, which is an integer index into an internal table that keeps states (file cursor, NFSv4 stateid and sequenceid [158], etc.) of open files. vclose closes multiple opened files and releases their resources.

3.3.3 vgetattrs/vsetattrs

These two functions manipulate several attributes of many files at once, combining multiple system calls (chmod, chown, utimes, and truncate, etc.) into a single compound, which is especially useful for tools like tar and rsync. The aging POSIX API is the only restriction on setting many attributes at once: the Linux kernel VFS already supports multi-attribute operations using the setattr method of inode_operations, and the NFSv4 protocol has similar SETATTRs support. vgetattrs and vsetattrs use an array of attribute structures as both inputs and outputs. Each structure contains a vfile structure, all attributes (mode, size, etc.), and a bitmap showing which attributes are in use.

3.3.4 vsscopy/vcopy

File copying is so common that Linux has added the sendfile and splice system calls to support it. Unfortunately, NFS does not yet support copying and clients must use READs and WRITEs instead, wasting time and bandwidth because data has to be read over the network and immediately written back. It is more efficient to ask the NFS server to copy the files directly on its side. This *Server Side Copy* (SSC) has already been proposed for the upcoming NFSv4.2 [77]. Being forward-looking, we included vsscopy in vNFS to copy many files (in whole or in part) using SSC; however, SSC requires server enhancements.

vsscopy accepts an array of copy structures, each containing the source file and offset, the destination file and offset, and the length. The destination files are created by vsscopy if necessary. The length can be UINT64_MAX, in which case the effective length is the distance between the source offset and the end of the source file. vsscopy can use a single RPC to copy many files in their entirety. The copy structures return the number of bytes successfully copied in the length fields.

`vcopy` has the same effect but does not use SSC. `vcopy` is useful when the NFS server does not support SSC; `vcopy` can copy N small files using three RPCs (a compound for each of `vgetattrs`, `vread`, and `vwrite`) instead of $7 \times N$ RPCs (2 OPENs, 2 CLOSEs, 1 GETATTR, 1 READ, and 1 WRITE for each file). A future API could provide only `vcopy` and silently switch to `vsscopy` when SSC is available; we include `vsscopy` separately in this thesis for comparison with `vcopy`.

3.3.5 `vmkdir`

vNFS provides `vmkdir` to create multiple directories at once (such as directory trees), which is common in tools such as `untar`, `cmake`, and recursive `cp`. vNFS contains a utility function `ensure_directory` that uses `vmkdir` to ensure a deep directory and all its ancestors exist. Consider `"/a/b/c/d"` for example: the utility function first uses `vgetattrs` with arguments `["/a"; "/a/b"; ...]` to find out which ancestors exist and then creates the missing directories using `vmkdir`. Note that simply calling `vmkdir` with vector arguments `["/a"; "/a/b"; ...]` does not work: the NFS server will fail (with `EEXIST`) when trying to recreate the first existing ancestor and stop processing all remaining operations.

3.3.6 `vlistdir`

This function speeds up directory listing with four improvements to `readdir`: (1) `vlistdir` lists multiple directories at once; (2) a prior `opendir` is not necessary for listing; (3) `vlistdir` retrieves attributes along with directory entries, saving subsequent `stats`; (4) `vlistdir` can work recursively. It can be viewed as a fast vectorized `ftw(3)` that reads NFS directory contents using as few RPCs as possible.

`vlistdir` takes five arguments: an array of directories to list, a bitmap indicating desired attributes, a flag to select recursive listing, a user-defined callback function (similar to `ftw`'s second argument [115]), and a user-provided opaque pointer that is passed to the callback. `vlistdir` processes directories in the order given; recursion is breadth-first. However, directories at the same level in the tree are listed in an arbitrary order.

3.3.7 `vsymlink/vreadlink/vhardlink`

Three vNFS operations, `vsymlink`, `vreadlink`, and `vhardlink`, allow many links to be created or read at once. Together with `vlistdir`, `vsymlink` can optimize operations like `"cp -sr"` and `"lndir"`. All three functions accept a vector of paths and a vector of buffers containing the target paths.

3.3.8 `vremove`

`vremove` removes multiple files and directories at once. Although `vremove` does not support recursive removal, a program can achieve this effect with a recursive `vlistdir` followed by properly ordered `vremoves`; vNFS provides a utility function `rm_recursive` for this purpose.

3.3.9 `vrename`

Renaming many files and directories is common, for example when organizing media collections. Many tools [8, 88, 95, 191] have been developed just for this purpose. vNFS provides `vrename` to facilitate and speed up bulk renaming. `vrename` renames a vector of source paths to a vector of destination paths.

3.4 vNFS Implementation

We have implemented a prototype of vNFS in C/C++ on Linux. As shown in Figure 3.4, vNFS has a library and a client, both running in user space. The vNFS library implements the vNFS API. Applications use the library by including the API header file and linking to it. For NFS files, the library redirects API function calls to the vNFS client, which builds large compound requests and sends them to the server via the TI-RPC library. For non-NFS files, the library translates the API functions into POSIX calls. (Our current prototype considers a file to be on NFS if it is under any exported directory specified in vNFS's configuration file.) The vNFS client builds on NFS-Ganesha [44, 136], an open-source user-space NFS server. NFS-Ganesha can export files stored in many backends, such as XFS and GPFS. Our vNFS client uses an NFS-Ganesha backend called PROXY, which exports files from *another* NFS server and can be easily repurposed as a user-space NFS client. The original PROXY backend uses NFSv4.0; we added NFSv4.1 support by implementing session management [158]. Our implementation of the vNFS client and library added 10,632 lines of C/C++ code and deleted 1,407. vNFS is thread-safe; we regression-tested it thoroughly. Our current prototype does not have a data or metadata cache. We have open-sourced all our code at `https://github.com/sbu-fsl/txn-compound`.

3.4.1 RPC size limit

The vNFS API functions (shown in Table 3.1) do not impose a limit on the number of operations per call. However, each RPC has a configurable memory size limit, defaulting to 1MB. We ensure that vNFS does not generate RPC requests larger than that limit no matter how many operations an API call contains. Therefore, we split long arguments into chunks and send one compound request for each chunk. We also merge RPC replies upon return, to hide any splitting.

Our splitting avoids generating small compounds. For data operations (`vread` and `vwrite`), we can easily estimate the sizes of requests and replies based on buffer lengths, so we split a compound only when its size becomes close to 1MB. (The in-kernel NFS client similarly splits large READs and WRITEs according to the `rsize` and `wsize` mount options, which also default to 1MB.) For metadata operations, it is more difficult to estimate the reply sizes, especially for READDIR and GETATTR. We chose to be conservative and simply split a compound of metadata operations whenever it contains more than k NFS operations. We chose a default of 256 for k, which enables efficient concurrent processing by the NFS server, and yet is unlikely to exceed the size limit. For example, when listing the Linux source tree, the average reply size of READDIR—the largest metadata operation—is around 3,800 bytes. If k is still too large (e.g., when listing large directories), the server will return partial results and use cookies to indicate where to resume the call for follow-up requests.

3.4.2 Protocol extensions

vNFS contains two extensions to the NFSv4.1 protocol to support file appending (see Chapter 3.3.1) and Server Side Copy (see Chapter 3.3.4). Both extensions require changes to the protocol and the NFS server. We have implemented these changes in our server, which is based on NFS-Ganesha [43, 44, 136]. The file-appending extension was easy to implement, adding only an `if` statement with 5 lines of C code. In the NFS server, we only need to use the file size as the effective offset whenever the write offset is `UINT64_MAX`.

Our implementation of Server Side Copy follows the design proposed in the NFSv4.2 draft [77]. We added the new COPY operation to our vNFS client and the NFS-Ganesha server. On the server side, we copy data using `splice(2)`, which avoids unnecessarily moving data across the kernel/user boundary. This extension added 944 lines of C code to the NFS-Ganesha server.

3.4.3 Path compression

We created an optimization that reduces the number of LOOKUPs when a compound's file paths have locality. The idea is to shorten paths that have redundancy by making them relative to preceding ones in the same compound. For example, when listing the directories `"/1/2/3/4/5/6/7/a"` and `"/1/2/3/4/5/6/7/b"`, a naïve implementation would generate eight LOOKUPs per directory (one per component). In such cases, we replace the path of the second directory with `"../b"` and use only one LOOKUPP and one LOOKUP; LOOKUPP sets the current filehandle to its parent directory. This simple technique saves as many as six NFS operations for this example.

Note that LOOKUPP produces an error if the current filehandle is not a directory, because most file systems have metadata recording parents of directories, but not parents of files. In that case, we use SAVEFH to remember the lowest common ancestor in the file-system tree (i.e., `"/1/2/3/4/5/6/7"` in the above example) of two adjacent files, and then generate a RESTOREFH and LOOKUPs. However, this approach cannot be used for LINK, RENAME, and COPY, which already use the saved filehandle for other purposes. Also, we use this optimization only when it saves NFS operations: for example, using `"../../c/d"` does not save anything for paths `"/1/a/b"` and `"/1/c/d"`.

3.4.4 Client-side caching

Our vNFS prototype does not yet have a client-side cache, which would be useful for re-reading recent data and metadata, streaming reads, and asynchronous writes. We plan to add it in the future. Compared to traditional NFS clients, vNFS does not complicate failure handling in the presence of a dirty client-side cache: cached dirty pages (not backed by persistent storage) are simply dropped upon a client crash; dirty data in a persistent cache (e.g., FS-Cache [80]), which may be used by a client holding write delegations, can be written to the server even faster during client crash recovery. Note that a client-side cache does not hold dirty metadata because all metadata changes are performed synchronously in NFS (except with directory delegations, which Linux has not yet implemented).

3.5 vNFS Evaluation

To evaluate vNFS, we ran micro-benchmarks and also ported applications to use it. We now discuss our porting experience and evaluate the resulting performance.

3.5.1 Experimental Testbed Setup

Our testbed consists of two 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, and an Intel 10GbE NIC. One machine acts as the NFS server and runs NFS-Ganesha with our file-appending and Server Side Copy extensions; the other machine acts as a client and runs vNFS. The NFS server exports an Ext4 file system stored on an Intel DC S3700 200GB SSD to the client. The two machines are directly connected to a Dell PowerConnect 8024F 10GbE switch, and we measured an average RTT of 0.2ms between them. To emulate different LAN and WAN conditions, we injected delays of 1–30ms into the outbound link of the server using `netem`: 30ms is the average latency we measured from our machines to the Amazon data center closest to us.

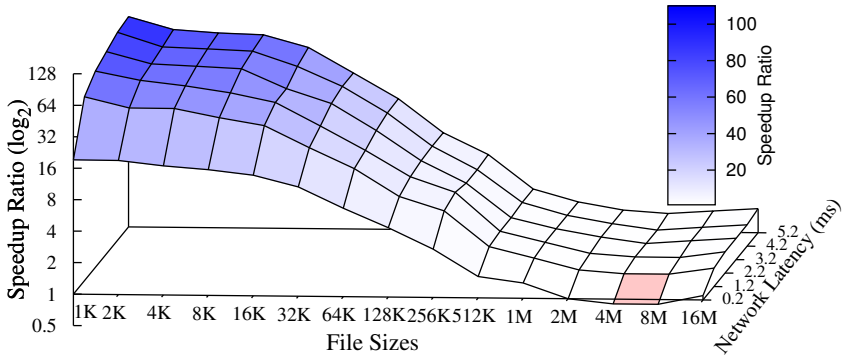
To evaluate vNFS's performance, we compared it with the in-kernel NFSv4.1 client (called *baseline*), which mounts the exported directory using the default options: the attribute cache (`ac` option) is enabled and the maximum read/write size (`rsize`/`wsize` options) is 1MB. The vNFS client does not use `mount`, but instead reads the exported directory from a configuration file during initialization. We ran each experiment at least three times and plotted the average value. We show the standard deviation as error bars, which are invisible in most figures because of their tiny values. We ran all experiments on network latencies $\leq$5.2ms. For longer latencies, we ran only a subset of tests because the baseline experiments were taking up to five hours for a single run. vNFS's benefits would only be magnified on higher-latency networks. Before each run, we flushed the page and dentry caches of the in-kernel client by unmounting and re-mounting the NFS directory. Currently, vNFS has no cache. The NFS-Ganesha server uses an internal cache, plus the OS's page and dentry caches.

3.5.2 Micro-workloads

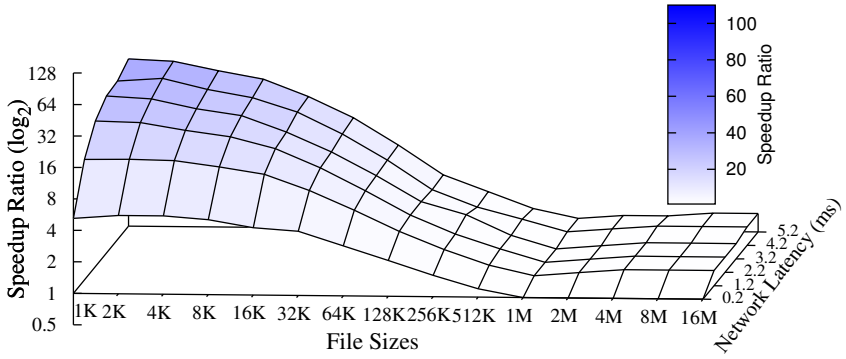
3.5.2.1 Small vs. big files

The most important goal of vNFS is to improve performance for workloads with many small NFS operations, while staying competitive for data-intensive workloads. To test whether vNFS has achieved this goal, we compared the time used by vNFS and the baseline to read and write 1,000 equally-sized files in their entirety while varying the file size from 1KB to 16MB. We repeated the experiment in networks with 0.2ms to 5.2ms latencies, and packed as many operations as possible into each vNFS compound. The results are shown (in logarithmic scale) in Figure 3.6, where *speedup ratio* (SR) is the ratio of the baseline's completion time to vNFS's completion time. SR values greater than one mean that vNFS performed better than the baseline; values less than one mean vNFS performed worse.

Because vNFS combined many small read and write operations into large compounds, it performed much better than the baseline when the file size was small. With a 1KB file size and 0.2ms network latency, vNFS is 19$\times$ faster than the baseline when reading (Figure 3.6(a)), and 5$\times$ faster



(a) Reading whole files



(b) Writing whole files

Figure 3.6: vNFS's speedup ratio (the vertical Z-axis, in logarithmic scale) relative to the baseline when reading and writing 1,000 equally-sized files, whose sizes (the X-axis) varied from 1KB to 16MB. vNFS is faster than (blue), equal to (white), or slower than (red) the baseline when the speedup ratio is larger than, equal to, or smaller than 1.0, respectively. The network latency (Y-axis) starts from 0.2ms (instead of zero) because that is the measured base network latency of our testbed (see Chapter 4.5.1).

when writing (Figure 3.6(b)). As the network latency increased to 5.2ms, vNFS's speedup ratio improved further to 103× for reading and 40× for writing. vNFS's speedup ratio was higher for reading than for writing because once vNFS was able to eliminate most network round trips, the NFS server's own storage became the next dominant bottleneck.

As the file size (the X-axis in Figure 3.6) was increased to 1MB and beyond, vNFS's compounding effect faded, and the performance of vNFS and the baseline became closer. However, in networks with 1.2–5.2ms latency, vNFS was still 1.1–1.7× faster than the baseline: although data operations were too large to be combined together, vNFS could still combine them with small metadata operations such as OPEN, CLOSE, and GETATTR. Combining metadata and data operations requires vNFS to split I/Os below 1MB due to the 1MB RPC size limit (see Chapter 3.4). When a large I/O is split into pieces, the last one may be a small I/O; this phenomenon made vNFS around 10% slower when reading 4MB and 8MB files in the 0.2ms-latency network. However, this is not a problem in most cases because that last small piece is likely to be combined into later compounds. This is why vNFS performed the same as the baseline with even larger file sizes (e.g., 16MB) in the 0.2ms-latency network. This negative effect of vNFS's splitting was unnoticeable for writing because writing was bottlenecked by the NFS server's storage. Note that the baseline (the in-kernel NFS client) splits I/Os strictly by 1MB size, although it also adds a few trivial NFS operations such as PUTFH (see Figure 3.1) in its compounds, meaning that the baseline's RPC size is actually larger than 1MB.

3.5.2.2 Compounding degree

The degree of compounding (i.e., the number of non-trivial NFS operations per compound) is a key factor determining how much vNFS can boost performance. The ideal scenario is when there is a large number of file system operations to perform at once, which is not always the case because applications may have critical paths that depend on only a single file. To study how the degree of compounding affects vNFS's performance, we compared vNFS with the baseline when calling the vNFS API functions with different numbers of operations in their vector arguments.

Figure 3.7 shows the speedup ratio of vNFS relative to the baseline as the number of operations per API call was increased from 1 to 256 in the 0.2ms-latency network. Even with a vector size of 1, vNFS outperformed the baseline for all API functions except two, because vNFS could still save round trips for single-file operations. For example, the baseline used three RPCs to rename a file: one RENAME, a LOOKUP for the source directory, and a LOOKUP for the destination directory, whereas vNFS used only one compound RPC that combined all three operations. `Getattrs` and `Setattr1` are the two exceptions where vNFS performed slightly worse (17% and 14% respectively) than the baseline. This is because both `Getattrs` and `Setattr1` need only a single NFS operation; therefore vNFS could not combine anything but suffered the extra overhead of performing RPCs in user space.

When there was more than one operation per API call, compounding became effective and vNFS significantly outperformed the baseline for all API calls; note that the Y axis of Figure 3.7 is in logarithmic scale. All calls except `Write4KSync` (bottlenecked by the server's storage stack) were more than 4× faster than the baseline when multiple operations were compounded. Note that `vsetattrs` can set multiple attributes at once, whereas the baseline sets one attribute at a time. We observe in Figure 3.7 that the speedup ratio of setting more attributes (e.g., `Setattr4`) at once was always higher than that of setting fewer (e.g., `Setattr3`).

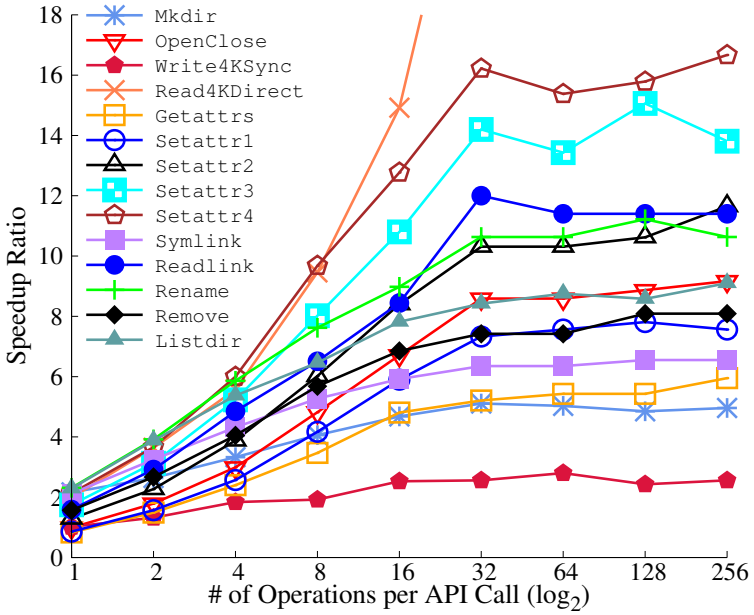


Figure 3.7: vNFS's speedup ratio relative to the baseline under different degrees of compounding. The X-axis is $\log_2$. The network latency is 0.2ms. Write4KSync writes 4KB data to files opened with `O_SYNC`; Read4KDirect reads 4KB data from files opened with `O_DIRECT`; `Setattr`N sets N files' attributes (mixes of mode, owner, timestamp, and size). The vector size of the baseline is actually the number of individual POSIX calls issued iteratively. The speedup ratio of Read4KDirect goes up to 46 at 256 operations per call; its curve is cut off here.

In our experiments with slower networks (omitted for brevity), vNFS's speedups relative to the baseline were even higher than in the 0.2ms-latency network: up to two orders of magnitude faster.

3.5.2.3 Caching

Our vNFS prototype does not yet support caching. In contrast, the baseline (in-kernel NFS client) caches both metadata and data. To study the cache's performance impact, we compared vNFS and the baseline when repeatedly opening, reading, and closing a single file whose size varied from 1KB to 16MB. Figure 3.8 shows the results, where a speedup ratio larger than one means vNFS outperformed the baseline; and a speedup ratio less than one means vNFS performed worse.

The baseline served all reads except the first from its cache, but it was slower than vNFS (which did not cache) when the file size was 256KB or smaller. This is because three RPCs per read are still required to maintain close-to-open semantics: an OPEN, a GETATTR (for cache revalidation), and a CLOSE. In comparison, vNFS used only one compound RPC, combining the OPEN, READ (uncached), and CLOSE. The savings from compounding more than compensated for vNFS's lack of a cache. For a 512KB file size, vNFS was still faster than the baseline except in the 0.2ms-latency network; for 1MB and larger file sizes, vNFS was worse than the baseline because read operations became dominant and the baseline served all reads from its client-side cache whereas vNFS had to send all the reads to the server without the benefit of caching.

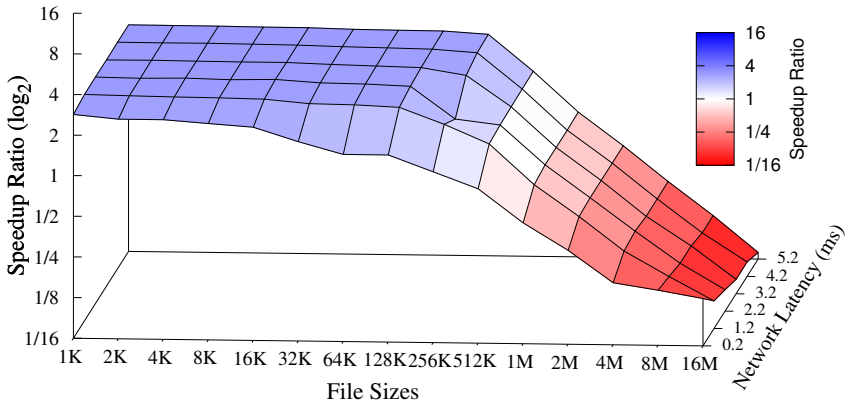


Figure 3.8: The speedup ratio of vNFS over the baseline (in logarithmic scale) when repeatedly opening, reading, and closing a single file, whose size is shown in the X-axis. vNFS is faster than (blue), equal to (white), and slower than (red) the baseline when the speedup ratio is larger than, equal to, and smaller than 1, respectively. Our vNFS prototype does not have a cache yet, whereas the baseline does. The Z-axis is in logarithmic scale; the higher the better.

