Is NFSv4.1 Ready for Prime Time?

MING CHEN, DEAN HILDEBRAND, GEOFF KUENNING, SOUJANYA SHANKARANARAYANA, BHARAT SINGH, AND EREZ ZADOK

Ming Chen is a PhD candidate in computer science at Stony Brook University. His research interests include file systems, storage systems, and cloud computing. He is working together with Professor Erez Zadok in building storage systems using NFS and cloud services. mchen@cs.stonybrook.edu

Dean Hildebrand manages the Cloud Storage Software team at the IBM Almaden Research Center and is a recognized expert in the field of distributed and parallel file systems. He has authored numerous scientific publications, created over 28 patents, and chaired and sat on the program committee of numerous conferences. Hildebrand pioneered pNFS, demonstrating the feasibility of providing standard and scalable access to any file system. He received a BSc in computer science from the University of British Columbia in 1998 and a PhD in computer science from the University of Michigan in 2007. dhildeb@us.ibm.com

Geoff Kuenning spent 15 years working as a programmer before changing directions and joining academia. Today he teaches computer science at Harvey Mudd College in Claremont, CA, where he has developed a reputation for insisting that students write readable software, not just working code. When not teaching or improving his own programs, he can often be found trying—and usually failing—to conquer nearby Mount Baldy on a bicycle. geoff@cs.hmc.edu.

NFSv4.1, the latest version of the NFS protocol, has improvements in correctness, security, maintainability, scalability, and cross-OS interoperability. To help system administrators decide whether or not to adopt NFSv4.1 in production systems, we conducted a comprehensive performance comparison between NFSv3 and NFSv4.1 on Linux. We found NFSv4.1 to be stable and its performance to be comparable to NFSv3. Our new (and sometimes surprising) observations and analysis cast light on the dark alleys of NFS performance on Linux.

The Network File System (NFS) is an IETF-standardized protocol to provide transparent file-based access to data on remote hosts. NFS is a highly popular network-storage solution, and it is widely supported in OSes, including FreeBSD, Linux, Solaris, and Windows. NFS deployments continue to increase thanks to faster networks and the proliferation of virtualization. NFS is also playing a major role in today's cloud era by hosting VM disk images in public clouds and providing file services in cloud storage gateways.

The continuous evolution of NFS has contributed to its success. The first published version, NFSv2, operates on UDP and limits file sizes to 2 GB. NFSv3 added TCP support, 64-bit file sizes and offsets, and performance enhancements such as asynchronous writes and `READDIRPLUS`.

The latest version, NFSv4.1, includes additional items such as (1) easier deployment with one single well-known port (2049) that handles all operations, including locking, quota management, and mounting; (2) operation coalescing via `COMPOUND` procedures; (3) stronger security using RPCSEC_GSS; (4) correct handling of retransmitted non-idempotent operations with *sessions* and Exactly-Once Semantics (EOS); (5) advanced client-side caching using *delegations*; and (6) better scalability and more parallelism with pNFS [3].

We investigated NFSv4.1's performance to help people decide whether to take advantage of its improvements and new features in production. For that, we thoroughly evaluated Linux's NFSv4.1 implementation by comparing it to NFSv3, the still-popular older version [5], in a wide range of environments using representative workloads.

Our study has four major contributions:

- a demonstration that NFSv4.1 can reach up to 1177 MB/s throughput in 10 GbE networks with 0.2–40 ms latency while ensuring fairness to multiple clients;
- a comprehensive performance comparison of NFSv3 and NFSv4.1 in low- and high-latency networks, using a wide variety of micro- and macro-workloads;
- a deep analysis of the performance effect of NFSv4.1's unique features (statefulness, sessions, delegations, etc.); and
- fixes to Linux's NFSv4.1 implementation that improve its performance by up to 11χ.

Bharat Singh is a graduate student in computer science at Stony Brook University. Before that he worked at NetApp developing storage solutions. His research interests include file and storage systems. He is working with Professor Erez Zadok in building high performance storage systems using NFS. bharat.singh.1@stonybrook.edu

Soujanya Shankaranarayana earned her master's degree from Stony Brook University in 2014 and was advised by Professor Erez Zadok. She is presently working at Tintri Inc., a Bay Area storage startup. Her research interests involve NFS, distributed storage, and storage solutions for virtualized environments. In addition to academic research experience, she also has industry experience in distributed systems and storage systems. soshankarana@cs.stonybrook.edu

Erez Zadok received a PhD in computer science from Columbia University in 2001. He directs the File Systems and Storage Lab (FSL) at the Computer Science Department at Stony Brook University, where he joined as faculty in 2001. His current research interests include file systems and storage, operating systems, energy efficiency, performance and benchmarking, security, and networking. He received the SUNY Chancellor's Award for Excellence in Teaching, the US National Science Foundation (NSF) CAREER Award, two NetApp Faculty awards, and two IBM Faculty awards. ezk@fsl.cs.sunysb.edu

Methodology

Hardware Configuration	
Server & Clients	Dell PowerEdge R710 (1 server, 5 clients)
CPU	Six-core Intel Xeon X5650, 2.66 GHz
Memory	64 GB
NIC	Intel 82599 10 GbE
Server Disk	RAID-0
RAID Controller	Dell PERC 6/i
Disks	Eight Intel DC S2700 (200 GB) SSDs
Read Throughput	860 MB/s
Network Switch	Dell PowerConnect 8024F
Jumbo Frames	Enabled
MTU	9000 bytes
TCP Segmentation Offload	Enabled
Round-Trip Time	0.2 ms (`ping`)
TCP Throughput	9.88 Gb/s (`iperf`)
Software Settings	
Linux Distribution	CentOS 7.0.1406
Kernel Version	3.14.17
Server File System	`ext4`
Network Settings	
NFS Implementation	Linux in-kernel
NFS Server Export Options	Default (sync set)
NFSD Threads	32
`tcp_slot_table_entries`	128
NFS Client Settings	
`rsize` & `wsize`	1MB
`actimeo`	60
NFS Security Settings	Default (no `RPCSEC_GSS`)

Table 1: Details of the experimental setup

Our experimental testbed consists of six identical machines (one server and five clients). Table 1 gives details of the hardware and software configuration.