3.5.3 Macro-workloads

To evaluate vNFS using realistic applications, we modified `cp`, `ls`, and `rm` from GNU Coreutils, Filebench [56, 178], and nghttp2 [138] to use the vNFS API; we also implemented an equivalent of GNU `tar` using vNFS.

3.5.3.1 GNU Coreutils

To benchmark realistic applications, we ported several popular GNU Coreutils programs to vNFS: `cp`, `ls`, and `rm`. We used the modified versions to copy, list, and remove the entire Linux-4.6.3 source tree: it contains 53,640 files with an average size of 11.6KB, 3,604 directories with average 17 children per directory, and 23 symbolic links. The large number of files and directories can thoroughly exercise vNFS and demonstrate the performance impact of compounding.

Porting `cp` and `rm` to vNFS was easy. For `cp`, we added 170 lines of code and deleted 16; for `rm`, we added 21 and deleted 1. Copying files can be trivially achieved using `vsscopy`, `vgetattrs`, and `vsetattrs`. Recursively copying directories requires calling `vlistdir` on the directories and then invoking `vsscopy` for plain files, `vmkdir` for directories, and `vsymlink` for symbolic links—all of which is done in `vlistdir`'s callback function. We tested our modified `cp` with `diff -qr` to ensure that the copied files and directories were exactly the same as the source. Removing files and directories recursively in `rm` was similar, except that we used `vremove` instead of `vsscopy`.

Porting `ls` was more complex because batching is difficult when listing directories recursively in a particular order. Unlike `cp` and `rm`, `ls` has to follow a certain order (e.g., by size or by last-modified timestamp) when options such as `--sort` are specified. We could not use the recursive mode of `vlistdir` because the underlying READDIR NFS operation does not follow any specific order when reading directory entries, and the whole directory tree may be too large to fit in memory. Instead, vNFS maintains a list of all directories to read in the proper order as specified by the `ls`

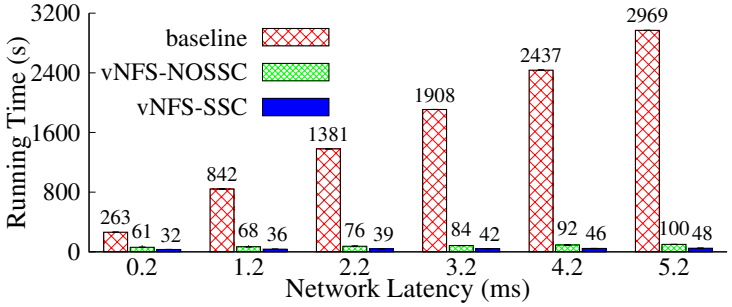


Figure 3.9: Running time to copy (`cp -r`) the entire Linux source tree. The lower the better. vNFS runs much faster than the baseline both with and without Server Side Copy (SSC); thus the vNFS bars are tiny.

options, and repeatedly calls `vlistdir` (not recursively) on directories at the head of the list until it is empty. Note that (1) a directory is removed from the list only after all its children have been read; and (2) sub-directories should be sorted and then inserted immediately after their parent to maintain the proper order in the list. This algorithm enables our ported `ls` to efficiently compound READDIR operations while still keeping a small memory footprint. We added 392 lines of code and deleted 203 to port `ls` to vNFS. We verified that our port is correct by comparing the outputs of our `ls` with the vanilla version.

Figure 3.9 shows the results of copying the entire Linux source tree; vNFS outperformed the baseline in all cases. vNFS uses either `vsscopy` or `vcopy` depending on whether Server Side Copy (SSC) is enabled. However, the baseline cannot use SSC because it is not yet supported by the in-kernel NFS client. For the same workload of copying the Linux source tree, vNFS used merely 4,447 compounding RPCs whereas the baseline used as many as 506,697: two OPENs, two CLOSEs, one READ, one WRITE, and one SETATTR for each of the 53,640 files; 60,873 ACCESSs; 62,327 GETATTRs; and 8,017 other operations such as READDIR and CREATE. vNFS-NOSSC saved more than 99% of RPCs compared to the baseline, with each vNFS compounding RPC containing an average of 250 operations. This simultaneously reduced latency and improved network utilization. Therefore, even for a fast network with only a 0.2ms latency, vNFS-NOSSC is still more than 4× faster than the baseline. The speedup ratio increases to 30× with a 5.2ms network latency. When Server Side Copy (SSC) was enabled, vNFS ran even faster, and vNFS-SSC reduced the running time of vNFS-NOSSC by half. The further speedup of SSC is only moderate because the files are small and our network bandwidth (10GbE) is not a bottleneck. The speedup ratio of vNFS-SSC to the baseline is 8–60× in networks with 0.2ms to 5.2ms latency. Even when the Linux kernel adds SSC support to its NFSv4 implementation, vNFS would still outperform it because this workload's bottleneck is the large number of small metadata operations, not data operations.

With the `-Rs` options, `cp` copies an entire directory tree by creating symbolic links to the source directory. Figure 3.10 shows speedups for symlinking, for recursively listing (`ls -Rl`), and for recursively removing (`rm -Rf`) the Linux source tree. In each of these three metadata-heavy workloads, vNFS outperformed the baseline regardless the network latency: all speedup ratios are larger than one.

When recursively listing the Linux tree, `ls`-baseline used 10,849 RPCs including 3,678 READDIRs, 3,570 ACCESSes, and 3,570 GETATTRs. Note that the in-kernel NFS client did not issue a

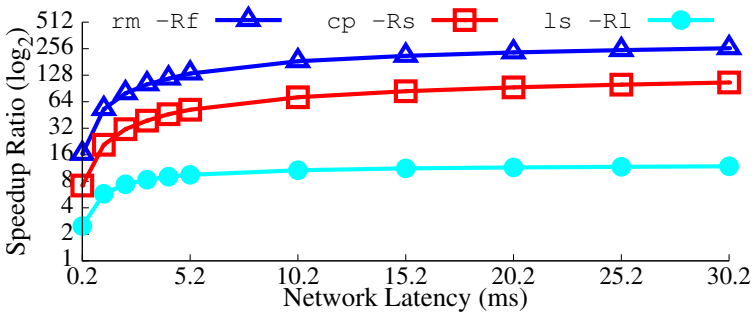


Figure 3.10: vNFS's speedup relative to the baseline when symbolically copying (`cp -Rs`), listing (`ls -Rl`), and removing (`rm -Rf`) the entire Linux source tree. The Y-axis uses a logarithmic scale; the higher the better.

separate GETATTR for each directory entry although the vanilla `ls` program called `stat` for each entry listed. This is because the in-kernel NFS client pre-fetches the attributes using `readdirs` and serves the `stat` calls from the local client's dentry metadata cache. This optimization enables `ls`-baseline to finish the benchmark in just 5 seconds in the 0.2ms-latency network. However, with our vectorized API, `ls`-vNFS did even better and finished the benchmark in 2 seconds, using only 856 RPCs. Moreover, vNFS scales much better than the baseline. When the latency increased from 0.2 to 30.2ms, vNFS's running time rose to only 28 seconds whereas the baseline increased to 336 seconds. `ls`-vNFS is 10× faster than `ls`-baseline in high-latency (>5.2ms) networks.

Compared to the directory-listing benchmark, the speedup of vNFS is even higher in the symbolic copying and removing benchmarks shown in Figure 3.10. In the 0.2ms-latency network, vNFS was 7× and 18× faster than the baseline for symbolic copying and removing, respectively. This is because the baseline always operated on each file at a time, whereas vNFS could copy or remove more than 200 files at once. Compared to the baseline, vNFS reduced the number of RPCs by 99.1% in the copying benchmark, and as much as 99.7% in the removing benchmark. This massive saving of RPCs improved `cp` by 52× and `rm` by 133× in the 5.2ms-latency network; for 30.2ms the speedup ratios became 106× for `cp`, and 263× for `rm`. For both removing and symbolic copying, vNFS ran faster in the 30.2ms-latency network (25 and 15 seconds, respectively) than the baseline did with 0.2ms latency (38s and 55s, respectively), demonstrating that compounds can indeed help NFSv4 realize its design goal of being WAN-friendly [121].

3.5.3.2 `tar`

Because the I/O code in GNU `tar` is closely coupled to other code, we implemented a vNFS equivalent using `libarchive`, in which the I/O code is clearly separated. The `libarchive` library supports many archiving and compression algorithms; it is also used by FreeBSD `bsdtar`. Our implementation needed only 248 lines of C code for `tar` and 210 for `untar`.

When archiving a directory, we use the `vlistdir` API to traverse the tree and add subdirectories into the archive. We gather the listed files and symlinks into arrays, then read their contents using `vread` and `vreadlink`, and finally compress and write the contents into the archive. During extraction, we read the archive in 1MB (RPC size limit) chunks and then use `libarchive` to extract and decompress objects and their contents, which are then passed in

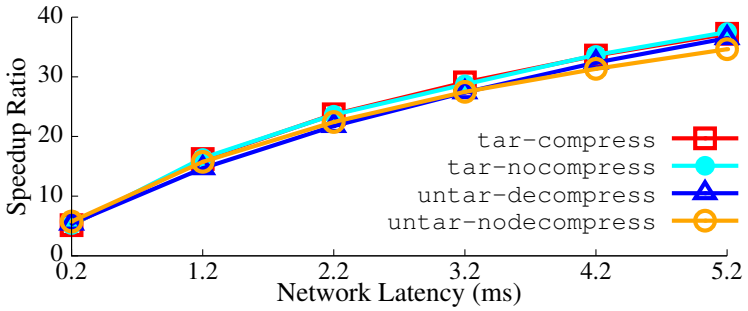


Figure 3.11: Speedup ratios of vNFS relative to the baseline when archiving (`tar`) and extracting (`untar`) the Linux-4.6.3 source tree, with and without xz compression.

batches to `vmkdir`, `vwrite`, or `vsymlink`. We always create parent directories before their children. We ensured that our implementation is correct by feeding our `tar`'s output into our `untar` and compared the extracted files with the original input files we fed to `tar`. We also tested for cross-compatibility with other tar implementations including `bsdtar` and GNU `tar`.

We used our `tar` to archive and `untar` to extract a Linux 4.6.3 source tree. Archiving read 53,640 small files and wrote a large archive: 636MB uncompressed, and 86MB with the `xz` option (default compression used by `kernel.org`). Extracting reversed the process. There are also metadata operations on 23 symbolic links and 3,604 directories. Therefore, this test exercised a wide range of vNFS functions.

Figure 3.11 shows the `tar`/`untar` results, compared to `bsdtar` (running on the in-kernel client) as the baseline. For `tar-nocompress` in the 0.2ms-latency network, vNFS was more than 5× faster than the baseline because the baseline used 446,965 RPCs whereas vNFS used only 2,144 due to compounding. This large reduction made vNFS 37× faster when the network latency increased to 5.2ms. In terms of running time, vNFS used 69 seconds to archive the entire Linux source tree in the 5.2ms-latency network, whereas the baseline, even in the faster 0.2ms-latency network, still used as much as 192 seconds. For `untar-nodecompress`, vNFS is also 5–36× faster, depending on the network latency.

Figure 3.11 also includes the results when `xz` compression was enabled. Although compression reduced the size of the archive file by 86% (from 636MB to 86MB) and thus saved 86% of the I/Os to the archive file, this had a negligible performance impact (less than 0.5%) because the most time-consuming operations are for small I/Os, not large ones. This test shows that workloads with mixed I/O sizes are slow if there are many small I/Os, each incurring a network round trip; vNFS can significantly improve such workloads by compounding those small I/Os.

3.5.3.3 Filebench

We have ported Filebench to vNFS and added vectorized *flowops* to the Filebench workload modeling language (WML) [198]. We added 759 lines of C code to Filebench, and removed 37. We converted Filebench's File-Server, NFS-Server, and Varmail workloads to equivalent versions using the new flowops: for example, we replaced N adjacent sets of `openfile`, `readwholefile`, and `closefile` (i.e., $3 \times N$ old flowops) with a single `vreadfile` (one new flowop), which internally uses our `vread` API that can open, read, and close N files in one call.

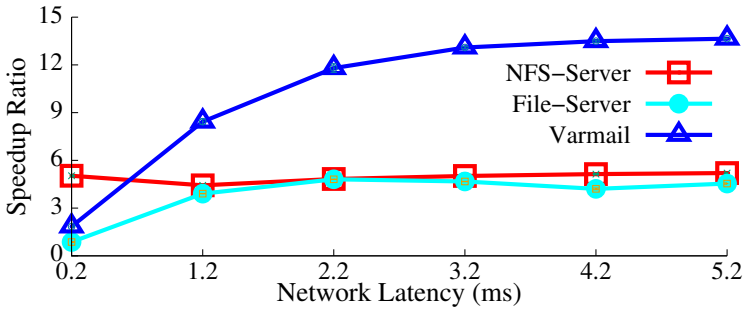


Figure 3.12: vNFS's speedup ratios for Filebench workloads.

The Filebench NFS-Server workload emulates the SPEC SFS benchmark [170]. It contains one thread performing four sets of operations: (1) open, entirely read, and close three files; (2) read a file, create a file, and delete a file; (3) append to an existing file; and (4) read a file's attributes. The File-Server workload emulates 50 users accessing their home directories and spawns one thread per user to perform operations similar to the NFS-Server workload. The Varmail workload mimics a Unix-style email server operating on a `/var/mail` directory, saving each message into a file. This workload has 16 threads, each performing create-append-sync, read-append-sync, read, and delete operations on a set of 10,000 16KB files. Since none of these workloads perform many same-type operations at a time, their vNFS versions have only moderate degrees of compounding.

Figure 3.12 shows the results of the Filebench workloads, comparing vNFS to the baseline. For the NFS-Server workload, vNFS was 5× faster than the baseline in the 0.2ms-latency network because vNFS combined multiple small reads and their enclosing opens and closes into a single compound. vNFS was also more efficient (and more reliable) when appending files since it does not need a separate GETATTR to read the file size (see Chapter 3.3.1). This single-threaded NFS-Server workload is light, and its only bottleneck is the delay of network round trips. With compounding, vNFS can save network round trips; the amount of savings depends on the compounding degree (the number of non-trivial NFS operations per compound). This workload has a compounding degree of around 5, and thus we observed a consistent 5× speedup regardless of the network latency.

As shown in Figure 3.12, vNFS's speedup ratio in the File-Server workload is about the same as the NFS-Server one, except in the 0.2ms-latency network. This is because these two workloads have similar file-system operations and thus similar compounding degrees. However, in the 0.2ms-latency network, vNFS was 13% slower (i.e., a speedup ratio of 0.87) than the baseline. This is caused by two reasons: (1) the File-Server workload has as many as 50 threads and generates a heavy I/O load to the NFS server's storage stack, which became the bottleneck; (2) without a cache, vNFS sent all read requests to the overloaded server whereas the in-kernel client's cache absorbed more than 99% of reads. As the network latency increased, the load on the NFS server became lighter and vNFS became faster thanks to saving round trips, which more than compensated for the lack of cache in our current prototype.

Because the Varmail workload is also multi-threaded, its speedup ratio curve in Figure 3.12 has a trend similar to that of the File-Server workload. However, vNFS's speedup ratio in the Varmail workload plateaued at the higher value of 14× because its compounding degree is higher than the File-Server workload.

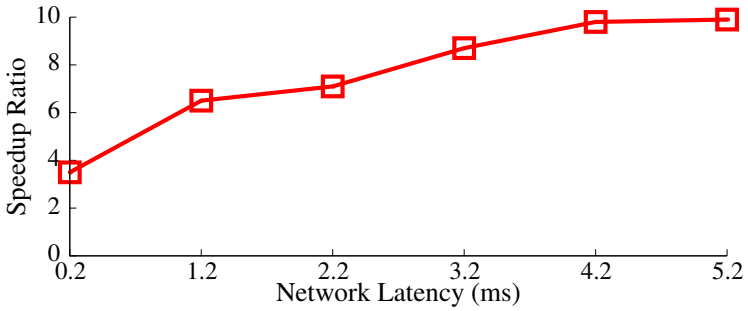


Figure 3.13: vNFS speedup ratio relative to the baseline when requesting a set of objects with PUSH enabled in nghttp2.

3.5.3.4 HTTP/2 server

Similar to the concept of NFSv4 compounds, HTTP/2 improves on HTTP/1.*x* by transferring multiple objects in one TCP connection. HTTP/2 also added a PUSH feature that allows an HTTP/2 server to proactively push related Web objects to clients [21, Section 8.2]. For example, upon receiving an HTTP/2 request for `index.html`, the server can proactively send the client other Web objects (such as Javascript, CSS, and image files) embedded inside that `index.html` file, instead of waiting for it to request them later. PUSH can reduce a Web site's loading time for end users. It also allows Web servers to read many related files together, enabling efficient processing by vNFS.

We ported *nghttp2* [138], an HTTP/2 library and tool-set containing an HTTP/2 server and client, to vNFS. Our port added 543 lines of C++ code and deleted 108.

The HTTP Archive [81] shows that, on average, an HTTP URL is 2,480KB and contains ten 5.5KB HTML files, 23 20KB Javascript files, seven 7.5KB CSS files, and 56 28KB image files. We created a set of files with those characteristics, hosted them with our modified nghttp2 server, and measured the time needed to process a PUSH-enabled request to read the file set. Figure 3.13 shows the speedup ratio of vNFS relative to the baseline, which runs vanilla nghttp2 and the in-kernel NFS client. vNFS needed only four NFS compounds for all 96 files: one `vgetattrs` call and three `vreads`. In contrast, the baseline used 309 RPCs including one OPEN, READ, and CLOSE for each file. The reduced network round trips made vNFS 3.5× faster in the 0.2ms-latency network and 9.9× faster with the 5.2ms latency.

Although current Web servers are often deployed close to their underlying storage, we believe that new technologies such as HTTP/2 and vNFS will allow wider separation of servers and storage, enabling widely distributed systems with higher scalability and reliability.

3.6 Related Work of vNFS

3.6.1 Improving NFS performance

The ubiquitous Network File System, which is more than 30 years old, has continuously evolved to improve performance. Following the initial NFSv2 [175], NFSv3 added asynchronous COMMITs to improve write performance, and READDIRPLUS to speed up directory listing [31]. NFSv4.0 [164]

added more performance features including compounding procedures that batch multiple operations in one RPC, and delegations that enable the client cache to be used without lengthy revalidation. To improve performance further with more parallelism, NFSv4.1 [158] added pNFS [78] to separate data and metadata servers so that the different request types can be served in parallel. The upcoming NFSv4.2 has yet more performance improvements such as I/O hints, *Application Data Blocks*, and Server Side Copy [77].

In addition to improvements in the protocols, other researchers also improved NFS's performance: Duchamp found it inefficient to look up NFSv2 paths one component at a time, and reduced client latency and server load by optimistically looking up whole paths in a single RPC [49]. Juszczak improved the write performance of an NFS server by gathering many small writes into fewer larger ones [92]. Ellard and Seltzer improved read performance with read-ahead and stride-read algorithms [52]. Batsakis et al. [17] developed a holistic framework that adaptively schedules asynchronous operations to improve NFS's performance as perceived by applications. Our vNFS uses a different approach, improving performance by making NFSv4's compounding procedures easily accessible to programmers.

3.6.2 I/O compounding

Compounding, also called batching and coalescing, is a popular technique to improve throughput and amortize cost by combining many small I/Os into fewer larger ones. Disk I/O schedulers coalesce adjacent I/Os to reduce disk seeks [12] and boost throughput. Purohit et al. [147] proposed Compound System Calls (Cosy) to amortize the cost of context switches and to reduce data movement across the user-kernel boundary. These compounding techniques, as well as NFSv4's compounding procedures, are all hidden behind the POSIX file-system API, which cannot convey the required high-level semantics [32]. The Batch-Aware Distributed File System (BAD-FS) [22] demonstrated the benefits of using high-level semantics to explicitly control the batching of I/O-intensive scientific workloads. *Dynamic sets* [172] took advantage of the fact that files can be processed in any order in many bulk file-system operations (e.g., `grep foo *.c`). Using a set-based API, distributed file system clients can pre-fetch a set of files in the optimal order and pace so that computation and I/O are overlapped and the overall latency is minimized. However, dynamic sets did not reduce the number of network round trips. To the best of our knowledge, vNFS is the first attempt to use an overt-compounding API to leverage NFSv4's compounding procedures.

3.6.3 Vectorized APIs

To achieve high throughput, Vilayanur et al. [186] proposed `readx` and `writex` to operate at a vector of offsets so that the I/Os can be processed in parallel. However, these operations are limited to a single file, helping only large files, whereas our `vread`/`vwrite` can access many files at once, helping with both large and small files.

Vasudevan et al. [184] envisioned the Vector OS (VOS), which offered several vectorized system calls, such as `vec_open()`, `vec_read()`, etc. While VOS is promising, it has not been fully implemented yet. In their prototype, they succeeded in delivering millions of IOPS in a distributed key-value (KV) store backed by fast NVM [185]. However, they implemented a key-value API, not a file-system API, and their vectorized KV store focuses on serving parallel I/Os on NVM,

whereas vNFS focuses on saving network round trips by using NFSv4 compound procedures. The vectorized key-value store and vNFS are different but complementary.

Our vNFS API is also different from other vectorized APIs [184, 186] in three aspects: (1) `vread`/`vwrite` supports automatic file opening and closing; (2) `vsscopy` takes advantage of the NFS-specific Server Side Copy feature; and (3) to remain NFSv4-compliant, vNFS's vectorized operations are executed in order, in contrast to the out-of-order execution of `lio_listio(3)` [108], `vec_read()` [184], and `readx` [186].

3.7 vNFS Conclusions

We designed and implemented vNFS, an NFSv4.1 client and API library that maximize NFS performance in LAN and WAN environments. vNFS uses a vectorized high-level API to leverage standard NFSv4 compounds, which have the potential to reduce network round trips but were severely underutilized due to the low-level and serialized nature of the POSIX API. vNFS makes maximal use of compounds by enabling applications to operate on many file-system objects in a single RPC. To further improve performance, vNFS supports reliable file appends and Server Side Copy that reduce the latency and bandwidth demands of file appending and copying.

We ported several applications to use vNFS, including `cp`, `ls` and `rm` from GNU Coreutils; `bsdtar`; Filebench; and nghttp2. The porting was generally easy.

Micro-benchmarks demonstrated that—compared to the in-kernel NFS client—vNFS significantly boosts the throughput of workloads with many small I/Os and metadata operations even in fast networks, and performs comparably for large I/Os or with low compounding degrees. Our benchmarks using the ported applications show that vNFS can make these applications faster by up to two orders of magnitude.

Limitations and future work Currently vNFS does not include a cache; an implementation is underway. To simplify error handling, we plan to support optionally executing a compound as an atomic transaction. Finally, compounded operations are processed sequentially by current NFS servers; we plan to execute them in parallel with careful interoperation with transactional semantics.

Chapter 4

SeMiNAS: A Secure Middleware for Cloud-Backed Network-Attached Storage

4.1 SeMiNAS 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 and Chapter 3, we decided to use NFSv4.1 as Kurma's storage protocol because of NFSv4.1's advanced features such as delegations and compound procedures. Now 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 gateways for security enhancement and performance improvement. The high-level architectures of SeMiNAS and Kurma are similar: they differ mostly in their gateway architecture and components. SeMiNAS provides the same caching and security features including confidentiality and integrity. 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 SeMiNAS requires new NFS features not standardized yet or available from current cloud providers. These limitations are solved in our final Kurma design in Chapter 5.

SeMiNAS consists of on-premises gateways that allow clients to outsource data securely to clouds using the same file system API as traditional NAS appliances. As shown in Figure 4.1, SeMiNAS inherits many advantages from the popular middleware architecture, as exemplified by network firewalls. For instance, SeMiNAS can protect a large number of clients by consolidating a small number of SeMiNAS gateways; SeMiNAS also minimizes migration costs by requiring only minimal configuration changes to existing clients and servers.

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

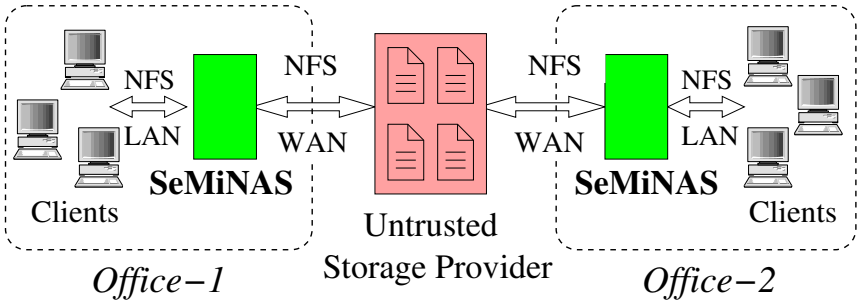


Figure 4.1: SeMiNAS high-level architecture. Each geographic office has a trusted SeMiNAS gateway to protect clients from the untrusted storage provider. SeMiNAS can securely share files among geo-distributed offices.

SeMiNAS retrieves and decrypts 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 simple and robust key exchange scheme, SeMiNAS can share files securely among geo-distributed offices without relying on any trusted third-party or secret channel for key management.

SeMiNAS ensures its security scheme is efficient with three mechanisms: (1) SeMiNAS stores Message Authentication Codes (MAC) together with file data using NFS data-integrity extension [137], so that no extra WAN round trips are needed for checking and updating MACs. (2) SeMiNAS uses NFSv4's compound procedures to combine operations on file headers (added by SeMiNAS for distributing file keys securely among gateways) with small metadata operations. Therefore, SeMiNAS does not introduce any extra WAN round trips for its security metadata. (3) SeMiNAS contains a persistent write-back cache that stores recently used data and coalesces writes to the server; this further reduces the communication to remote servers. With these three optimizations, SeMiNAS is able to provide integrity and confidentiality with a small overhead of less than 18%.

This chapter makes three contributions: **(1)** a middleware system that allows clients to securely and efficiently store and share files in remote NAS providers; **(2)** a study of leveraging NFSv4 compound procedures for a highly efficient security scheme; and **(3)** an implementation and evaluation of NFSv4 end-to-end Data Integrity eXtension (DIX) [137]. This SeMiNAS study has been published in the 9^{th} ACM International Systems and Storage Conference (SYSTOR 2016) [35].

The rest of this chapter is organized as follows. Chapter 4.2 presents the background and motivation behind SeMiNAS. Chapter 4.3 and Chapter 4.4 describe its design and implementation, respectively. Chapter 4.5 evaluates SeMiNAS under different networks and workloads. Chapter 4.6 discusses related work and Chapter 4.7 concludes this chapter.

4.2 SeMiNAS Background and Motivation

We present the background and motivation behind SeMiNAS by answering two questions: (1) Why we need yet another cryptographic file system on an untrusted server—an area studied more than a decade ago? (2) Why SeMiNAS uses an NFS back-end instead of a key-value object back-end?

4.2.1 A revisit of cryptographic file systems.

Utility computing requires storing data in external providers. Consequently, security concerns arise because of the opaque and multi-tenant nature of external servers, as well as the large exploitation surface area of public wide-area networks. Using cryptographic file systems to securely outsource data has been studied before in SFS [120], Cepheus [94], SiRiUS [70], SUNDR [104], Iris [173], and others [65, 66, 93, 189, 201]. However, a revisit of these studies is needed for three reasons.

First, modern cryptographic file systems should protect data not only from potentially malicious servers, but also from the deep storage stack (with multiple layers of virtualization, software, firmware, hardware, and networking), which is much more complex and error-prone than before [15, 111]. Data Integrity eXtensions (DIX) [45] is a growing trend of making the once hidden Data Integrity Fields (DIF) of storage devices available to applications. This can help keep data safe from both malicious servers and a misbehaving storage stack. By providing an eight-byte out-of-band storage for security checksums per 512-byte sector, DIX can protect the whole data path from applications all the way down to physical storage media.

Second, newer and more powerful networking storage protocols have emerged, particularly NFSv4 [77, 158, 164]. Compared to its previous versions, NFSv4 is superior not only in performance, scalability, and manageability [32, 121], but also in security with RPCSEC_GSS [51] and ACLs [77, 158, 164]. Moreover, with advanced features including compound procedures and delegations, NFSv4 provides great opportunities for making cryptographic file system flexible and efficient when storage servers are remote over WANs.

Third, the performance penalty of modern cryptographic file systems should be small enough to maintain a lower total cost of ownership—a key incentive for outsourcing storage. Some researchers [37] argued that encrypting data in cloud was too expensive, whereas others [1, 189] claimed new hardware acceleration makes encryption viable and cheap for cloud storage. These debates highlight the importance of reducing performance overhead when securing cloud storage. Therefore, SeMiNAS strives for high performance as well as security, whereas many prior systems [65, 70, 104] sacrificed performance by up to 90% for security.

4.2.2 An NFS vs. a key-value object back-end.

Currently, most cloud storage vendors provide key-value object stores. However, SeMiNAS uses an NFS back-end instead for four reasons. First, we believe that cloud storage is still in its early age and future cloud storage will offer richer, file-system APIs in addition to key-value object APIs. Key-value object stores are popular now primarily because of simplicity. File-systems APIs in clouds are likely to grow in popularity as cloud applications demand more functions from cloud storage vendors. This is a trend as seen by the recent cloud offering of the NFSv4-based Amazon Elastic File System [85].

Second, the open, pervasive, and standard NFS API has many advantages over vendor-specific object APIs. NFS is compatible with the POSIX standard, and most applications based on direct-attached storage can continue to work on NFS without change. Therefore, migration from on-premises storage to cloud NAS providers requires only minimal effort. By contrast, a full migration from file systems to key-value object stores can be prohibitive because object stores support fewer features (i.e., absence of file attributes, links, locks) and sometimes use weaker consistency models (e.g., eventual consistency [42]) than file systems.

Third, file systems have much richer semantics than key-value object stores, and can significantly simplify application development. As more applications are deployed in clouds, rudimentary object stores have begun to fall short of functionalities to support complex systems [142]. The richer semantics of file systems also provide more optimization opportunities than key-value stores. For example, NFS can be much more efficient with pNFS [158], server-side copying [77], and Application-Data Blocks [77].

Fourth, NFSv4 is optimized for WANs, and its speed over WANs can be considerably improved by caching, as was demonstrated by both academia [112] and industry [154]. The performance boost of an NFS cache, such as SeMiNAS, can be particularly significant with NFSv4 delegations—a client caching mechanism that enables local file operations without communication to remote servers. Our benchmarking study in Chapter 2 showed that delegations can reduce the number of NFS messages by almost 31×. Delegations do not compromise NFS's strong consistency [158]; they are effective as long as concurrent and conflicting file sharing among clients is rare, which is often true [101].

4.3 SeMiNAS Design

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

4.3.1 Threat Model

Our threat model reflects the settings of an organization with offices in multiple locations, and employees in each office store and share files via a SeMiNAS gateway (see Figure 4.1). We discuss the trustability of the cloud, clients, and the middleware 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 [10]. 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%) [199]. Consequently, SeMiNAS currently uses a single cloud back-end; our final Kurma design in Chapter 5 will use multiple clouds for even higher availability.

Clients. Clients are 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 [158].

The Middleware. SeMiNAS is trusted. It provides centralized and consolidated security services. Physically, the middleware is a small cluster of computers and appliances, which can fit in a guarded machine room. Thus, securing the middleware is easier than securing all clients that might scatter over multiple buildings. An organization can also dedicate experienced security personnel to fortify the middleware. We also trust that only SeMiNAS gateways can authenticate themselves to the cloud NFS servers using RPCSEC_GSS [51]; therefore, adversaries cannot fake a gateway. SeMiNAS currently does not handle replay attacks, which we address later in Kurma.

4.3.2 Design Goals

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

- **High security**: SeMiNAS should ensure high integrity and confidentiality while storing and sharing data among geo-distributed clients.
- **Low overhead**: SeMiNAS should have minimal performance penalty by using a low-overhead security scheme and effectively caching data.
- **Modularity**: SeMiNAS should be modular so that more security features, such as anti-virus and intrusion detection, can be easily added in the future.
- **Simplicity**: SeMiNAS should have a simple architecture that eases development, deployment, and maintenance.

4.3.3 Architecture

SeMiNAS is a cryptographic file system that serves as a gateway between clients and remote cloud servers. SeMiNAS has a stackable file system architecture so that its security mechanisms can be easily added as layers on top of existing storage gateways and WAN accelerators. Stackable file systems, such as Linux's UnionFS [200] and OverlayFS [29], are flexible for three reasons: (1) they can intercept all file operations including `ioctl`s; (2) they can be stacked on top of any other file systems (e.g., `ext4` and NFS); and (3) the stacking can be composed in different orders to achieve a wider range of functionalities. Stackable file systems are also 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 [74, 89, 129, 201].

SeMiNAS consists of multiple gateways in geo-distributed offices that share files securely via a common storage provider. Each office has a SeMiNAS gateway, which acts as an NFS server to clients and as a client to remote NFS servers. SeMiNAS protects files transparently and stores files in ciphertext format in remote cloud servers. A client writes a file by first sending an NFS request to SeMiNAS. Then, SeMiNAS 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 verifies and decrypts the data from the ciphertext and MACs when reading from remote servers. Each file has a unique encryption key, which is secretly shared among geo-distributed offices using a PGP-like scheme. SeMiNAS combines the encryption and authentication functionality in one stackable file-system layer.

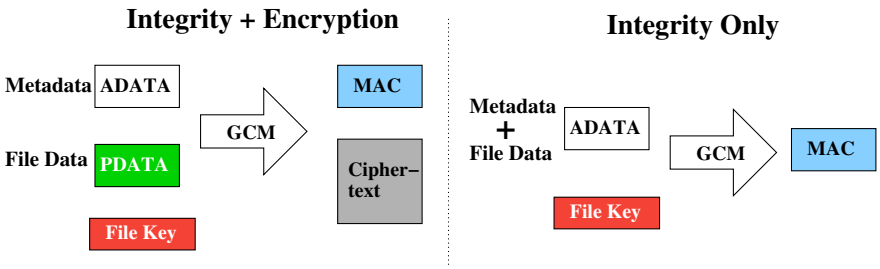


Figure 4.2: GCM for integrity and optional encryption

In addition to the security stackable layer, SeMiNAS contains another stackable file-system layer that caches file content in persistent storage devices in SeMiNAS gateways. Caching is indispensable to avoid the long latency in WANs. SeMiNAS stacks the caching layer on top of the security layer so that file content is cached in cleartext format and reading from the cache does not require decryption. Saving the file's content as cleartext in SeMiNAS is secure because SeMiNAS is fully trusted in our threat model.

4.3.4 Integrity and Confidentiality

In the security stackable file-system layer, SeMiNAS uses authenticated-encryption to simultaneously authenticate and encrypt files [195]. Authenticated-encryption is desirable for strong security because combining a separate encryption layer and an authentication layer is 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" [98]. Only one out of the three combinations (i.e., EtA) is considered to be secure [119]. The security ramifications of these combinations are rather complex [20]: even experts can make mistakes [99]. Therefore, SeMiNAS avoids separating encryption and authentication, and instead uses one of the standard authenticated encryption schemes that perform both operations simultaneously.

Out of the ISO-standardized authenticated encryption modes, we chose the Galois/Counter Mode (GCM) because of its superior performance [41] to other modes such as CCM [192] and EAX [19]. SeMiNAS strictly follows NIST's guidance of using GCM and meets the "uniqueness requirements on IVs and keys" [50].

As shown in Figure 4.2, 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 metadata as ADATA.

On write operations, GCM uses the data to be written as PDATA and additional security metadata (discussed in Chapter 4.3.4.3) 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.

4.3.4.1 Key Management

Key management is critical for strong security. SeMiNAS uses a simple yet robust key management scheme. Each SeMiNAS gateway has a master key pair, which is used for asymmetric encryption (RSA) and consists of a public key (MuK) and a private key (MrK). The public keys are exchanged among geo-distributed gateways manually by security personnel. This is feasible because one geographic office usually has only one SeMiNAS gateway, and key exchange is only needed when opening an office in a new site. This scheme has the advantages of not relying on any third-party for public key distribution. Each file has a symmetric file key (FK) and a 128-bit initialization vector (IV); both FK and IV are 128-bit long and randomly generated. To avoid reusing IVs [50], SeMiNAS adds to the IV the block offset number to generate a unique IV for each block.

Because each SeMiNAS gateway maintains the MuKs of all other gateways, the file keys (FKs) can be shared among all SeMiNAS gateways under the protection of MuKs. When creating a file, a SeMiNAS gateway (*creator*) generates an FK. Then for each SeMiNAS gateway 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 $\langle$SID, EFK$\rangle$ pair where SID is the unique ID of the accessor and EFK is the encrypted FK. All the $\langle$SID, EFK$\rangle$ pairs are then stored in the file header. With the upcoming sparse file support [77], the file header can reserve sufficiently large space with a hole following the header. Therefore, adding a new SeMiNAS gateway need only add its $\langle$SID, EFK$\rangle$ in the header by filling the hole without shifting the file data following the header. When opening a file, a SeMiNAS gateway, which needs to be an accessor of the file, first finds its $\langle$SID, EFK$\rangle$ pair in the header, and then it retrieves the file key FK by decrypting the EFK using its private key (MrK).

4.3.4.2 File-System Namespace Protection

SeMiNAS protects not only file data but also file-system metadata. SeMiNAS applies authenticated encryption to file and directory names so that attackers in a compromised cloud server could not guess sensitive data from the names. SeMiNAS also generates a key for each directory and uses the key to encrypt the names of file-system objects inside the directory. Similar to a file key, a directory key (DK) is also encrypted by SeMiNAS gateways' master key pairs. However, the encrypted directory key pairs (i.e., $\langle$SID, EDK$\rangle$) are saved in a hidden KeyMap file under the directory because we could not prepend a header to a directory (as SeMiNAS does for each file). When processing a directory creation request from a client, SeMiNAS also create the KeyMap file in addition to creating the directory in the cloud server. When processing a directory deletion request, SeMiNAS needs to delete its KeyMap file before deleting the directory in the cloud server; otherwise, an ENOTEMPTY error will occur. When accessing child objects inside a directory by names, SeMiNAS encrypts the names to obtain the real names stored in the cloud. When listing a directory, SeMiNAS decrypts the names returned from the cloud.

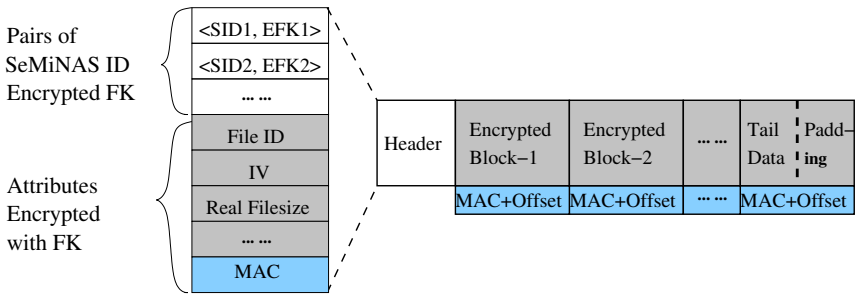


Figure 4.3: SeMiNAS metadata management

4.3.4.3 Security Metadata Management

SeMiNAS maintains per-file metadata so that files can be encrypted and secretly shared among geo-distributed offices. As shown in Figure 4.3, the most important per-file metadata is the encrypted key pairs discussed in Chapter 4.3.4.1; other per-file metadata is authenticated and encrypted file attributes including unique file ID, IV, real file size flags, etc. SeMiNAS saves the per-file metadata in a 4KB file header. More space can be reserved for the header by punching a hole in the file [77] following the header. Thus, when accommodating more ⟨SID, EFK⟩ pairs, the header can grow beyond 4KB by filling the hole without shifting any file data.

SeMiNAS divides a file into fix-sized data blocks and applies GCM to each block (with padding if necessary). Therefore, it also maintains per-block metadata including a 16-byte MAC and an 8-byte block offset (Figure 4.3). The block offset is combined with the per-file IV to generate the unique per-block IV, and is also used to detect an attack of swapping blocks. SeMiNAS can detect inter-file swapping as well because each file has a unique key. The per-block metadata is stored using DIX as detailed in Chapter 4.3.5.1.

4.3.5 NFSv4-Based Performance Optimizations

SeMiNAS leverages two advanced NFSv4 features to ensure its security scheme has low overhead: Data-Integrity eXtension (DIX) and compound procedures, discussed next.

4.3.5.1 NFS Data-Integrity eXtension

DIX gives applications access to the long-existed out-of-band channel of information in storage media. With NFSv4, NFS clients can utilize DIX to store extra information in NFS servers [137]. Figure 4.4 shows how SeMiNAS leverages DIX and stores the per-block metadata (a MAC and an offset). This is particularly beneficial in wide-area environments because it saves many extra network round trips for metadata operations.

Storing MACs and offsets using DIX is better than the other three alternative methods: (1) The first alternative is to write the concatenation of each encrypted block and its MAC as one file in the cloud. This method not only burdens file system metadata management with many small files [67], but also negates the benefits of using a file-system API such as the file-level close-to-open consistency (which means once clients close a file, all their changes to the file will be available to clients who open the file later). (2) The second alternative method is to use an extra file for all per-block metadata of each file. However, this is suboptimal, especially considering the high latency of

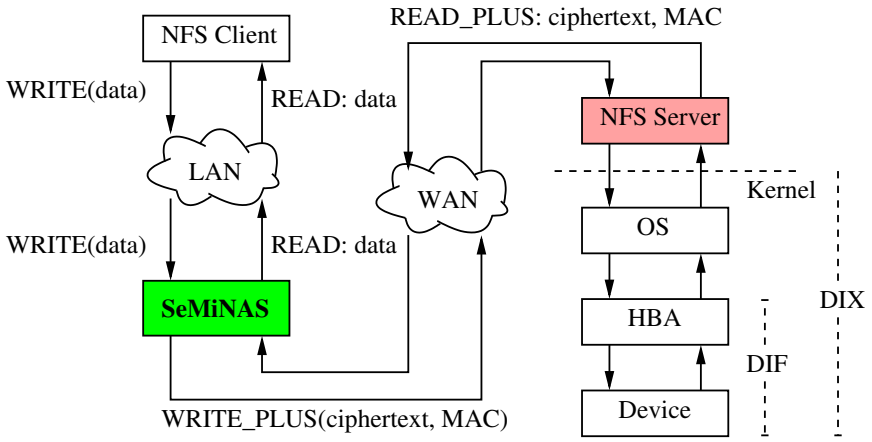


Figure 4.4: NFS end-to-end data integrity using DIX

WANs, because writing data blocks incurs an extra write request to the metadata file. (3) The third alternative is to map one block to a larger block in the cloud-stored 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 metadata). However, this method suffers from extra read-modify-update operations caused by breaking block alignment. Using a larger block size (e.g., 256KB instead of 16KB) alleviates this problem by having fewer extra read-modify-update operations, but it also makes each extra operation more expensive.

Using DIX frees SeMiNAS from all aforementioned problems. To accommodate the 24-byte per-block metadata, we require a block to be at least 2KB large, because each sector uses at least two DIX bytes by itself for an internal checksum and leaves at most six DIX bytes for applications.

4.3.5.2 Compound Procedures

To maintain the security metadata in file headers, SeMiNAS performs many extra file-system operations (e.g., a read of the header when opening a file). Sending separate NFS requests for these extra operations incurs extra WAN round trips and consequently large performance overheads. To avoid this, SeMiNAS leverages *compound procedures*—a new NFSv4 feature that combines multiple operations into one NFS request. Compound procedures can significantly shorten the average latency of operations. This is because in high-latency networks, a single multi-operation RPC takes almost the same amount of time to process as a single-operation RPC.

Extra operations on file headers are great candidates for compound procedures because all file-header operations immediately follow some other user-initiated file-system operations. By packing extra file-header operations into the same NFS request of the initiating operations, no extra requests are needed. Extra operations on the KeyMap files, each of which stores the secret key of its parent directory (see Chapter 4.3.4.2), are also good candidates for compound procedures. Specifically, SeMiNAS packs extra operations into compound procedures in the following scenarios:

- creating the header after creating a file;
- reading the header after opening a file;
- updating the header before closing a dirty file;

- reading the header when getting file attributes;
- getting the attributes (GETATTRS) after writing to a file;
- creating its KeyMap file after creating a directory;
- reading its KeyMap file after looking up a directory; and
- unlinking its KeyMap file before unlinking a directory.

Compound procedures are highly effective for SeMiNAS. We have benchmarked the optimization of compound procedures separately using Filebench's File-Server workload: compound procedures cut the performance overhead of SeMiNAS from 52% down to merely 5%.

4.3.6 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" [164] before replying. Because the NFS protocol demands stable writes to survive server crashes, the cache layer also maintains additional metadata in stable storage to ensures correct crash recovery. The metadata includes a list of dirty files and a per-block dirty flag to distinguish dirty blocks from clean blocks.

For each cached file, SeMiNAS maintains a sparse file of the same size in the gateway'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 metadata 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.

4.4 SeMiNAS 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` [203] cases that are applicable to NFS. We present the technical background and the implementation of the security and caching features.

4.4.1 NFS-Ganesha

Our SeMiNAS prototype is based on NFS-Ganesha [43, 44, 136], 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 gateway.

Like their stackable counterparts in Linux [206], 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. NFS-Ganesha originally allowed only one stackable layer; we added the support of multiple stackable layers [140]. 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 separately.

4.4.2 Authenticated Encryption

We implemented SeMiNAS's authenticated encryption in an FSAL called `FSAL_SECNFS`. We used `cryptopp` as our cryptographic library because it supports a wide range of cryptographic schemes such as AES, GCM, and VMAC [180]. We used AES as the block cipher for GCM. We implemented the NFS DIX in NFS-Ganesha so that ciphertext and the security metadata can be transmitted together between SeMiNAS gateways and the cloud. First, we implemented the READ_PLUS and WRITE_PLUS operations of NFSv4.2 [77] so that the out-of-band DIX bytes can be transfered together with file block data in one request. Then, at the gateway side, we changed `FSAL_PROXY` to use READ_PLUS and WRITE_PLUS for communications with the cloud NFS server (Figure 4.4). Lastly, at the cloud side (running `FSAL_VFS`), we changed `FSAL_VFS` to use WRITE_PLUS and READ_PLUS, and to write the ciphertext and security metadata together to storage devices. Currently, Linux does not have system calls to pass file data and their DIX bytes from user space to kernel; so we used a DIX kernel patchset [144] after we fixed its bugs.

We implemented `FSAL_SECNFS` carefully to avoid any side effects caused by the security metadata. For example, updating metadata in a file header has the side effect of changing the file's `ctime` and `mtime`, with an unexpected consequence of invalidating NFS clients' page cache and hurting performance: an NFS client uses `ctime` to check the validity of an NFS file's page cache; an external change of `ctime` implies the file has been modified by another NFS client, and demands the client to invalidate the cache to prevent reading stale data. To avoid this inadvertent cache invalidation, `FSAL_SECNFS` maintains the effective `ctime` and `mtime` in the file header instead of using the real `ctime` and `mtime` attributes of the file.

We also implemented two additional performance optimizations in `FSAL_SECNFS`: (1) We cache the file key (FK) and the ⟨SID, EFK⟩ pairs in memory to reduce the frequency of expensive RSA decryptions of FKs. This is secure because `FSAL_SECNFS` runs in the trusted SeMiNAS gateways. (2) We use the faster VMAC [41, 180] (3.2× faster on our testbed) instead of GCM when only integrity (but not encryption) is required.

4.4.3 Caching

The persistent caching file-system layer of SeMiNAS is implemented as another FSAL named `FSAL_PCACHE`. Because `FSAL_PCACHE` needs to write back dirty data using separate threads, we implemented `FSAL_PCACHE` on top of a home-built external caching library to avoid complicating NFS-Ganesha's threading model. The caching library provides caching (lookup, insert, invalidate, etc.) and write-back APIs for `FSAL_PCACHE`. When inserting dirty blocks of a file into the cache using this library, `FSAL_PCACHE` registers a write-back callback function along with the dirty buffer to the library. The callbacks are invoked periodically as long as the file remains dirty. When closing a file, `FSAL_PCACHE` calls the write-back function directly, and deregisters the callback to the library.

4.4.4 Lines of Code

The implementation of our SeMiNAS prototype took about 25 man-months (of serveral graudate students over 3 years), and added around 14,000 lines of C/C++ 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 have also fixed bugs in the DIX kernel patchset. We plan to release all code as open source in the near future.

4.5 SeMiNAS Evaluation

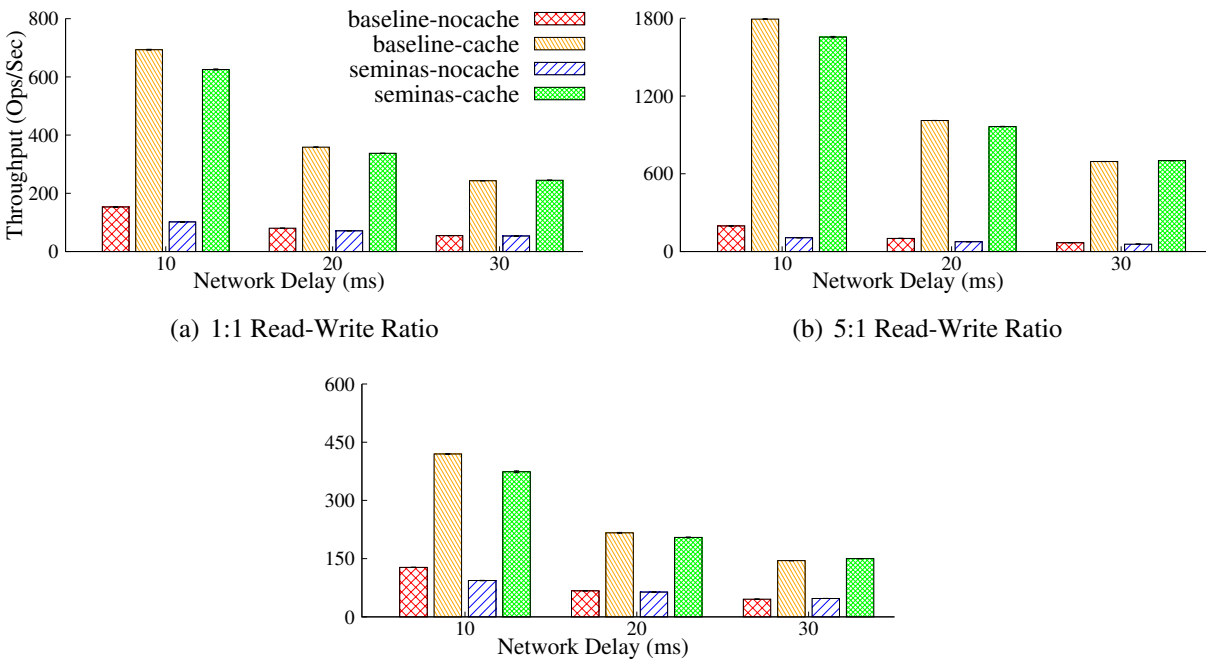
We now present the evaluation of SeMiNAS under different workloads, security polices, and network settings.

4.5.1 Experimental Setup

Our testbed consisted of seven Dell R710 machines running CentOS 7.0 with a 3.14 Linux kernel. Each machine has a six-core Intel Xeon X5650 CPU, a Broadcom 1GbE NIC, and an Intel 10GbE NIC. Five machines run as NFS clients and each of them has 1GB RAM. Both remaining machines have 64GB: one of them runs as a SeMiNAS gateway and the other emulates a cloud NFS server. Clients communicated to the gateway using the 10GbE NIC, whereas the gateway communicated to the server using the 1GbE NIC (to simulate a slower WAN). The average RTT between the clients and the SeMiNAS gateway is 0.2ms. The SeMiNAS gateway 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 better emulate the network between the SeMiNAS gateway and the cloud, we injected 10–30ms delays 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 [144] (with our bug fixes) that allows DIX bytes to be passed from user space to kernel.

Physical storage devices that support DIX are still rare [76], so we had to set up a 20GB DIX-capable virtual SCSI block device backed by RAM using `targetcli` [135]. Using RAM, instead



(a) 1:1 Read-Write Ratio

(b) 5:1 Read-Write Ratio

(c) 1:5 Read-Write Ratio

Figure 4.5: Aggregate throughput of baseline and SeMiNAS with 4KB I/O size, 5 NFS clients, and one thread per client under different read-write ratios and network delays.

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 [141] and Memcached [57]. If storage latency in the cloud was counted, the extra latency added by SeMiNAS's security mechanisms 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 integrity, 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 encryption, we manually confirmed that file data was an unreadable ciphertext when reading directly from the server, but its plaintext was identical to what was written by clients.

We used the vanilla `FSAL_PROXY` as baseline. `FSAL_PROXY` uses up to 64 concurrent requests each with a 2MB-large RPC buffer. We benchmarked two cases—with and without the persistent cache (`FSAL_PCACHE`) for both the baseline and SeMiNAS. We benchmarked a set of simple synthetic micro-workloads, and Filebench [56] macro-workloads including the NFS Server, Web Proxy, and Mail Server.

4.5.2 Micro-Workloads

We benchmarked SeMiNAS using three micro-workloads: (1) random file accesses with different read-write ratios, (2) file creation, and (3) file deletion.

4.5.2.1 Read-Write Ratio Workload

Read-write ratio is an important workload characteristic that influences the performance impact of SeMiNAS's security mechanisms and the persistent cache (`FSAL_PCACHE`). We studied read-write ratios from write-intensive (1:5) to read-intensive (5:1) to cover common ratios in real workloads [101, 156]. We pre-allocated 100 1MB-large 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 4KB reads and m 4KB writes at random offsets, and close it. We varied n and m to control the read-write ratio. We also ensured $n + m$ is a constant (i.e., 60) so that dirty contents are written back in the same frequency.

Figure 4.5 shows the results when the read-write ratios are 1:1, 5:1, and 1:5. Overall, the configurations with caching outperform their "nocache" counterparts. For the 1:1 read-write ratio, caching speeds the workloads up by 4–6×. The degree of speed-up grows to 9–16× as the workload becomes read-intensive (Figure 4.5(b)), but drops to 3–4× as the workloads become write-intensive (Figure 4.5(c)). The cache's help to writes is smaller than to reads because SeMiNAS has to write dirty writes back to the cloud server upon file close, so that clients in other offices can observe the latest changes.

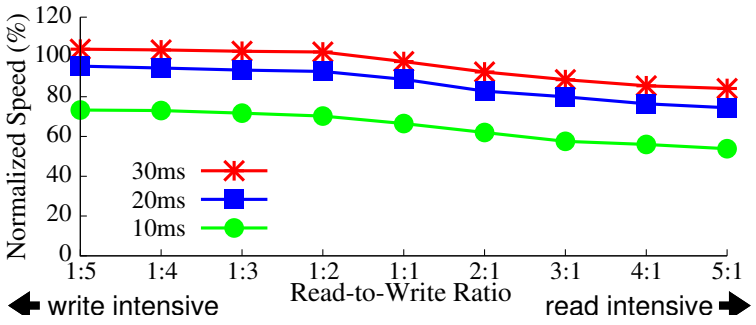
To better illustrate the performance impact of SeMiNAS, we show SeMiNAS's relative throughput to the baseline in Figure 4.6. When it is write-intensive, SeMiNAS can be up to 3% faster than the baseline regardless of the presence of `FSAL_PCACHE`. This is because the baseline uses extra COMMITs following WRITEs to make write operations stable, whereas SeMiNAS does so by simply setting the stable flag of WRITE_PLUS requests. The normalized throughput of SeMiNAS drops as the workload becomes more read-intensive (Figure 4.6(a)) for two reasons: (1) the effect of saving COMMITs becomes smaller as the number of writes goes down; and (2) SeMiNAS has to authenticate and decrypt (encrypt) data when reading from (writing to) the cloud server.

When cache is on (Figure 4.6(b)), the normalized throughput decreases much slower and is almost flat. This is because (1) the cache content is in plaintext format and reading from cache needs no more authentication or decryption; and (2) writes are acknowledged once dirty data is inserted into the cache and the real write-back happens asynchronously.

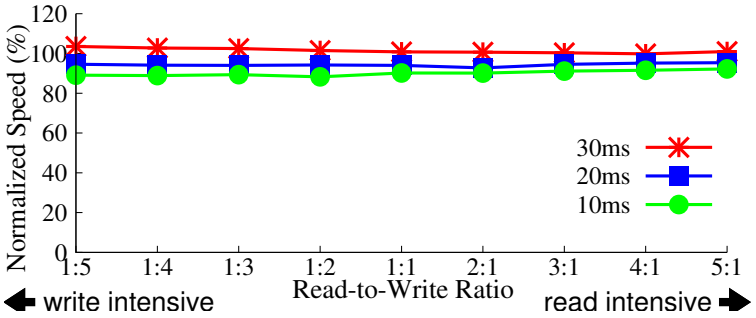
Note that the normalized throughput of SeMiNAS is better for longer network delay no matter if the cache is on or off. This is because SeMiNAS is optimized for wide-area environments and minimizes the number of round trips between the gateway and the cloud.

4.5.2.2 File-Creation Workload

Depending on the number of threads, SeMiNAS has different performance impact over the baseline for file creation. As shown in Figure 4.7, SeMiNAS has only negligible performance impact when there are only one or ten threads. Surprisingly, SeMiNAS makes file creation 35% faster than the baseline when the number of threads grows to 100. This is caused by the TCP connection between the gateway and the server, particularly due to the TCP Nagle algorithm [194]. The algorithm adds extra delay to outbound packets in the hope of coalescing multiple small packets



(a) Persistent Cache (FSAL_PCACHE) Off



(b) Persistent Cache (FSAL_PCACHE) On

Figure 4.6: Relative throughput of SeMiNAS to the baseline under 10ms, 20ms, and 30ms network delays.

into fewer, larger ones; TCP Nagle trades off latency for bandwidth. This trade-off hurts the baseline performance of this file-creation workload, which is metadata intensive and generates many small network packets. In contrast, the algorithm favors SeMiNAS because SeMiNAS uses compound procedures to pack file creations and extra secure operations (e.g., creating file headers) together to form larger packets.

The number of threads influences the performance because all threads share one common TCP connection between the gateway and the server. More threads bring more coalescing opportunities; otherwise, the extra waiting of TCP Nagle is useless if the current request is blocked and no other requests are coming. To verify this explanation, we temporarily disabled TCP Nagle by setting the TCP_NODELAY socket option, and observed that SeMiNAS's throughput became about the same (99%) as the baseline thereafter.

Figure 4.7 also shows that, as expected, the persistent cache (FSAL_PCACHE) does not make a difference in file creation because FSAL_PCACHE caches only data, but not metadata.

4.5.2.3 File-Deletion Workload

Figure 4.8 shows the results of deleting files, where SeMiNAS have the same throughput as the baseline with and without the persistent cache. This is because SeMiNAS does not incur any extra operations upon file deletion. However, adding FSAL_PCACHE makes file deletion 12–18% slower. This is because FSAL_PCACHE needs one extra lookup operation to delete a file. FSAL_PCACHE uses file handles as unique keys of cached content, but the file deletion function

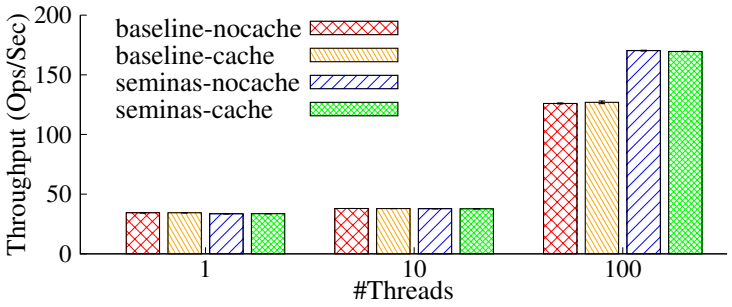


Figure 4.7: Throughput of creating empty files in a 10ms-delay network with one NFS client.

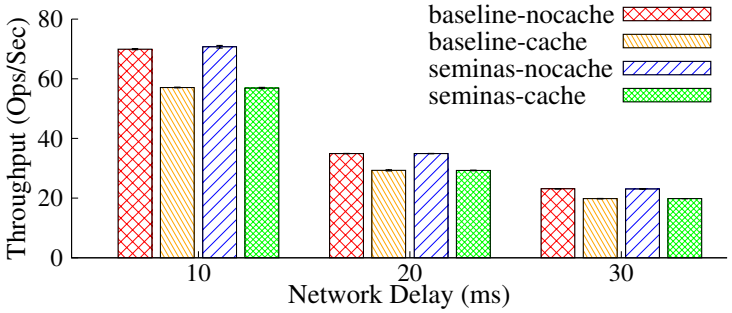


Figure 4.8: Throughput of deletion of 256KB files, with one NFS client and 100 threads.

(i.e., `unlink`) uses the parent directory and file name, rather than the file handle, to specify the file. Those extra lookups could be saved if `FSAL_PCACHE` maintains a copy of the file-system namespace, which we left as future work.

4.5.3 Macro-Workloads

We evaluated SeMiNAS using three Filebench macro-workloads: (1) NFS Server, (2) Web Proxy, and (3) Mail Server.

4.5.3.1 Network File-System Server Workload

Filebench's NFS-Server workload emulates the I/O activities experienced by an NFS server. We used the default settings of the workload, which contains 10,000 1KB-to-1700KB-large files totaling 2.5GB. The read sizes of the workload range from 8K to 135K with 85% reads 8KB-large; the write sizes range from 9K to 135K with 50% writes 9KB- to 15KB-large. The workloads perform a variety of operations including open, read, write, append, close, create, and delete.

Figure 4.9 shows the results of running this workload. Without cache, the baseline gateway's throughput decreases from 72 ops/sec to 26 ops/sec as the network latency between the gateway and the server increased from 10ms to 30ms. After adding the persistent data cache, the baseline throughput increases but only slightly. The performance boost of caching is small because the workload contains many metadata operations that cannot be cached; for example, open and close operations have to talk to the server in order to maintain close-to-open consistency.

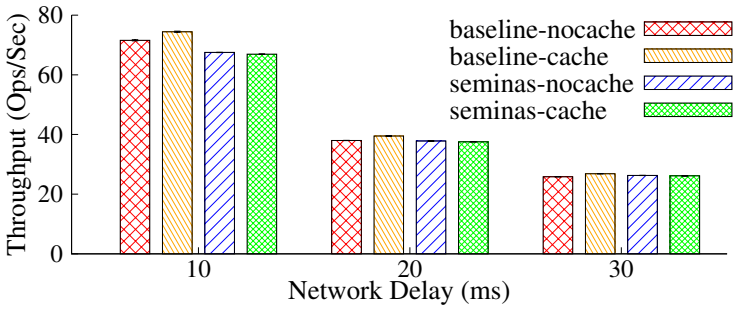


Figure 4.9: Throughput of Filebench NFS-Server workload, with one benchmarking client and one thread.

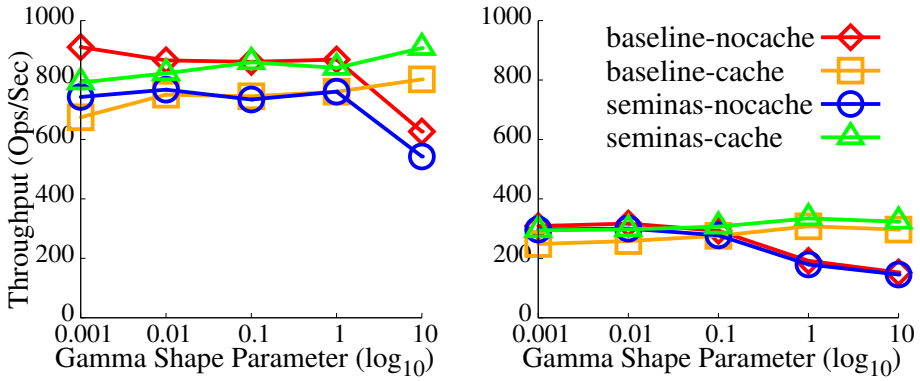
In this NFS-Server workload, SeMiNAS is between 6% slower and 2% faster than the baseline without cache; SeMiNAS is 3–10% slower with cache enabled. As the network delay grows, the performance penalty of SeMiNAS becomes smaller regardless of the presence of cache. This is because we optimized SeMiNAS for wide-area environment by minimizing the number of round trips between the gateway and the cloud server.

We noticed that adding cache to SeMiNAS actually makes the performance slightly worse (the last two bins in each group of Figure 4.9). This is because `FSAL_PCACHE` makes file deletions slower with extra lookups (see Chapter 4.5.2.3), and file deletions count for as much as 8% of all WAN round trips in this workload. The extra lookups incurred by file deletions are also one of the reasons why the cache's performance boost to the baseline is small, although a lookup in the baseline is cheaper than in SeMiNAS (because SeMiNAS needs extra bookkeeping during lookups).

4.5.3.2 Web-Proxy Workload

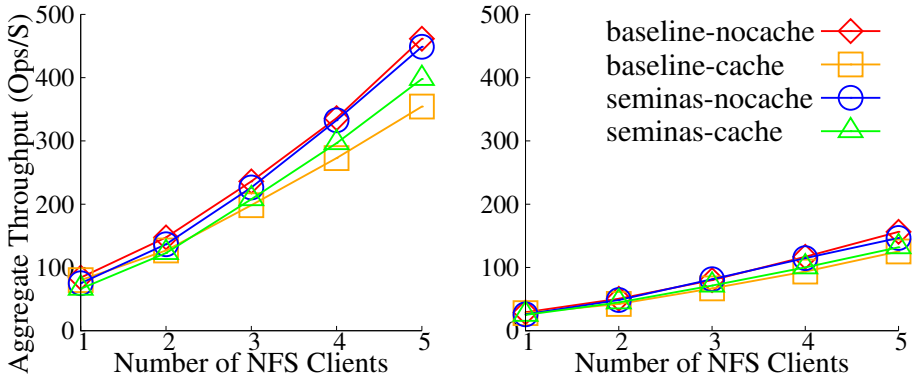
Filebench's Web-Proxy workload emulates the I/O activities of a simple Web-Proxy server, which fits well with SeMiNAS's gateway architecture. The workload has a mix of file creation, deletion, many open-read-close operations, and a file append operation to emulate logging. The default Web-Proxy workload has 10,000 files with an average size of 16KB in a flat directory, and 100 benchmarking threads. We made three changes to the default settings: (1) we placed the files in a file-system directory tree with a mean directory width of 20 because a flat directory made the baseline so slow (around 20 ops/sec) that SeMiNAS did not show any performance impact at all; (2) we enlarges the average file size to 256KB so that the working set size (2.56GB) is more than twice the size of the NFS client's RAM (1G) but smaller than the size of the persistent cache; and (3) we used a Gamma distribution [193, 196] to control the access pattern of the files, but varied the Gamma's shape parameter (k) to emulate access patterns with different degrees of locality.

Figure 4.10 shows the Web-Proxy workload results. With 10ms network delay, the throughput of "baseline-nocache" drops from 910 to 630 ops/sec as the degree of workload locality decreases. The "seminas-nocache" curve in Figure 4.10(a) has a similar shape to its baseline counterpart, but at 11–18% lower throughputs as a result of extra security mechanisms in SeMiNAS. With high locality ($k <= 1$), adding `FSAL_PCACHE` (blue circle curve) actually slows down the baseline (red diamond curve) because (1) `FSAL_PCACHE` is not useful when most reads are served from the client's page cache; and (2) `FSAL_PCACHE` also introduces extra overhead for file deletions.



(a) 10ms Network Delay (b) 30ms Network Delay

Figure 4.10: Web-proxy results with different access patterns, one NFS client, and 100 threads. A larger value of the shape parameter means less locality in the access pattern.



(a) 10ms Network Delay (b) 30ms Network Delay

Figure 4.11: Filebench Mail Server throughput.

Conversely, as the locality drops ($k = 10$), the client's page cache becomes less effective and the persistent cache, which is larger than the working set size, becomes effective.

Figure 4.10(a) shows that SeMiNAS actually makes the workload up to 15% faster than the baseline when there is a cache (i.e., the green triangle curve is higher than the orange rectangle curve). This is because SeMiNAS makes file creations faster in this highly-threaded workload thanks to the TCP Nagle algorithm (see Chapter 4.5.2.2).

For a slower network of 30ms latency (Figure 4.10(b)), the throughputs of baseline and SeMiNAS are both slower than in the faster network (10ms). However, the relative order of the four configurations remains the same. Without the cache, SeMiNAS has a small performance penalty of 4–6%; with the cache, SeMiNAS sees a performance *boost* of 9–19%.

4.5.3.3 Mail-Server Workload

Filebench's Mail-Server workload emulates the I/O activity of an `mbox`-style e-mail server that stores each e-mail in a separate file. The workload consists of 16 threads, each performing create-

append-sync, read-append-sync, read, and delete operations on a fileset of 10,000 16KB files.

We used this Mail-Server workload to test the scalability of SeMiNAS by gradually increasing the number of NFS clients. As shown in Figure 4.11, both the baseline and SeMiNAS scales well as the number of clients grows. The relative order and trend of the four curves in Figure 4.11 share similarity with the curves of the Web-Proxy workload results for similar reasons. In terms of relative speed to the baseline, SeMiNAS is 1–11% slower without cache, and is between 17% slower (fewer clients) and 12% faster (more clients) with cache depending on the network delay and effectiveness of TCP Nagle algorithm (see Chapter 4.5.2.2).

4.6 Related Work of SeMiNAS

SeMiNAS is related to (1) secure distributed storage systems, (2) cloud NAS, and (3) cloud storage gateways.

4.6.1 Secure Distributed Storage Systems

SFS [120], Cepheus [94], SiRiUS [70], and SUNDR [104] are cryptographic file systems that provide end-to-end file integrity and confidentiality with minimal trust on the server; but they all used remote servers as block stores instead of file-system servers, and none of them took advantage of NFSv4, which was not invented at the time. SeMiNAS' sharing of file keys (FK) is similar to SiRiUS [70]. However, because access control is enforced by the trusted middleware, SeMiNAS needs only one key per file instead of two (one for reading and the other for writing) in SiRiUS. NASD [69] and SNAD [127] 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.

4.6.2 Cloud NAS

Panache [112] is a parallel file-system cache that enables efficient global file access over WANs but without WAN's fluctuations and latencies. It uses pNFS to read data from remote cloud servers 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 [85], SoftNAS [169] and Zadara Storage [205]. These services focus on providing file system services in public clouds. These cloud NAS service providers control and trust the ultimate storage devices, whereas SeMiNAS cannot control or trust the devices. FileWall [168] combines the idea of network firewalls 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). FileWall can protect cloud NAS servers from malicious clients, whereas SeMiNAS is for protecting clients from clouds.

4.6.3 Cloud storage gateways

Using the cloud as back-end, a cloud gateway gives a SAN or NAS interface to local clients, and can provide security and caching features. There are several cloud gateway technologies, in both industry and academia. In academia, Hybris [47], BlueSky [189], and Iris [173] 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 [133] is a cloud integrated storage for backup. Riverbed SteelFusion [154] 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.

4.7 SeMiNAS Conclusions

We presented the design, implementation, and evaluation of SeMiNAS, a secure middleware using cloud NAS as back-end. SeMiNAS provides end-to-end data integrity and confidentiality while allowing files to be securely shared among geo-distributed offices. SeMiNAS uses authenticated encryption to safely and efficiently encrypt data and generate MACs at the same time. SeMiNAS is optimized for WANs and has a persistent cache to hide high WAN latencies. SeMiNAS leverages advanced NFSv4 features, including Data Integrity eXtention (DIX) and compound procedures, to manage its secure metadata without incurring extra network round trips. Our benchmarking with Filebench workloads showed that SeMiNAS has a performance penalty less than 18%, and occasionally improve performance by up to 19% thanks to its effective use of compound procedures.

4.7.1 Limitations

SeMiNAS does not handle attacks based on side-channel information or file-access patterns. SeMiNAS is vulnerable to replay attacks, which usually requires building a Merkle tree [124] for the entire file system and is thus expensive in WANs. We are developing an efficient scheme to thwart replay attacks. We solve all these limitations in Chapter 5.

Chapter 5

Kurma: Multi-Cloud Secure Gateways

5.1 Kurma 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 gateways 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 gateways. 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%) [199], availability and business continuity remain the largest obstacles for cloud computing [10]. A single cloud provider is itself a single point of failure [10]; 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 gateway server; Kurma eliminates this by storing metadata 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 metadata and detecting replay attacks. SeMiNAS encrypts only file data but not file system metadata 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 metadata. In contrast, Kurma saves on the cloud only encrypted data blocks, but not any file system metadata whatsoever. Moreover, SeMiNAS's vulnerability to replay attacks is fixed by Kurma. Kurma keeps file system metadata on premises and replicates the metadata across the gateways (regions) via secure communication channels. A part of the replicated metadata 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 4 that

SeMiNAS's NFS file system consistency among geo-distributed gateways 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 (Chapter 4.5), 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 gateways, Kurma maintains NFS's close-to-open consistency at only the gateway (or regional) level. That is, NFS operations are synchronized with operations to the same gateway 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 gateways still share a common namespace by asynchronously replicating gateway-level changes to other gateways. Without overall consistency, however, the asynchronous replication may cause conflicts, which Kurma has to resolve automatically or with end users' intervention.

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 [137], which simplifies management of security metadata (see Chapter 4.3.4.3), 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. Chapter 5.2 introduces Kurma's supporting systems. Chapter 5.3 and Chapter 5.4 discuss Kurma's design and implementation, respectively. Chapter 5.5 evaluates its security and performance. Chapter 5.6 studies related work. Chapter 5.7 concludes this chapter.

5.2 Kurma Background

In addition to NFS-Ganesha introduced in Chapter 4.4, 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.

5.2.1 ZooKeeper: A Distributed Coordination Service

Apache ZooKeeper [83] is a distributed coordination service. ZooKeeper achieves consensus among distributed systems using an algorithm called ZAB, short for ZooKeeper Atomic Broadcast [90]. ZooKeeper is popular and regarded as "The King of Coordination" [18]. 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 [62]. Although ZooKeeper is implemented in Java, it provides both C and Java APIs to clients. ZooKeeper also has a helper library called Apache Curator [59]. 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 [59] to simplify the programming of ZooKeeper.

5.2.2 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" [60]. 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 metadata management, and uses BookKeeper [58], 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 gateway asynchronously propagates changes to the namespace to other gateways 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. Hedwig also supports customizable traffic throttling.

5.2.3 Thrift: A Cross-Language RPC Framework

Apache Thrift is cross-language RPC framework for scalable cross-language services development [61]. 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 [63].

Kurma uses Thrift for two purposes: (1) defining messages stored in ZooKeeper and replicated among gateways, and (2) implementing RPC communication between the Kurma services running in a gateway. 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.

5.3 Kurma Design

Kurma has the same threat model as SeMiNAS; the model is described in Chapter 4.3.1. This section presents other aspects of Kurma design including its design goals, architecture, consistency model, caching, file system partition, and security features.

5.3.1 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 four design goals by descending importance:

- **Strong security**: Kurma should ensure confidentiality, integrity, and freshness to both file data and metadata while outsourcing storage to clouds.
- **High availability**: Kurma should have no single point of failure, and be available despite network partitions and outage of a small subset of clouds.
- **High performance**: Kurma should minimize the performance penalty of its security features, and overcome the high latency of remote cloud storage.
- **High flexibility**: Kurma should be configurable in many aspects to support flexible trade-off among security, availability, performance, and cost.

5.3.2 SeMiNAS Architecture

Kurma's architecture is analogous to SeMiNAS: on-premises gateways (trusted) acts as security bridges between trusted clients and untrusted public clouds. However, Kurma has three major architectural differences, as illustrated in Figure 5.1. First, each geographic region has a distributed gateway instead of a centralized one. A conceptual Kurma gateway consists of a cluster of machines which are properly coordinated using ZooKeeper. This distributed gateway avoids any single point of failure and enjoys better scalability and availability.

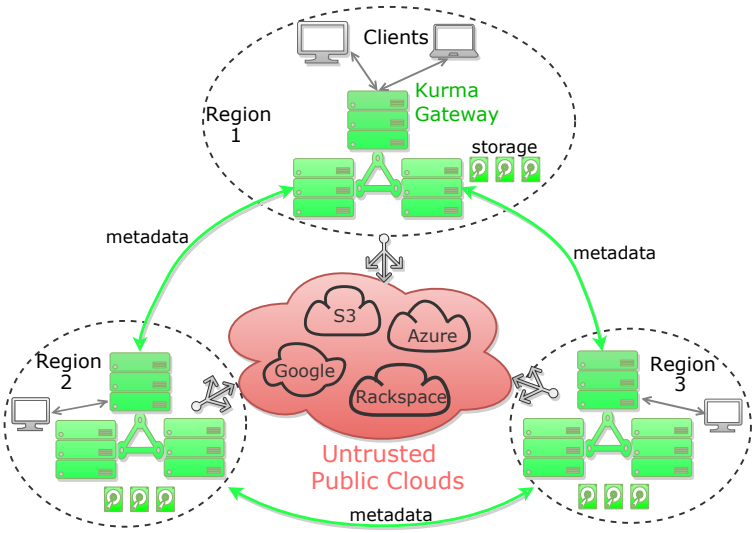


Figure 5.1: Kurma architecture when there are three gateways. Each dashed oval represents an office in a region, where there are clients and a Kurma gateway. Each gateway is a cluster of coordinated machines represented by three inter-connected racks. The green arrows connecting gateways are private secret channels for replicating file-system and security metadata. Each gateway has directly attached storage to cache hot data. Clocks of all machines are synchronized using NTP [128].

Second, Kurma gateways in geo-distributed regions are interconnected. In SeMiNAS, gateways 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 gateways, secret file system metadata 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.

5.3.2.1 Gateway components

Each Kurma gateway has three types of servers. These servers are loosely coupled and use RPC or network sockets for mutual communication. They can be deployed in many machines, so that Kurma enjoys the flexibility of adjusting the number of specific servers according to load. Figure 5.2 shows the Kurma servers and their components. NFS Servers export Kurma files to clients via NFS; each NFS Server has a persistent Cache Module that holds hot data (see Chapter 4.3.6). To process metadata requests and uncached data requests, each NFS Server uses its Gateway Module to talk to Gateway Servers' FS Module using RPCs defined using Apache Thrift [61].

Gateway Servers break files into blocks, and use its Cloud Module to store encrypted blocks as key-value objects in clouds. The Cloud Module also performs replication, erasure coding, and

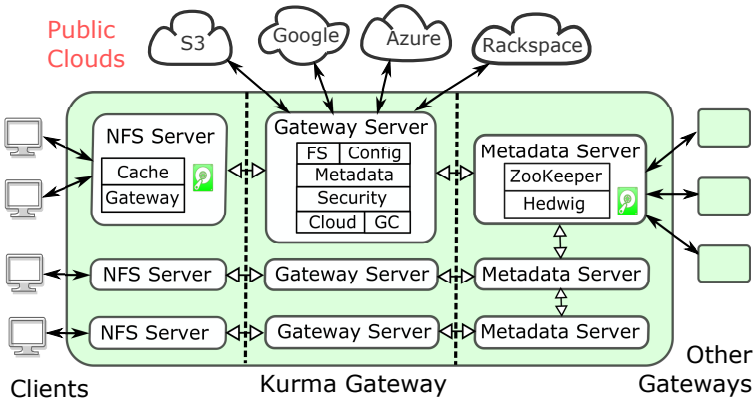


Figure 5.2: Kurma gateway components. A gateway consists of three types of servers as separated by dashed lines: NFS, Gateway, and Metadata Servers. Each NFS Server has a persistent Cache Module and a Gateway Module. Each Gateway Server has six modules: file system (FS), configuration (Config), metadata, security, cloud, and garbage collection (GC). Each Metadata Server has a ZooKeeper Module and a Hedwig Module. NFS Servers and Metadata Servers have local storage for data cache and metadata backups, respectively.

secret sharing if configured to do so. Each Gateway Server also has a Config Module that parses configuration parameters, a Security Module that performs authenticated encryption of each block, and a Garbage Collection (GC) Module that deletes stale data blocks from clouds and stale metadata from metadata servers. Gateway Servers also detect and resolve file-system conflicts (detailed in Chapter 5.3.6).

Metadata Servers run ZooKeeper to store the metadata (e.g., attributes, block versions) of each file-system object as a *znode*—a ZooKeeper data node. They also run Apache Hedwig [60] to receive messages of metadata updates from other gateways. For each message, Hedwig notifies a responsible Gateway Server to apply the metadata update in the local gateway. The Gateway Servers' Metadata Module communicates with Metadata Servers using ZooKeeper and Hedwig APIs.

Kurma groups file-system objects into volumes and assigns an NFS Server and a Gateway Server to each volume. Each volume is a separate file-system tree that can be exported via NFS to clients. Splitting Kurma services by volumes simplifies the load balancing over servers. Kurma also uses ZooKeeper to coordinate the assignment of volumes to servers. A client can mount to any NFS Server, which will either process the client's NFS requests, or redirect the requests to the responsible NFS Server using NFSv4 referrals [165].

5.3.3 Metadata Management

Kurma stores file-system metadata in Metadata Servers. Compared to storing metadata also in clouds, this is not only safer but also faster because metadata operations do not incur slow cloud accesses. Since each Kurma gateway maintains a full replica of the entire file-system metadata, Kurma does not need to synchronize with other gateways when processing metadata operations. Note that each ZooKeeper instance runs completely inside one gateway and does not communicate directly with gateways in other regions. Therefore, Kurma's metadata operations are free of any

```
typedef  i16  GatewayID
struct ObjectID {
  i128              id;
  GatewayID         creator;
  byte              type;
}
struct Attributes {
  i64               size;
  i32               flags;
      ... ...
  i64       remote_ctime;
}
```

```
struct File {
  ObjectID                  id;
  ObjectID                  parent;
  Attributes                attrs;
  i32                       block_shift;
  list<i64>                 block_versions;
  list<GatewayID>           block_creators;
  string                    redundancy;
  list<string>              cloud_ids;
  map<GatewayID, binary> keymap;
}
```

Figure 5.3: Simplified Kurma data structures in Thrift's interface definition language. `i16` is a 16-bit integer. Thrift does not have a native `i128`, so we emulated it using two `i64`s. `list` and `map` are builtin linear and associative containers, respectively. We omit common attributes such as `mode`, `uid`, and other data structures for directories and volumes.

WAN traffic and thus fast. Each Kurma gateway asynchronously replicates metadata changes to all other gateways, detecting and resolving conflicting updates (see Chapter 5.3.6).

As shown in Figure 5.3, Kurma's file-system metadata format is defined using Apache Thrift [61]. Thrift can generate Java and C/C++ data structures from its interface definition language. Kurma uses a unique 16-bit integer named `GatewayID` to represent a gateway. Kurma identifies each file-system object using a unique `ObjectID` that contains an 128-bit integer `id`, the gateway that created the object, and the object type (file, directory, or link). Using a 128-bit integer for `id` is enough for one billion machines to create one billion files per second for more than 10,000 billion years, so Kurma does not reuse `ObjectID`s. Each Kurma gateway keeps in ZooKeeper the largest `id` that has been used so far. To avoid updating the `id` in ZooKeeper too frequently, Kurma always allocates N `id`s at a time, where N defaults to 1,024. The `creator` in `ObjectID` can distinguish objects that happened to be created simultaneously in multiple gateways with the same `id`. Therefore, an `ObjectID` uniquely identifies a file-system object across all gateways.

In addition to common attributes (`size`, `mode`, `uid`, timestamps, etc.), each Kurma file-system object has extra attributes including a set of flags and a timestamp of the last update by any remote gateway. One example of such flags is to indicate if a file-system object is only visible in one gateway but not in other gateways; this flag is helpful for resolving conflicts among gateways.

Figure 5.3 also shows the structure of a Kurma file. It includes `ObjectID`s for itself and its parent, attributes, and the block shift (i.e., log_2 of the block size). For each data block, the file structure records the block version, and the `GatewayID` of the gateway that creates the version of the block. The file structure also records the redundancy configuration and cloud providers of the file. The redundancy string identifies the redundancy type of the file and its parameters (e.g., "r-4" means replication with four replicas). `cloud_ids` represent the cloud storage providers and buckets; for example, "S3a" represents a bucket named "a" in Amazon's AWS S3. There is also a `keymap` field which we detail in Chapter 5.3.4.

Storing metadata in distributed and highly-available ZooKeeper makes Kurma resilient to machine and disk failures. Kurma's metadata is also durable because ZooKeeper saves transaction logs and snapshots in persistent storage [83]. ZooKeeper keeps as much of its metadata (ideally all) in memory to speed up metadata operations; keeping file-system metadata in memory is a vi-

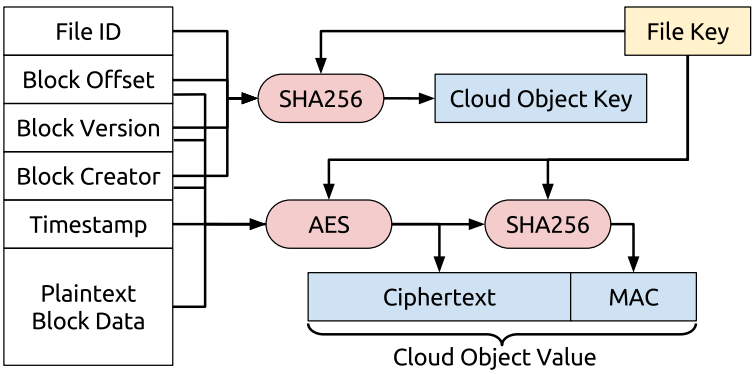


Figure 5.4: Authenticated encryption of a Kurma file block. AES runs in CTR mode without padding. The initialization vector of AES is the concatenation of the block offset and version.

able solution also used by HDFS [25] and GFS [68]. Kurma minimizes the memory footprint of its metadata using three strategies:

1. Kurma uses a large block size so that files have fewer blocks and thus less metadata. The default block size is 1MB and can be configured to be larger. Using a large block size for cloud storage is beneficial in throughput because cloud stores are often bottlenecked by network latency instead of bandwidth. A large block size can also save costs because some cloud storage providers (e.g., AWS and Azure) charge by request counts instead of sizes. A large block size can further save costs using a write-back cache, which can coalesce many small writes into a single, larger cloud request.

2. Kurma uses only a 64-bit version number and a 16-bit `GatewayID` for each block. Kurma generates an unique block key on the fly when storing the block into cloud key-value stores (see Figure 5.4). Kurma stores other necessary per-block metadata (e.g., the offset and timestamp) in clouds instead of in ZooKeeper.

3. Kurma also compresses its metadata. The block version numbers, which is the largest metadata for large files, are particularly compressible because most version numbers are small and neighboring versions are often numerically close due to locality. Our study of a large set of NFS traces [7] shows that version numbers of files (larger than 100MB) have an average compression ratio of 4:1 if using a 1MB block size with LZO level-1 [107, 139]. That means that a 1TB file's version numbers take only 2.5MB (i.e., $1T/1M \times (8+2)/4 = 2.5M$).

Minimizing metadata size also reduces the amount of metadata that needs to be replicated among gateways so that bandwidths of the inter-gateway channels do not become a bottleneck of the whole system.

5.3.4 Security

Kurma provides integrity and confidentiality as SeMiNAS does. 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 metadata including file system tree structure, and file access patterns.

Kurma stores each encrypted block as a cloud key-value object: the key of the cloud object is derived from the block's metadata; the value is a concatenation of ciphertext and message authentication code (MAC) as shown in Figure 5.4. Kurma's authenticated encryption uses the secure encrypt-then-authenticate scheme [119].

5.3.4.1 Data integrity

In addition to confidentiality and authenticity, Kurma also protects data integrity by embedding encrypted block metadata into each cloud object. As shown in Figure 5.4, the encrypted metadata includes the offset, version, creator, and timestamp of a block. Kurma uses the offset to detect attacks of swapping two blocks of the same file; Kurma can also detect inter-file block swapping because each file has a unique key. Kurma uses the version and the creator to detect replay attacks, and uses the timestamp to estimate the length of stale time when data freshness could not be guaranteed.

Data freshness is important because clouds may return stale data due to eventual consistency or malicious attacks. Checking data freshness requires a file reader (i.e., a client that reads the file) to be notified of the latest version of the file. It is difficult for a file writer to notify all file readers of the write because of the large number of clients. However, Kurma has only one gateway in each region, so it is feasible for gateways to notify each other of file updates. Kurma replicates block versions among all gateways, and uses the versions to help data freshness in three ways.

First, by incorporating a block's version number into its cloud object key, updating blocks does not overwrite existing cloud objects but always creates new objects instead. When a Kurma gateway reads a block during an inconsistency window of an eventually consistent cloud, the gateway, instead of reading stale data, may find the new block missing and then fetch it from other clouds.

Second, a version number uniquely identifies a revision of a data block. When a file is truncated, Kurma does not discard the version number of truncated blocks until the file is completely removed. When the truncated blocks are added back later, their version numbers are incremented from the existing values instead of starting at zero. By not reusing the version number of each block, Kurma ensures that each version of a block has a unique key for its cloud object. Kurma uses the most significant bit of a 64-bit integer to tell whether the block belongs to a file hole, and uses the remaining 63 bits for versioning. The version number space is large enough to last for 292,000 years even if a block is updated one million times a second. Note that gateways are not synchronized and may simultaneously create the same version of a block; Kurma detects this by including the creator `GatewayID` in the per-block metadata.

Third, block versions also help prevent replay attack. Since each block version has a unique cloud object key, attackers could not tell whether two objects are two different versions of one block. Even if attackers managed to replay a block with an old version, Kurma will find that out because the old block would contain the wrong version number in the encrypted cloud object value.

With the help of version numbers, Kurma guarantees that a client always reads fresh data that contains all updates made by clients in the same region. However, Kurma cannot guarantee that a client would always read fresh data that contains all updates made by any clients in other regions. This is because network partitions may separate a Kurma gateway in one region from other gateways, and thus delay the replication of version numbers. This is a trade-off Kurma makes between availability and partition tolerance as per the CAP Theorem [27].

5.3.4.2 Key Management

Each Kurma gateway has a master key pair, which is used for asymmetric encryption (RSA) and consists of a public key (PuK) and a private key (PrK). The public keys are exchanged manually among geo-distributed gateways by security personnel. This is feasible because one geographic office has only one Kurma gateway, and key exchange is only needed when opening an office in a new site. This scheme has the advantages of not relying on any third parties for public key distribution. Knowing the public keys of all other gateways makes it easy for gateways to authenticate each other, and allows Kurma to securely replicate metadata among all gateways.

When creating a file, the creator gateway randomly generates a 128-bit symmetric encryption file key (FK). Then, for each Kurma gateway with which the creator is sharing the file (i.e., an *accessor*), the creator encrypts the FK using the accessor's public key (PuK) using RSA, and then generates a ⟨GatewayID, EFK⟩ pair where EFK is the encrypted FK. All the ⟨GatewayID, EFK⟩ pairs are then stored in the file's metadata (i.e., `keymap` in Figure 5.3). When opening a file, an accessor gateway first finds its ⟨GatewayID, EFK⟩ pair in the `keymap`, and then retrieves the file key by decrypting the EFK using its private key (PrK). Kurma uses the file key to encrypt all blocks of the file. When encrypting a block, Kurma uses the concatenation of the block offset and block version as the initialization vector (IV). This avoids the security flaws of reusing IVs [50].

5.3.5 Multiple Clouds

Most cloud providers have availability higher than 99% [199]. However, cloud outage does happen and can be significant at times [182]. Researchers consider availability as the top obstacle to the growth of cloud computing; the solution is to use multiple cloud providers [10]. By saving data redundantly on multiple clouds, Kurma can achieve high availability and ensure business continuity in the presence of cloud failures. Assuming that 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-nines of availability (99.9999%) when saving replicas on two clouds, each of which has an availability of 99.9%.

Depending on the configuration, Kurma uses one of three redundancy types: (1) replication, (2) erasure coding, and (3) secret sharing (described below). They represent different trade-offs among reliability, security, space overhead, and computational overhead. Each redundancy type also has its own parameters. A file's redundancy type and parameters are determined by Kurma's configuration when creating the file; the parameters are then stored in the file's metadata (i.e., `redundancy` and `cloud_ids` in Figure 5.3).

5.3.5.1 Replication

To tolerate the failure of f clouds using replication, we need to replicate data over $f + 1$ clouds. A write operation (i.e., PUT) finishes only when we have successfully placed a replica in each of the $f + 1$ clouds. A read operation (i.e., GET), however, finishes as soon as one replica is read and found to be valid by checking the MAC and embedded metadata (see Figure 5.4). 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 little extra

computation than using a single replica. Kurma encrypts each data block before replicating it over clouds.

5.3.5.2 Erasure coding

An erasure code transforms a message of $k > 1$ symbols into a longer message with $k + m$ symbols. It can recover the original message with any k symbols of the longer message. Therefore, to tolerate the failure of f clouds using erasure coding, Kurma writes to $k+f$ clouds and read from at least k clouds. In the best case, a read operation 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. Erasure coding is more secure than replication because each cloud sees only part of a block's data. However, even the partial data may leak significant information, so Kurma always encrypts data blocks first before applying erasure coding. Kurma's erasure coding scheme uses Reed-Solomon [151] from the Jerasure library [145].

5.3.5.3 Secret sharing

A secret sharing algorithm has three parameters: (n, k, r), where $n > k > r$. It transforms a *secret* of k symbols into n *shares* (where $n > k$) such that the secret can be recovered from any k shares but it cannot be recovered even partially from any r shares. A secret sharing algorithm, such as Shamir's secret sharing scheme [162] and Rabin's information dispersal algorithm [148], simultaneously provides fault tolerance and confidentiality. It is thus more secure than erasure coding. To tolerate the failure of f clouds using secret sharing, Kurma writes to $k + f$ clouds and read from at least k clouds. Secret sharing also prevents r (or fewer) conspiring clouds from getting any secret from the cloud objects. The storage overhead and read/write amplification of secret sharing are the same as erasure coding. Kurma's secret sharing algorithms include AONT-RS (All-Or-Nothing-Transform-Reed-Solomon) [153] and CAONT-RS [106]. CAONT-RS supports data deduplication on top of AONT-RS.

In sum, replication uses more extra space, but reads from fewer clouds, and does not cost any computation; erasure coding uses less extra space, but reads from more clouds, and costs extra computation; secret sharing use about the same space as erasure coding but is more secure thus requiring more computation. Kurma leaves the choice to end user by supporting all three types of redundancy and uses the method as specified in its configuration file.

5.3.6 File Sharing Across Gateways

To share files across geo-distributed gateways, Kurma gateways use a common set of cloud providers for file data blocks, and maintain a unified file-system namespace by replicating metadata changes across all gateways. In this section, we present Kurma's consistency model and conflict resolution.

5.3.6.1 Consistency Model

Taking advantage of the observation that file sharing is "rarely concurrent" [101], Kurma gateways replicate metadata changes asynchronously to other gateways, and detect and resolve conflicts

after they occur. Asynchronous replication allows Kurma gateways to process metadata operations without waiting for remote gateways. This significantly lowers operation latency. The asynchrony also keeps a Kurma gateway available to its local clients when its secret channels to other gateways are disconnected due to network partitions. In other words, Kurma trades off consistency for performance, availability, and partition tolerance [27].

This degree of trade-off is acceptable because the relaxed consistency is still the same as provided by traditional NFS servers. That is, NFS clients in a local region follow the close-to-open consistency model [103]: when a client opens a file, the client sees all changes made by other clients in the same region who closed the file before the open. The client, however, may not see changes from remote gateways until the changes propagate to the local region. This propagation process is the inconsistency window when a local client may perform operations conflicting with the remote change. Since concurrent file updates are rare [101], conflicts are rare as well. Kurma detects and resolves these conflicts.

Kurma replicates metadata asynchronously using the "all-to-all broadcast" method [14]: once a gateway receives a metadata-mutating request from a client, the gateway processes the request and then broadcasts the metadata change to all other gateways. This broadcasting is feasible because Kurma was designed for a small number of gateways (less than a hundred).

Specifically, Kurma uses Apache Hedwig [60] to replicate file-system metadata across gateways. Hedwig is a publish-subscribe system optimized for communication across geo-distributed data-centers. Hedwig also protects its communications using SSL, so that file-system metadata is securely replicated among the gateways. Hedwig has been used in production by Yahoo! [91]. Using pub-sub services is a common technique to replicate data in geo-distributed systems. Facebook also uses a pub-sub systems called Wormhole to replicate data [163].

We defined the format of Kurma's inter-gateway Hedwig messages using Thrift [61]. A gateway broadcasts a Hedwig message after each successful local metadata-mutating operation (including a write that updates version numbers). Although Hedwig ensures ordered message delivery, Kurma still needs to synchronize dependent operations. For example, Kurma ensures that a preceding directory-creation operation is finished before we process a dependent operation that creates a file in that directory. Kurma does not synchronize independent operations so that they can be processed quickly in parallel.

Since metadata changes are replicated asynchronously, conflicts may arise when geo-distributed gateways perform incompatible changes simultaneously. Kurma needs to detect and resolve the conflicts, and the strategy of resolving conflicts determines Kurma's consistency model. A simple strategy is to prioritize gateways so that metadata changes made by higher-priority gateways always overwrite conflicting changes from lower-priority gateways. With this simple conflict-resolution strategy, geo-distributed Kurma gateways will become eventually consistent: with no additional updates to a given file, all reads from clients in all gateways will eventually return the same file data. This also implies that a gateway separated from other gateways due to network partitions may not see the changes made by other gateways until the partition is healed. Similar to the classic eventual consistency [188], Kurma's eventual consistency assumes all network partitions will eventually recover. Hedwig has also been used in other eventually consistent distributed systems: for instance, Yahoo's Web-scale data-serving system, PNUTS [167], uses Hedwig as well.

If Kurma uses a conflict resolution strategy that allows gateways to diverge, Kurma is no longer eventually consistent: with no additional updates, the gateways may eventually have different states. For example, when two gateways simultaneously create files with the same path, a diverg-

ing conflict-resolution strategy may choose to keep both files by making them visible only to their creator gateway; thus the two gateways will eventually contain different files. With a diverging conflict-resolution strategy, Kurma offers FIFO consistency [109]: Kurma preserves the partial ordering of operations in a single gateway, but not ordering among geo-distributed gateways. This is because Hedwig delivers messages from a particular region in the same order to all subscribers globally, but it may mix messages from different regions in any order [64]. For example, considering a Kurma deployment with three gateways (A, B, and C): if Gateway-A updates a file (say a) from a_{v0} to a_{v1} and then to a_{v2}, and simultaneously Gateway-B updates another file (say b) from b_{v0} to b_{v1} and then to b_{v2}, then Gateway-C will always see the $a_{v0} \rightarrow a_{v1}$ change before $a_{v1} \rightarrow a_{v2}$, and it will also see $b_{v0} \rightarrow b_{v1}$ before $b_{v1} \rightarrow b_{v2}$. However, Gateway-C may see $a_{v0} \rightarrow a_{v1}$ before $b_{v1} \rightarrow b_{v2}$ or in the reverse order.

Because there is no ordering among messages from different gateways, Kurma does not provide causal consistency [4, 72, 110]. Messages with causal dependency may be delivered out of order. For example, Gateway-A updates a file (say a) from a_{v0} to a_{v1}; then the $a_{v0} \rightarrow a_{v1}$ change is quickly replicated to Gateway-B, and after that Gateway-B updates the same file from a_{v1} to a_{v2} before Gateway-C sees the $a_{v0} \rightarrow a_{v1}$ change. In the end, it is therefore possible for Gateway-C to see the $a_{v1} \rightarrow a_{v2}$ change from Gateway-B before it sees the $a_{v0} \rightarrow a_{v1}$ change from Gateway-A.

5.3.6.2 Conflict resolution

Kurma adds extra information inside inter-gateway Hedwig messages to detect conflicts that might happen during the inconsistency window of asynchronous replication. For example, each file-creation message contains not only the parent directory and the file name, but also the `ObjectID` of the locally created file. If a remote gateway happens to create a file with the same name simultaneously, Kurma can differentiate the two files using their `ObjectID`s.

Conflicts are rare [152], but their resolution can be complex requiring application-specific knowledge and even human intervention [96, 161]. Fortunately, the majority of conflicts can be resolved automatically [152]. Kurma has a framework to support conflict detection and resolution. Kurma also contains default conflict resolvers for three common types of conflicts: (1) content conflicts when two or more gateways write to the same file block simultaneously, (2) name conflicts when two or more gateways create objects with the same name in one directory, and (3) existence conflicts when one gateway deletes or moves a file-system object (e.g., delete a directory) while another gateway's operations depend on the object (e.g., create a file in that directory).

A Kurma gateway detects content conflicts when processing a write operation, which may be initiated by a local client or a Hedwig write message replicated from a remote gateway. As shown in Figure 5.3, Kurma maintains a version and a creator `GatewayID` for each block. Overwriting a block with another block of the same creator is not a conflict. Note that Hedwig guarantees in-order delivery of messages, so that block versions with the same creator are written in the proper order. Overwriting a block with a different creator may be a content conflict, and Kurma uses a default resolver that follows Bayou's "last writer wins" policy [181]. Kurma reads two conflicting versions of the block from clouds, compares the timestamps of the two block versions, and uses the version with a later timestamp as the winner.

A gateway detects a name conflict if it finds a different file-system object with the same name already exists when replicating an object from a remote gateway. Note that each object has an `ObjectID` that is unique across all gateways. Our default resolver for name conflicts appends

the creator's name to the file name so that conflicting objects can coexist. To indicate this name change, Kurma sets a flag in the attributes of the newer object. For example, if file "foo" already exists when processing a conflicting Hedwig create message from the remote New York office, a new file "foo_ny" will be created instead. Note that Kurma does not append a suffix to the name of the file created by local clients. In this example, the resolution procedure creates a new file "foo_ny", but keeps the original "foo" intact.

A gateway detects an existence conflict if a dependant condition is false when processing a Hedwig message from a remote gateway. For example, a gateway may find the directory not empty while processing a message that unlinks the directory. Kurma's default policy for this type of conflicts is to mark a deleted object private to the remaining gateways that still need the object. This policy does not delete any objects in question and thus does not lose data. For the directory unlinking example, the gateway processing the unlinking Hedwig message will mark the non-empty directory as private to the local gateway instead of deleting it. Once an object is marked private, subsequent changes to it will not be broadcast to other gateways. A private object is actually removed once its private gateway also deletes it.

More conflict resolution policies can be added: for example, instead of picking the versions of blocks with the latest timestamp, it may be desirable to merge two versions of a file using content-based merging as Git does. We leave these as future work.

5.3.7 Partition over Multiple NFS Servers

To make Kurma gateways robust, Kurma partitioned file system objects (files and directories) of Kurma volumes across multiple NFS servers. Each Kurma gateway maintains a list of the NFS servers running in that gateway, 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 [143]. 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 gateway stores file system metadata 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 metadata backing both NFS servers.

5.3.8 Garbage Collection

When updating a file block, Kurma creates a new cloud object with a higher version number instead of overwriting the existing cloud object. Therefore, Kurma needs a separate garbage-collection process to remove old cloud objects not used by anyone. Each Kurma gateway deletes only the cloud objects it created so that a cloud object is deleted only once. For each gateway that no longer exists (may be removed), Kurma assigns an existing gateway to delete old cloud objects on behalf of the removed gateway. Each Kurma gateway reads its assignment from its configuration file.

Kurma adds a delay before deleting old-version cloud objects to avoid removing old objects that may still be needed by remote gateways. Consider a gateway that updates a block twice in a short time window and generates two versions, say "V-11" and "V-12". The updates generate two Hedwig messages that are broadcast to remote gateways. If a remote gateway processes the Hedwig message of "V-11" and reads "V-11" before processing the next message, the read of "V-11" from clouds may fail if the local gateway has already deleted "V-11" from the clouds. The duration of the delay is a parameter in Kurma's configuration file. A previous study shows that many files are updated in bursts of short time windows, separated by longer periods of inactivity [160]. Therefore, a reasonable delay should be longer than the cross-gateway latency and the active window size.

5.3.9 Persistent Caching

Each Kurma NFS Server (see Figure 5.2) has a persistent cache so that hot data can be read in the low-latency on-premises network instead of from remote clouds. The cache stores plaintext instead of ciphertext so that reading from the cache does not need decryption; this is safe because on-premises machines are trusted in our threat model. The cache is a write-back cache that can hide the high latency of writing to clouds. Being write-back, the cache is persistent because some NFS requests—WRITEs with the stable flag and COMMITs—require dirty data to be flushed to stable storage [164] before replying. The cache also maintains additional metadata in stable storage so that dirty data can be recovered and written back after crashes. The metadata includes a list of dirty files and the dirty extents of each file.

For each cached file, the cache maintains a sparse file of the same size in the server's local file system. Insertion of file blocks is performed by writing to the corresponding block offsets of the sparse files. Evictions are done by punching holes at the corresponding locations using `fallocate` [114]. This design delegates file block management to the local file system, and thus significantly simplifies the Cache Module.

Traditionally, NFS provides close-to-open cache consistency, which guarantees that when a client opens an NFS file, it can observe the changes made by clients that have closed the file before. This requires an NFS client to flush a file's dirty pages upon closing the file, and to revalidate its local cache when re-opening the file (i.e., check if any other client has invalidated the cached data). To be consistent, Kurma's Cache Module also revalidates a file's persistent cache content when processing an NFS open request on the file. As discussed in Chapter 4.3.4.3, Kurma stores a file attribute that is the timestamp of the last change made by any other remote gateway (called `remote_ctime`). The cache compares its locally-saved `remote_ctime` with the latest `remote_ctime`: if they match, it means that no other gateway has changed the file, and the content is still valid; otherwise, the content should be invalidated.

To allow flexible trade-off between consistency and latency, the Cache Module uses a parameter

Components	Language	LoC
Kurma NFS Server	C/C++	15,802
Kurma Gateway Server	Java	27,976
Secret Sharing JNI	C/C++	2,480
RPC & Metadata Definition	Thrift	668

Table 5.1: Lines of code of the Kurma prototype, excluding code generated by Thrift.

called *write-back wait time* ($WBWT$) to control whether the write-back should be performed synchronously or asynchronously upon file close. When $WBWT$ is set to zero, write-back is performed right away and the close request is blocked until the write-back finishes. When $WBWT$ is greater than zero, Kurma first replies to the close request, and then waits $WBWT$ seconds before starting to write-back dirty cache data to the clouds.

5.4 Kurma Implementation

We have implemented a Kurma prototype that includes all features described in the design, except for the partition of volumes among multiple NFS servers. We have tested our prototype thoroughly using unit tests and ensured that it passed all `xfstests` [203] cases applicable to NFS. Table 5.1 shows the lines of code of the prototype. We plan to open-source all of our code in the near future.

5.4.1 NFS Servers

Kurma's NFS Servers (see Figure 5.2) were built on top of NFS-Ganesha [44, 136], a user-space NFS server. NFS-Ganesha can export files from many backends to NFS clients through its *File System Abstraction Layer* (FSAL). FSAL is similar to Linux's Virtual File System (VFS). Multiple FSAL layers can also be stacked to add features in a modular manner. We implemented the Cache Module as an `FSAL_PCACHE` layer and the Gateway Module as an `FSAL_KURMA` layer; we stack `FSAL_PCACHE` on top of `FSAL_KURMA`. `FSAL_PCACHE` always tries to serve NFS requests from the local cache. It only redirect I/Os to the underlying `FSAL_KURMA` in case of cache miss or write back. `FSAL_PCACHE` groups adjacent small I/Os to form large I/Os that are multiples of the file's block size so that slow cloud accesses are amortized. `FSAL_PCACHE` uses the LRU algorithm to evict blocks and ensures that evicted dirty blocks were written back first. `FSAL_KURMA` requests file-system operations to Gateway Servers using a custom protocol defined using Apache Thrift.

5.4.2 Gateway Servers

The Gateway Servers were implemented in Java because many dependent services such as ZooKeeper and Hedwig are also Java libraries. The File-System Module is a Thrift RPC server that communicates with Kurma's NFS Servers; it is implemented using Thrift's Java RPC library. The Metadata Module uses the ZooKeeper client API to store metadata; it also uses Apache Curator [59], a ZooKeeper utility library. Before been stored into ZooKeeper, metadata is compressed using Thrift's compressing data serialization library (`TCompactProtocol`). The Metadata Module uses the Hedwig client API to subscribe to remote metadata changes and to publish local changes.

The secret channels connecting Kurma gateways are SSL socket connections. The Security Module uses Java 8's standard cryptographic library. The Cloud Module includes cloud drivers for Amazon S3, Azure Blob Store, Google Cloud Storage, and Rackspace Cloud Files; it also includes a redundancy layer for replication, erasure coding, and secret sharing. We adapted the cloud drivers code from Hybris [46, 47]. Our erasure coding uses the Jerasure library [145] and its JNI wrapper [183]. Our secret sharing library uses the AONS-RS [153] and CAONS-RS [106] code from CDStore [105]; we also added a JNI wrapper for the secret sharing library.

5.4.3 Optimizations

Our Kurma implementation includes five optimizations:

1. Generating a file's `keymap` needs to encrypt the file's key using slow RSA for each gateway (see Section 5.3.4). To hide the high latency of RSA encryptions, Kurma uses a separate thread to pre-compute a pool of `keymaps`, so that Kurma can quickly take one `keymap` out of the pool when creating a file.

2. To reduce the metadata size written to ZooKeeper, Kurma stores a file's `keymap` in a child `znode` (a ZooKeeper data node) under the file's `znode`. For large files, Kurma also splits their block versions and creators into multiple child `znodes` so that a write only updates one or two small `znodes` of block versions.

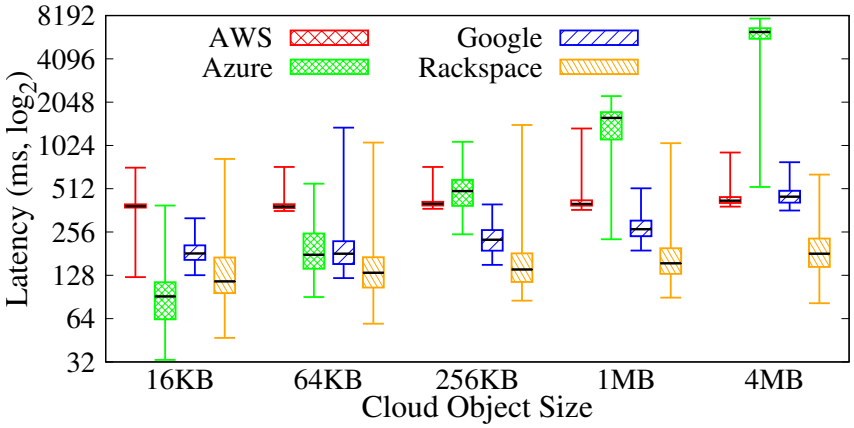
3. Kurma metadata operations are expensive because one ZooKeeper update requires many network hops among the distributed ZooKeeper nodes. Furthermore, a file-system operation may incur multiple ZooKeeper changes. For example, creating a file requires one creation of the file's `znode`, one creation of its `keymap` znode, and one update of its parent directory's `znode`. To amortize high ZooKeeper latency, we batch multiple ZooKeeper changes into a single ZooKeeper transaction [83].

4. Latencies of clouds vary significantly over time. To achieve the best performance, Kurma sorts cloud providers by their latencies every N seconds (N is a configurable parameter) and uses the fastest clouds as backends.

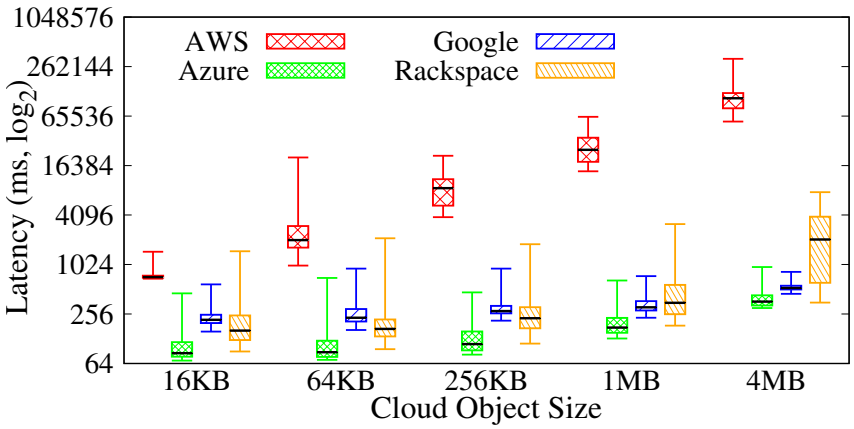
5. To reduce the frequency of accessing the Metadata Servers, Kurma's Gateway Servers cache clean metadata in memory using Guava's `LoadingCache` [86]. The cached metadata includes attributes of hot file-system objects, block versions of opened files, and hot directory entries.

5.5 Kurma Evaluation

We evaluated Kurma's security and performance. To put its performance into perspective, we also compared Kurma to a traditional NFS server.



(a) Read



(b) Write

Figure 5.5: Latency of reading and writing objects from public clouds. The five ticks of each boxplot represent (from bottom to top): the minimum, 25^{th} percentile, median, 75^{th} percentile, and the maximum. Note: both axes are in log_2 scales.

5.5.1 Testbed Setup

Our testbed consists of two identical Dell PowerEdge R710 machines, each with a six-core Intel Xeon X5650 CPU, 64GB of RAM, and an Intel 10GbE NIC. Each machine runs Linux KVM [97] to host a set of identical VMs that represent a cluster of Kurma servers in one gateway. Each VM has two CPU cores and 4GB of RAM. Each VM runs Fedora 25 with a Linux 4.8.10 kernel. To emulate WAN connections among gateways, we injected a network latency of 100ms using `netem` between the two sets of VMs; we chose 100ms because it is the average latency we measured between the US east and west coasts. We measured a network latency of 0.6ms between each pair of servers in the same gateway.

For each gateway, we set three VMs as three Metadata Servers (see Figure 5.2) running ZooKeeper 3.4.9 and Hedwig 4.3.0. Each gateway also has a Kurma NFS Server and a Gateway Server; the two servers communicate using Thrift RPC 0.9.3. The Kurma NFS Server runs NFS-Ganesha 2.3 with our `FSAL_PCACHE` and `FSAL_KURMA` modules; `FSAL_PCACHE` uses an Intel DC S3700 200GB SSD for its persistent cache. The Gateway Server runs on Java 8. Each gateway has another VM

running as an NFSv4.1 client. For comparison, we set up a traditional NFS server on a VM. The traditional NFS server runs NFS-Ganesha with its vanilla `FSAL_VFS` module. `FSAL_VFS` exports to the client an Ext4 file system, stored on a directly-attached Intel DC S3700 200GB SSD. The traditional NFS server does not communicate with other VMs other than the client.

5.5.2 Security Tests

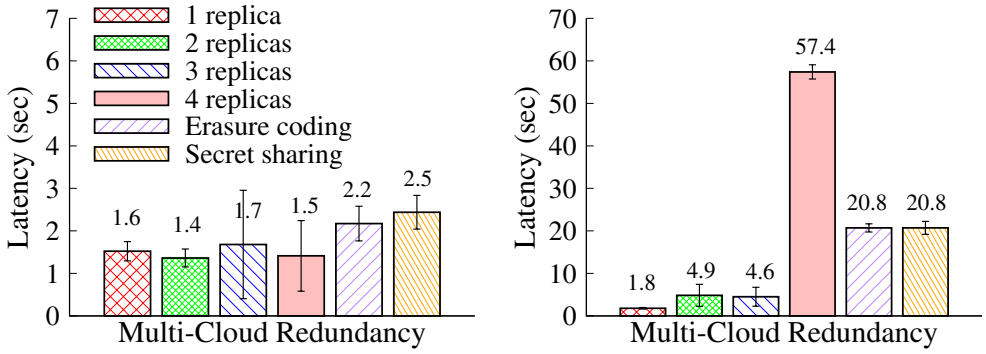
We tested and verified that Kurma can reliably detect security errors and return valid data available in other healthy clouds. To test availability, we manually deleted blocks of a file from one of the clouds, and then tried to read the file from an NFS client. We observed that Kurma first failed to read data from the tampered cloud, but then Kurma retried the read from other clouds, and finally it returned the correct file contents to the client.

For integrity tests, we injected four types of integrity errors by (1) changing one byte of a cloud object, (2) swapping two blocks of the same version at different offsets of a file, (3) swapping two blocks of the same version and offset of two files, and (4) replaying a newer version of a block with an old version. Kurma detected all four types of errors during authentication. It logged information in a local file on the secure gateway for forensic analysis; this information included the block offset and version, the cloud object key, the erroneous cloud, and a timestamp. Kurma also successfully returned the correct content by fetching valid blocks from other untampered clouds. We also tested that Kurma could detect and resolve the three types of conflicting changes made in multiple gateways (see Chapter 5.3.6).

5.5.3 Cloud Latency Tests

Kurma's cloud backends include AWS, Azure, Google, and Rackspace. Figure 5.5 (log_2 scale on both axes) shows the latency of these public clouds when reading and writing objects with different sizes. Kurma favors a large block size because larger blocks cost less (both AWS and Azure charge requests by count instead of size) and reduce the metadata size. Larger block sizes not only reduce cloud costs, but they also improve overall read throughputs. When the block size increased by 256× from 16KB to 4MB, the read latency of a block increased by only 1.1×, 3.1×, 1.2× for AWS, Google, and Rackspace, respectively. However, thanks to the larger block sizes, the read throughput increased by a lot: 234×, 83×, and 216× for AWS, Google, and Rackspace, respectively. However, Azure is an exception where reading a 4MB object takes 6.5 seconds and is 43 times slower than reading a 16KB object. Our measurements of Azure are similar to those reported in Hybris [47], where Azure took around 2 seconds to read an 1MB object, and round 20 seconds to read a 10MB object. Large performance variances of cloud storage in Figure 5.5 were also observed in other studies [47, 202].

A larger block size also improves write throughputs. When the block size increases from 16KB to 4MB, the write throughput increased by 1.9×, 76×, 82×, and 68× for AWS, Azure, Google, and Rackspace, respectively. Writes are significantly slower than reads. As shown in Figure 5.5(b), writing a 4MB object to AWS takes close to 2 minutes. However, the high write latency of large objects is acceptable because Kurma's persistent write-back cache can hide the latency from clients. Therefore, Kurma uses a default block size of 1MB. It can be configured to use larger blocks if occasional high latencies are acceptable during cache misses (e.g., 6.5 seconds for 4MB blocks in Figure 5.5(a)).



(a) Read (b) Write

Figure 5.6: Latency of reading and writing a 16MB file with different redundancy configurations over multiple clouds. The persistent write-back cache is temporarily disabled. The "N replicas" configuration uses the first N clouds out of the list of Google, Rackspace, Azure, and AWS (ordered by their performance with 1MB objects). The erasure coding algorithm is Reed-Solomon with $k = 3$ and $m = 1$. The secret sharing algorithm is CAONS-RS with $n = 4$, $k = 3$, and $r = 2$.

Figure 5.5 also shows that the latencies of different clouds can differ by up to 100 times for objects of the same size. The latency variance is high even for the same cloud. Note the large error bars and the logarithmic scale of the Y-axis. Therefore, when reading from only a subset of clouds, ordering the clouds by their speeds, and using the fastest ones can significantly improve performance. Our tests showed that ordering the clouds by their speeds can cut Kurma's average read latency by up to 54%. If not configured differently, our Kurma prototype reorders the clouds based on their speeds every minute, to decide where to send read requests to first.

5.5.4 Multi-Cloud Tests

For high availability, Kurma stores data redundantly over multiple clouds using replication, erasure coding, and secret sharing. Figure 5.6 shows the latency of reading and writing a 16MB file with different redundancy configurations. To exercise the clouds, we temporarily disabled Kurma's persistent cache in this test. The N-replica configuration uses the first N clouds out of the list of Google, Rackspace, Azure, and AWS. The list is ordered in decreasing overall performance with 1MB objects (see Figure 5.5). Figure 5.6(a) shows that reading a 16MB file takes around 1.6 seconds for all four replication configurations. This is because all N-replica configurations have the same data path for reads: fetching 16 $\times$ 1MB blocks from the single fastest cloud. Note that 1.6 seconds is smaller than 16$\times$ the read latency of a single 1MB-large object (around 0.28s as shown in Figure 5.5). This is because Kurma uses multiple threads to read many blocks in parallel. Both the erasure-coding and secret-sharing configurations need to read from three clouds, and thus the reading takes longer with these two configurations: 2.2s and 2.5s on average, respectively.

When writing a 16MB file, the N-replica setup writes a replica of 16 1MB-large blocks to each of N clouds. The write latency of N-replica is determined by the slowest one of the N clouds. AWS is the slowest cloud for writes, so when AWS was added as a 4^{th} replica to the 3-replica configuration, making it a 4-replica configuration, the write latency jumped to 57.4 seconds (Figure 5.6(b)). Both the erasure-coding and secret-sharing configurations write to four clouds;

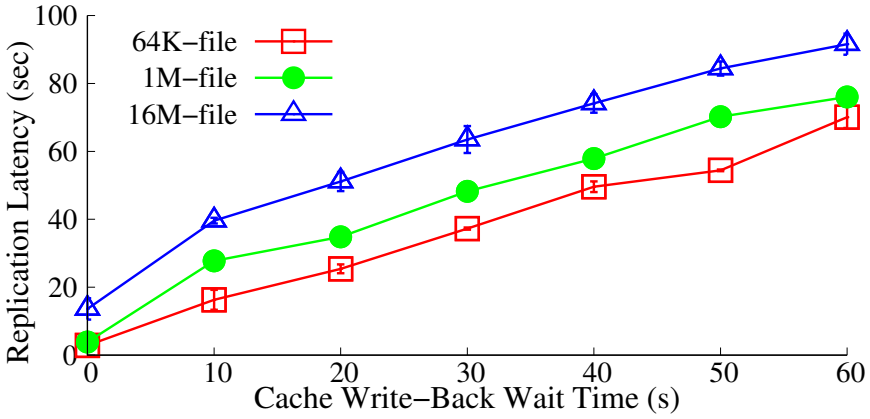


Figure 5.7: Latency of replicating files across geo-distributed gateways under different write-back wait times ($WBWT$s).

however, their write latencies are around one third of the latency of 4-replica. This is because both erasure coding and secret sharing split a 1MB block into four parts, each around 340KB large. Kurma also uses multiple threads to write blocks in parallel, so writing a 16MB-large file takes less time than sequentially writing 16 1MB-large objects.

In Figure 5.6, the 2-replica, erasure-coding, and secret-sharing configurations can all tolerate failure of one cloud. Among them, the 2-replicas configuration has the best performance. However, secret sharing provides extra security—resistance to cloud collusion—and has read performance comparable to the 2-replica configuration. Therefore, we used the secret-sharing configuration in the remaining tests. Note that in general, write latency is less important here because it will be hidden by the persistent write-back cache.

5.5.5 Cross-Gateway Replication

Kurma shares files across geo-distributed gateways by asynchronously replicating file-system metadata. Figure 5.7 shows the replication latency of files under different write-back wait times ($WBWT$s, see Chapter 4.3.6). The timer of a file's replication latency starts ticking after the file is created, written and closed in one gateway; the timer keeps ticking and does not stop until the file is found, opened, fully read and closed in another remote gateway. When $WBWT$ is zero, dirty data is synchronously written back to clouds when closing a file. So the replication latency does not include the time of writing to the clouds and thus is small: 2.9s, 3.9s, and 14s for a 64KB, 1MB, and 16MB files, respectively. When $WBWT$ is not zero, dirty data is written back after closing the file; the replication latency increases linearly with the wait time. In Figure 5.7, the replication latency for larger files is higher because larger files take more time to write to and read from clouds.

5.5.6 Data Operations

To test Kurma's performance with large files, we created a 1GB file: this took around 200 seconds writing to clouds. We then performed random reads on the file after emptying Kurma's persistent cache. Figure 5.8 shows the results. For all three I/O sizes (4KB, 16KB, and 64KB), the initial

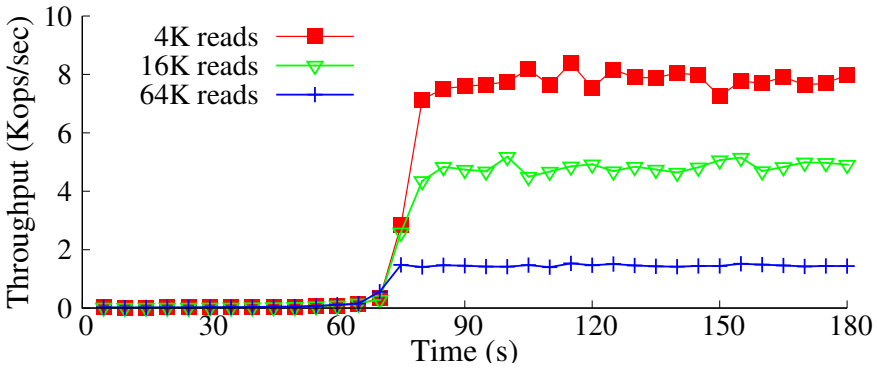


Figure 5.8: Aggregate throughput of randomly reading a 1GB file using 64 threads. The test starts with a cold cache. The I/O sizes are 4KB, 16KB, and 64KB.

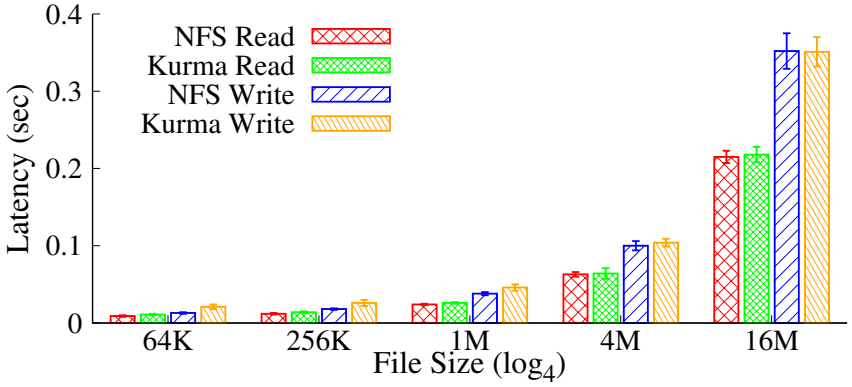


Figure 5.9: Latency of reading and writing files in Kurma and traditional NFS. Kurma's persistent cache is hot during reads; write-back wait time ($WBWT$) is 30 seconds.

throughput was merely around 20 ops/sec because all reads needed to fetch data from clouds over the Internet. The throughput slowly increased as more data blocks were read and cached. Once the whole file was cached, the throughput suddenly jumped high because reading from the cache was faster than reading from the clouds by two orders of magnitude. Afterwards, all reads were served from the cache, and the throughput plateaued. It took around 75 seconds to read the whole file regardless of the I/O size; this is because Kurma always uses the block size (1MB) to read from clouds.

To show Kurma's performance when its cache is in effect, we compared a Kurma gateway with a hot cache to a traditional NFS server. Figure 5.9 shows the latency results of reading and writing whole files. For 64KB files, Kurma's read latency is 22% higher and its write latency is 63% higher. This is because each Kurma metadata operation (e.g., OPEN, CLOSE, and GETATTR) involved multiple servers and took longer to process. In contrast, the traditional NFS server is simpler and each operation was processed by only one server. However, as the file size increased, Kurma's latency became close to that of the traditional NFS. This is because when the file data was cached, Kurma did not need to communicate with the Gateway Server or the Metadata Server; thus its data operations were as fast as NFS, and they amortized the high latency of the few metadata operations.

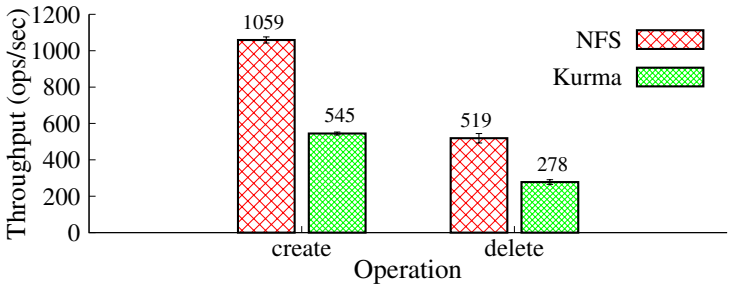


Figure 5.10: Throughput of creating and deleting empty files.

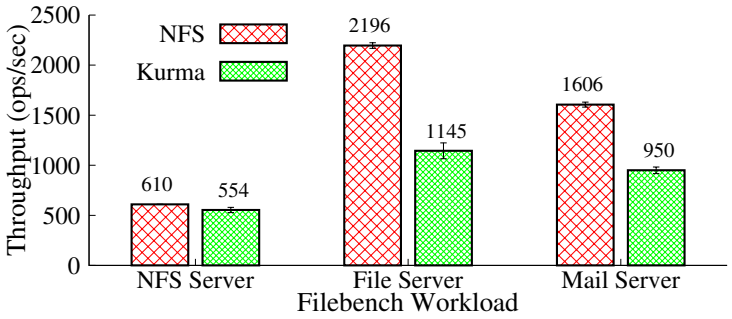


Figure 5.11: Throughput of Filebench workloads.

5.5.7 Metadata Operations

To quantify the performance impact of Kurma's slower metadata operations, we compared Kurma to the traditional NFS in two metadata-only workloads: creating and deleting empty files. Figure 5.10 shows the results. When processing metadata operations, Kurma needs to communicate with the Kurma NFS Server, the Gateway Server, and the Metadata Servers (see Figure 5.2). Moreover, a metadata update in ZooKeeper needs to reach a consensus over all ZooKeeper servers; in our setup with three ZooKeeper nodes, it means that at least three additional network hops. Because of these extra network hops, Kurma's throughput is 49% and 46% lower than NFS for file creation and deletion, respectively. These results were obtained after our optimization of batching ZooKeeper updates (see Chapter 5.4.3); without the optimization, Kurma would be even slower by more than 2×. Kurma's performance penalty in file creation is higher than that in file deletion. This is because creating a Kurma file requires extra computation to generate and encrypt the per-file secret key (see Chapter 5.3.4).

Despite the lower throughput, Kurma has four advantages over a traditional single-server NFS: (1) Kurma is more secure by protecting each file with a secret key; (2) Kurma can share files across geo-distributed gateways; (3) By saving metadata in distributed ZooKeeper, Kurma avoids the single-point failure of NFS's metadata store; and (4) Kurma can easily scale its metadata service by adding more nodes into the ZooKeeper instance.

5.5.8 Filebench Workloads

Figure 5.11 shows Kurma's performance under Filebench workloads that are more complex and realistic than micro-workloads. The Filebench NFS-Server workload emulates the SPEC SFS benchmark [170]. It contains one thread performing four sets of the following operations: (1) open, entirely read, and close three files; (2) read a file, create a file, and delete a file; (3) append to an existing file; and (4) read a file's attributes. The File-Server workload emulates 50 users accessing their home directories and spawns one thread per user to perform operations similar to the NFS-Server workload. The Varmail workload mimics the I/Os of a Unix-style email server operating on a `/var/mail` directory, saving each message as a file; it has 16 threads, each performing create-append-sync, read-append-sync, read, and delete operations on 10,000 16KB files.

For the Filebench NFS-Server workload, Kurma's throughput is around 9% lower than NFS. That is caused by Kurma's slow metadata operations which require extra network hops to process. For example, deleting a file took only 1ms for the traditional NFS server, but around 2ms for Kurma. The File-Server workload has similar operations to the NFS-Server workload, but contains 50 threads instead of one. Many concurrent metadata updates, such as deleting files in a common directory, need to be serialized using locks. This type of serializations makes Kurma's metadata operations even slower because of longer wait time. For example, deleting a file in the File-Server workload took around 16ms for the traditional NFS server, but as long as 188ms for Kurma. Consequently, Kurma's throughput is around 48% lower than the traditional NFS server. The same is true of the multi-threaded Mail-Server workload, where Kurma throughput is around 41% lower. The high latency of metadata operations is the result of trading off performance for security, availability, and scalability. This performance penalty can be minimized by batching operations using NFSv4 compound procedures [34, 158]. Batching metadata operations requires adding journaling into Kurma; otherwise, some metadata updates may be lost during power outage. We left this as future work.

5.6 Related Work of Kurma

Kurma is inspired by many previous studies in related areas, and we have already studied some of those in the discussion of SeMiNAS (Chapter 4.6). 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, and (2) guarantee data or metadata freshness in the face of replay attacks.

5.6.1 A Cloud-of-Clouds

Using multiple cloud providers is an effective way to ensure high availability and business continuity in case of cloud failures [10]. There are several studies of multi-cloud systems [1, 2, 39, 47, 82, 202], and most 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 clouds is more secure than storing whole copies of secrets on a single cloud [6]. For example, DepSky [24], SCFS [1], and CDStore [106] secretly store pieces of sensitive data on

multiple clouds and prevent any single cloud alone from accessing the data.

However, most of these multi-cloud storage systems [2, 23, 24, 47, 82, 106] provide only key-value stores, whereas Kurma provides a geo-distributed file-system service to NAS clients. SCFS [1] and CYRUS [39] use multiple clouds as beck-ends and provide POSIX-like file systems, but they are client-side cloud stores, not cloud storage gateways. Also, both SCFS and CYRUS are for personal use; they favor the scenario when all of a client's files can fit in the client's local storage. In contrast, each Kurma gateway has a large consolidated file data cache shared by many clients.

5.6.2 Freshness Guarantees

Not many cryptographic file systems guarantee data or metadata freshness [173] because replay attacks are difficult to handle. SiRiUS [70] ensures partial metadata freshness but not data freshness. SUNDR [104], SPORC [54], and Depot [113] 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 metadata, most of them [48, 65, 173] use Merkle trees [124] or its variants to detect replay attacks. Iris [173] uses a *balanced Merkle-tree* that supports parallel updates from multiple clients. Athos [71] 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 metadata service running on cloud.

Other cloud systems provide data freshness guarantees for key-value stores, instead of file system services. CloudProof [146] provides a mechanism for clients to verify freshness; Venus [166] and Hybris [47] 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 metadata and block version numbers among geo-distributed cloud gateways. Unlike SCFS [1], Kurma does not rely on any trusted third party for metadata management or key distribution.

5.7 Kurma Conclusions

We presented Kurma, which provides secure file-system services to clients in different regions via geo-distributed cloud gateways. Kurma protects file blocks with authenticated encryption before storing them in clouds. Kurma keeps file-system metadata in trusted gateways instead of in clouds. It also embeds a version number and a timestamp into each file block to ensure data freshness. Kurma tolerates cloud failures by storing data redundantly among multiple clouds using replication, erasure code, or secret sharing. Through secret channels connecting Kurma's gateways, each gateway replicates metadata and version updates to other gateways. The replication is asynchronous so that local operations do not incur long latency of contacting remote gateways. As a trade-off, Kurma detects and resolves conflicting operations that happened simultaneously in multiple gateways. We implemented and evaluated a Kurma prototype. Thanks to Kurma's persistent write-back cache, its performance of data operations is close to a baseline using a single-node NFS

server. Kurma's throughput is around 52–91% of the baseline for general purpose Filebench server workloads. This overhead is contributed by slower metadata operations. Kurma sacrifices some performance for significantly improved security, availability, scalability, and improved file sharing across regions.

Limitations and future work Kurma currently does not consider the insider problem. Currently, Kurma assumes a fixed number of gateways; we leave adding and deleting Kurma gateways to future work. Adding and deleting trusted participants in secure distributed storage systems has several security concerns and has been studied separately elsewhere [11]. Kurma supports resolution of only three common types of conflicts, and we plan to support more. We also plan to amortize the high latency of Kurma's metadata operations among all operations using an NFSv4 compound procedure [34, 158].

Chapter 6

Conclusions

Network-attached storage (NAS) is important, even in this cloud era. Many NAS-based applications demand high performance (especially low latency), which cannot be achieved in clouds due to long physical distances and thus high network latency between clients and clouds. This thesis focuses on improving network-attached storage (NAS) in two important aspects: (1) keep NAS performance in pace with modern high-speed networks and flash SSDs; and (2) integrate NAS systems with cloud storage. Particularly, we benchmarked and optimized the latest NFSv4.1 protocol to make sure excellent performance of high-speed networks and flash SSDs can be successfully delivered to NAS clients through NFSv4.1. We also built cloud storage gateway systems (SeMiNAS and Kurma) that provide NAS clients with seamless, efficient, and secure accesses to public cloud storage.

NFS is the standard storage protocol of network-attached storage (NAS). NFSv4.1 is the latest NFS version, which was not studied much in the literature. We began this thesis by comparing the performance of NFSv4.1 and NFSv3. 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 in 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 used effectively. We also identified NFSv4.1 compound procedures, which were underused, as an opportunity to significantly boost NFSv4.1 performance.

To maximize NFS performance, we then developed vNFS to take full advantage of the underused compound procedures. vNFS is an NFSv4.1-compliant client and library that exposes a vectorized high-level file-system API. We designed and implemented vNFS as a user-space RPC library that supports many bulk operations on files and directories. For example, using only one network round trip, vNFS can open, read, and close many files; set many attributes of many files; or copy many files wholly or partially without moving data over networks. We found it easy to modify applications to use the vectorized API. We evaluated vNFS under a wide range of workloads and network latency conditions, showing that vNFS improves performance even for low-latency networks. On high-latency networks, vNFS can improve performance by as much as two orders of magnitude.

Then, as a first step to develop Kurma, we designed, implemented, and evaluated a simple cloud storage gateway system called SeMiNAS. SeMiNAS explores the idea of using the WAN-friendly NFSv4.1 for communication between gateways and clouds. SeMiNAS achieves end-to-end data

integrity and confidentiality with a highly efficient authenticated-encryption scheme. SeMiNAS leverages advanced NFSv4 features, including compound procedures and data-integrity extensions, to minimize extra network round trips caused by security metadata. SeMiNAS also caches remote files locally to reduce accesses to providers over WANs. We designed, implemented, and evaluated SeMiNAS, which demonstrates a small performance penalty of less than 26% and an occasional performance boost of up to 19% for Filebench workloads. SeMiNAS assumes that cloud providers support an NFS extension [137] that is, however, not standardized yet; SeMiNAS uses only a single cloud as back-end and is susceptible to replay attacks.

Finally, based on our 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 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 gateways. For high security, Kurma keeps file-system metadata on-premises and encrypts data blocks before writing them to clouds. For higher confidentiality and availability, Kurma divides data across multiple clouds using erasure coding or secret sharing. To share files among distant clients, Kurma maintains a unified file-system namespace across geo-distributed gateways. Kurma is also faster and more realistic than SeMiNAS by trading off cross-region consistency for performance and using real clouds (we used Amazon AWS, Google Cloud Storage, Microsoft Azure, and Rackspace). Evaluations of our Kurma prototype showed that its performance is around 52–91% that of a local NFS server while providing geo-replication, confidentiality, integrity, and high availability.

6.1 Limitations and Future Work

The future work of this thesis includes the following:

1. Transactional NFS compounds. vNFS can be extended with transactional execution, which greatly simplifies error handling. Currently, compounded operations are processed sequentially by NFS servers (according to the standard [158]) and the first failed operation halts the execution of remaining operations in the compound. If a compound is executed atomically, we can then process independent operations inside in parallel. This parallel execution can reduce the compound processing time by taking advantage of the high parallelism inside modern SSDs. Therefore, transactional compounds may simultaneously simplify programming while boosting performance. We have explored this idea, but the main difficulty we faced is the lack of a transactional local file-system.

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 gateway of a file (e.g., the creator gateway), global consistency can be achieved by synchronizing changes to the file in all gateways with the master. This idea is somewhat similar to NFSv4.1 delegations.

3. Cost awareness and optimization. Cloud operations have different cost and performance. For example, reading data is generally more expensive than writing [1], and one cloud provider

may be slower but cheaper than other cloud providers. Lowering Kurma's cost under a reasonable performance is an interesting topic worth exploring.

4. Kurma load balancing. Kurma supports only volume-level load balancing which may be too coarse; finer load balance may be achieved by using NFSv4.1 referrals and pNFS. Also, Kurma currently sorts cloud providers and always uses the fastest one. A better method would be to load-balance the reads across several clouds, proportionally to their performance. This makes use of the bandwidths of slower clouds, and maximizes the overall throughput of Kurma.

6.2 Lessons Learned and Long-Term Visions

There are several important lessons I learned during this Ph.D. work:

- Good research problems are often buried in the details. We all know that finding a good problem to solve is the single most important step in research. Many important problems in this thesis caught our attention after in-depth analysis of details, such as performance anomalies. For example, when benchmarking NFSv4.1, we found different NFS clients had uneven throughput; we dove deep into the analysis and found the HashCast problem (see Chapter 2.3.2). In another example, we were puzzled that each NFSv4.1 compound contained only two to four operations. After analyzing this in detail, we realized that compounds are limited by the POSIX API and this problem led to the vNFS paper [34]. This compound problem also contributed to the optimization of our SeMiNAS system (see Chapter 4.3.5.2).

- Testing is our friend. Modern storage systems often contain many components and depend on many external libraries. We need to ensure that each component and library is working as expected; for that, we need a unit test for each function of every component. Tests should not only provide full coverage but also be thorough. During our initial benchmarking of SeMiNAS, we experienced frequent kernel crashes due to bugs in the DIX (see Chapter 4.3.5.1) implementation. It took us more than three months to realize and fix these kernel bugs. Our initial test of DIX did not consider the scenario of big files and thus failed to expose the bugs at an earlier stage. Integrating these components and libraries is also challenging, so we also needed integration tests and system tests. Tests are time-consuming in the short term, but they did save us a lot time in later development of our systems. Tests also helped us in assigning tasks to teammates; I often used tests to check their implementations.

- Keep it simple, stupid (KISS). KISS is a well-known design principle. Although I was aware of this principle, I neglected to follow it when I was tempted by some fancy features. One example is that we had anti-virus in our initial SeMiNAS. However, anti-virus makes the threat model, design, and implementation much more complex. For instance, most anti-virus systems requires scanning files in whole, which is very different from encryption that can be done on a per-block basis. We should have started the project without anti-virus, and only added it after the system has matured. In fact, anti-virus is complex and it alone deserves a long paper as my laboratory did for Avfs [129]. I also failed to follow the KISS principle when designing Kurma. In the initial design, the file block's size was lazily determined upon

```
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();
```

Figure 6.1: A C++ code sample of building a compound of multiple file-system operations.

the first write so that we could use large block sizes for big files. However, it led to a lot of bugs when we had an empty file with undetermined block size. A safer and simpler approach was to make the file block size configurable, which we did later.

I also would like to share some long-term visions of systems we built in this thesis:

- Transactional NFS compounds using transactional storage devices. High-performance SSDs have not only high internal parallelism but also transaction support thanks to their log-structured data management. If we can leverage storage devices' transaction support, we can then more efficiently and easily achieve transactional NFS compounds.
- Transactional NFS compounds API in general languages. We proposed a customized client file-system library to initiate large NFS in Chapter 3. However, it would be more convenient and elegant to support that in general programming languages. Some modern languages, such as the proposed C++17 [125], are showing the potential to do that. Figure 6.1 shows such an example that may initiate a compound comprising three file operations (i.e., open, read, and close).
- Make Kurma ready for production use. We will make Kurma's code public available. Kurma is currently a research prototype, but most of its features are already mature. Kurma can effectively alleviate the security concerns of using public clouds, and we would like it to be actually used, at least by fellow researchers. I hope consistent development from the open-source community will make Kurma ready for production in the future.

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] Mustaque Ahamad, Gil Neiger, James E Burns, Prince Kohli, and Phillip W Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[5] 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.

[6] 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.

[7] E. Anderson. Capture, conversion, and analysis of an intense NFS workload. In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST '09)*, pages 139–152, San Francisco, CA, February 2009. USENIX Association.

[8] Antoine Potten. Ant renamer, 2016. `http://www.antp.be/software/renamer`.

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

[10] 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.

[11] Giuseppe Ateniese, Kevin Fu, Matthew Green, and Susan Hohenberger. Improved proxy re-encryption schemes with applications to secure distributed storage. *ACM Trans. Inf. Syst. Secur.*, 9(1):1–30, February 2006.

[12] J. Axboe. CFQ IO scheduler, 2007. `http://mirror.linux.org.au/pub/linux.conf.au/2007/video/talks/123.ogg`.

[13] 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.

[14] Peter Bailis and Ali Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Communications of the ACM*, 56(5):55–63, 2013.

[15] 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.

[16] 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.

[17] A. Batsakis, R. Burns, A. Kanevsky, J. Lentini, and T. Talpey. CA-NFS: A congestion-aware network file system. *ACM Transaction on Storage*, 5(4), 2009.

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

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

[20] 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.

[21] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, Internet Engineering Task Force, May 2015.

[22] John Bent, Douglas Thain, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Explicit control in the batch-aware distributed file system. In *NSDI*, pages 365–378, 2004.

[23] David Bermbach, Markus Klems, Stefan Tai, and Michael Menzel. Metastorage: A federated cloud storage system to manage consistency-latency tradeoffs. In *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing*, CLOUD '11, pages 452–459, Washington, DC, USA, 2011. IEEE Computer Society.

[24] 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.

[25] Dhruba Borthakur et al. Hdfs architecture guide. *Hadoop Apache Project*, 53, 2008.

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

[27] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[28] C. Brooks. Cloud storage often results in data loss. `http://www.businessnewsdaily.com/1543-cloud-data-storage-problems.html`, October 2011.

[29] Neil Brown. Overlay filesystem, 2015.

[30] S. Byrne. Microsoft onedrive for business modifies files as it syncs. `http://www.myce.com/news/microsoft-onedrive-for-businessmodifies-files-as-it-syncs-71168`, April 2014.

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

[32] 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.

[33] 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.

[34] M. Chen, D. Hildebrand, H. Nelson, J. Saluja, A. Subramony, and E. Zadok. vNFS: Maximizing NFS performance with compounds and vectorized I/O. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, pages 301–314, Santa Clara, CA, February/March 2017. USENIX Association. Nominated for best paper award.

[35] M. Chen, A. Vasudevan, K. Wang, and E. Zadok. SeMiNAS: A secure middleware for wide-area network-attached storage. In *Proceedings of the 9th ACM International Systems and Storage Conference (ACM SYSTOR '16)*, Haifa, Israel, June 2016. ACM.

[36] 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`.

[37] 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.

[38] Stuart Cheshire. TCP performance problems caused by interaction between Nagle's algorithm and delayed ACK, May 2005.

[39] Jae Yoon Chung, Carlee Joe-Wong, Sangtae Ha, James Won-Ki Hong, and Mung Chiang. Cyrus: Towards client-defined cloud storage. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 17:1–17:16, New York, NY, USA, 2015. ACM.

[40] J. Curn. How a bug in dropbox permanently deleted my 8000 photos. `http://paranoia.dubfire.net/2011/04/how-dropbox-sacrificesuser-privacy-for.html`, 2014.

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

[42] 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.

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

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

[45] I/O Controller Data Integrity Extensions, 2016. `https://oss.oracle.com/~mkp/docs/dix.pdf`.

[46] Dan Dobre, Paolo Viotti, and Marko Vukolić. Hybris: robust and strongly consistent hybrid cloud storage, 2014. `https://github.com/pviotti/hybris`.

[47] 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.

[48] 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.

[49] D. Duchamp. Optimistic lookup of whole NFS paths in a single operation. In *Proceedings of the Summer 1994 USENIX Technical Conference*, pages 143–170, Boston, MA, June 1994.

[50] M. Dworkin. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. National Institute of Standards and Technology (NIST), November 2007.

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

[52] 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.

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

[54] 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.

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

[56] Filebench, 2016. `https://github.com/filebench/filebench/wiki`.

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

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

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

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

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

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

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

[64] The Apache Software Foundation. Hedwig Design, 2017.

[65] 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.

[66] K. Fu, M. F. Kaashoek, and D. Mazi'eres. Fast and secure distributed read-only file system. *Computer Systems*, 20(1):1–24, 2002.

[67] 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.

[68] S. Ghemawat, H. Gobioff, and S. T. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, Bolton Landing, NY, October 2003. ACM SIGOPS.

[69] 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.

[70] 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).

[71] 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.

[72] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 371–384, New York, NY, USA, 2016. ACM.

[73] 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.

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

[75] S. Han, S. Marshall, B. Chun, and S. Ratnasamy. MegaPipe: A new programming interface for scalable network I/O. In *The 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012.

[76] Red Hat. What is DIF/DIX (also known as PI)? `https://access.redhat.com/solutions/41548`, December 2015.

[77] T. Haynes. NFS version 4 minor version 2 protocol. RFC draft, Network Working Group, September 2015. `https://tools.ietf.org/html/draft-ietf-nfsv4-minorversion2-39`.

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

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

[80] D. Howells. FS-Cache: A Network Filesystem Caching Facility. In *Proceedings of the 2006 Linux Symposium*, volume 2, pages 427–440, Ottawa, Canada, July 2006. Linux Symposium.

[81] HTTP Archive. URL statistics, September 2016. `http://httparchive.org/trends.php`.

[82] 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.

[83] 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.

[84] G. Huntley. Dropbox confirms that a bug within selective sync may have caused data loss. `https://news.ycombinator.com/item?id=8440985`, October 2014.

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

[86] Google. Inc. Guava: Google core libraries for Java 6+, 2017. `https://github.com/google/guava`.

[87] J. Nagle. Congestion control in IP/TCP internetworks. RFC 896, Network Working Group, January 1984.

[88] Jason Fitzpatrick. Bulk rename utility, 2016. `http://www.bulkrenameutility.co.uk/Main_Intro.php`.

[89] 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.

[90] 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.

[91] Flavio Junqueira. Apache BookKeeper PoweredBy, 2017.

[92] Chet Juszczak. Improving the write performance of an NFS server. In *Proceedings of the USENIX Winter 1994 Technical Conference*, WTEC'94, San Francisco, California, 1994. USENIX Association.

[93] 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.

[94] Kevin Fu. Group sharing and random access in cryptographic storage file systems. Master's thesis, MIT, 1999.

[95] Kim Jensen. AdvancedRenamer, 2016. `https://www.advancedrenamer.com/`.

[96] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 213–225, Asilomar Conference Center, Pacific Grove, CA, October 1991. ACM Press.

[97] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the 2007 Ottawa Linux Symposium (OLS 2007)*, volume 1, pages 225–230, Ottawa, Canada, June 2007.

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

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

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

[101] 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.

[102] 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.

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

[104] 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.

[105] M. Li, C. Qin, and P. Lee. Convergent dispersal deduplication datastore, 2016. `https://github.com/chintran27/CDStore`.

[106] Mingqiang Li, Chuan Qin, and Patrick P. C. Lee. Cdstore: Toward reliable, secure, and cost-efficient cloud storage via convergent dispersal. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 111–124, Berkeley, CA, USA, 2015. USENIX Association.

[107] Z. Li, R. Grosu, P. Sehgal, S. A. Smolka, S. D. Stoller, and E. Zadok. On the Energy Consumption and Performance of Systems Software. In *Proceedings of the 4th Israeli Experimental Systems Conference (ACM SYSTOR '11)*, Haifa, Israel, May/June 2011. ACM.

[108] Linux Programmer's Manual. *lio_listio*, September 2016. `http://man7.org/linux/man-pages/man3/lio_listio.3.html`.

[109] R. J. Lipton and J. S. Sandberg. Pram: A scalable shared memory. Technical Report TR-180-88, Princeton University, Princeton, NY, 1988.

[110] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[111] 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.

[112] 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.

[113] 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.

[114] Linux man pages. fallocate(2) - manipulate file space. `https://linux.die.net/man/2/fallocate`.

[115] Linux man pages. ftw(3) - file tree walk. `http://linux.die.net/man/3/ftw`.

[116] Linux man pages. open(2) - open and possibly create a file or device. `http://linux.die.net/man/2/open`.

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

[118] R. Martin and D. Culler. NFS sensitivity to high performance networks. In *Proceedings of SIGMETRICS*. ACM, 1999.

[119] 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.

[120] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 124–139, Charleston, SC, December 1999. ACM.

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

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

[123] 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.

[124] 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.

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

[126] E. Mill. Dropbox bug can permanently lose your files. `https://konklone.com/post/dropboxbug-can-permanently-lose-your-files`, October 2012.

[127] 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.

[128] D. L. Mills. Internet Time Synchronization: the Network Time Protocol. Technical Report RFC 1129, Network Working Group, October 1989.

[129] 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.

[130] CNN Money. Hospital network hacked, 2014. `http://goo.gl/wSfzAx`.

[131] Martin Mulazzani, Sebastian Schrittwieser, Manuel Leithner, Markus Huber, and Edgar Weippl. Dark clouds on the horizon: Using cloud storage as attack vector and online slack space. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 5–5, Berkeley, CA, USA, 2011. USENIX Association.

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

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

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

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

[136] NFS-Ganesha, 2016. `http://nfs-ganesha.github.io/`.

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

[138] nghttp2. nghttp2: HTTP/2 C library, September 2016. `http://nghttp2.org`.

[139] M.F.X.J. Oberhumer. lzop data compression utility. `www.lzop.org/`.

[140] Arun Olappamanna Vasudevan. Support stacking multiple FSALs, 2014. `http://sourceforge.net/p/nfs-ganesha/mailman/message/32999686/`.

[141] 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.

[142] 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.

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

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

[145] James S Plank, Scott Simmerman, and Catherine D Schuman. Jerasure: A library in c/c++ facilitating erasure coding for storage applications-version 1.2, 2008. http://git-scm.com.

[146] 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.

[147] 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.

[148] Michael O Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM (JACM)*, 36(2):335–348, 1989.

[149] 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.

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

[151] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[152] 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.

[153] Jason K. Resch and James S. Plank. Aont-rs: Blending security and performance in dispersed storage systems. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 14–14, Berkeley, CA, USA, 2011. USENIX Association.

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

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

[156] 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.

[157] Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout. It's time for low latency. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, 2011.

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

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

[160] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 110–123, Charleston, SC, December 1999. ACM.

[161] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39:447–459, 1990.

[162] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[163] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary, Laurent Demailly, Thomas Fersch, Liat Atsmon Guz, Andrzej Kotulski, Sachin Kulkarni, Sanjeev Kumar, Harry Li, Jun Li, Evgeniy Makeev, Kowshik Prakasam, Robbert Van Renesse, Sabyasachi Roy, Pratyush Seth, Yee Jiun Song, Benjamin Wester, Kaushik Veeraraghavan, and Peter Xie. Wormhole: Reliable pub-sub to support geo-replicated internet services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 351–366, Oakland, CA, 2015. USENIX Association.

[164] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS version 4 protocol. RFC 3530, Network Working Group, April 2003.

[165] S. Shepler, M. Eisler, and D. Noveck. NFS version 4 minor version 1. Technical Report RFC 5661, Network Working Group, January 2010.

[166] 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.

[167] A. Silberstein, J. Chen, D. Lomax, B. McMillan, M. Mortazavi, P. P. S. Narayan, R. Ramakrishnan, and R. Sears. Pnuts in flight: Web-scale data serving at yahoo. *IEEE Internet Computing*, 16(1):13–23, Jan 2012.

[168] 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.

[169] SoftNAS. SoftNAS Cloud. `https://www.softnas.com/wp/`, 2016.

[170] SPEC. SPEC SFS97_R1 V3.0. `www.spec.org/sfs97r1`, September 2001.

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

[172] David C. Steere. Exploiting the non-determinism and asynchrony of set iterators to reduce aggregate file I/O latency. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, SOSP '97, 1997.

[173] 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.

[174] 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.

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

[176] 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.

[177] V. Tarasov, A. Gupta, K. Sourav, S. Trehan, and E. Zadok. Terra incognita: On the practicality of user-space file systems. In *HotStorage '15: Proceedings of the 7th USENIX Workshop on Hot Topics in Storage*, Santa Clara, CA, July 2015. USENIX.

[178] V. Tarasov, E. Zadok, and S. Shepler. Filebench: A flexible framework for file system benchmarking. *;login: The USENIX Magazine*, 41(1):6–12, March 2016.

[179] 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.

[180] 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`.

[181] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM.

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

[183] Jos van der Til. Jerasure library that adds Java Native Interface (JNI) wrappers, 2014. `https://github.com/jvandertil/Jerasure`.

[184] Vijay Vasudevan, David G. Andersen, and Michael Kaminsky. The case for VOS: The vector operating system. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 31–31, Berkeley, CA, USA, 2011. USENIX Association.

[185] 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.

[186] M. Vilayannur, S. Lang, R. Ross, R. Klundt, and L. Ward. Extending the POSIX I/O interface: A parallel file system perspective. Technical Report ANL/MCS-TM-302, Argonne National Laboratory, October 2008.

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

[188] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40, jan 2009.

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

[190] M. Mitchell Waldrop. The chips are down for Moore's law. *Nature*, 530(7589):144–147, 2016.

[191] Werner Beroux. Rename-It!, 2016. `https://github.com/wernight/renameit`.

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

[193] Wikipedia. Gamma Distribution. `http://en.wikipedia.org/wiki/Gamma_distribution`.

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

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

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

[197] 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.

[198] Filebench Workload Model Language (WML), 2016. `https://github.com/filebench/filebench/wiki/Workload-Model-Language`.

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

[200] 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.

[201] 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.

[202] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 292–308, New York, NY, USA, 2013. ACM.

[203] SGI XFS. xfstests, 2016. `http://xfs.org/index.php/Getting_the_latest_source_code`.

[204] 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.

[205] Zadara Storage. Virtual Private Storage Array. `https://www.zadarastorage.com/`, 2016.

[206] 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.