We developed *Benchmaster*, a benchmarking framework that launches workloads on multiple clients concurrently. Benchmaster also collects system statistics using tools such as `iostat` and `vmstat`, and by reading `procfs` entries. `/proc/self/mountstats` provides particularly useful per-procedure details, including request and byte counts, accumulated queueing time, and accumulated round-trip time.

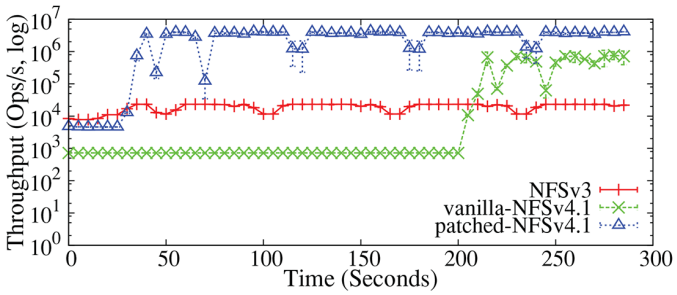

Figure 1: Aggregate throughput of five clients reading 10,000 4 KB files with 16 threads in a 10 ms-delay network ($\log_{10}$)

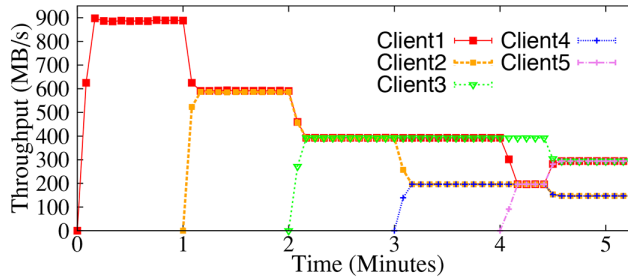

Figure 2: Sequential-read throughputs of individual clients when they were launched one after the other at an interval of one minute (results of one run of experiments)

We ran our tests long enough to ensure stable results, usually five minutes. We repeated each test at least three times, and computed the 95% confidence interval for the mean. Unless otherwise noted, we plot the mean of all runs' results, with the half-widths of the confidence intervals shown as error bars. To evaluate NFS performance over short- and long-distance networks, we injected delays ranging from 1 ms to 40 ms (Internet latency within New York State) using `netem` at the client side. For brevity, we call the network without extra delay "zero-delay," and the network with nms injected delay as "*n* ms-delay."

Major Results

We organize our major results around several questions that might arise when considering whether to adopt NFSv4.1. We hope our answers will help system administrators make well-informed deployment decisions.

Is NFSv4.1 Ready for Production?

Throughout our study, Linux's NFSv4.1 implementation functioned well and finished all workloads smoothly. This was true for three different kernel versions: 2.6.32, 3.12.0, and 3.14.17 (only the 3.14.17 results are shown here). We believe NFSv4.1's implementation to be stable, considering the wide scope and high intensity of our tests, which included dozens of micro- and macro-workloads, involved as many as 2560 threads, and reached throughput up to 1177 MB/s.

For data-intensive workloads that operate on one big NFS file, NFSv4.1 performed almost the same as the simpler NFSv3 while providing extra features. Both NFSv3 and NFSv4.1 were able to easily saturate the 10 GbE network, although we needed to enlarge the maximum TCP buffer sizes (i.e., `rmem_max` and `wmem_max`) when the network latency was long.

For metadata-intensive workloads that operate on many small files and directories, our early results showed vast differences between NFSv3 and NFSv4.1. Figure 1 shows one such result, where NFSv4.1 (the bottom curve at time 0) performed 24χ worse (note the $\log_{10}$ scale) than NFSv3 (the bottom curve at time 300) during the first 200 seconds, but jumped to be 25χ better after that. By looking at `mountstats` data and tracing the kernel with SystemTap, we found that NFSv4.1's poor early performance was due to a system bug. We fixed that with a patch [6], resulting in the upper curve (marked with triangles) in Figure 1. The performance jump at about 40 seconds was because of a new feature called *delegations*, on which we will elaborate later. For the rest of the article, we will report results of the patched NFSv4.1 instead of the vanilla version.

After the bug fix, NFSv4.1 performed close (slightly better or worse) to NFSv3 for most metadata-intensive workloads, but with exceptions in extreme settings. For example, with 512 threads per client, NFSv4.1 created small files 2.9χ *faster* or 3χ *slower* depending on the network latency.

Generally speaking, we think NFSv4.1 is almost ready for production deployment. It functioned well in our comprehensive and intense tests, although with a few performance issues in the metadata-intensive tests. However, the performance bug we found was easy to uncover, and its fix was also straightforward, suggesting that NFSv4.1 is not yet widely deployed; otherwise, these issues would already have been corrected. We argue that it is time for NFS users to at least start testing NFSv4.1 for production workloads.

Are You Affected by Hash-Cast?

Hash-Cast is a networking problem we discovered during our study; it affects not only NFS but any systems hosting concurrent data-intensive TCP flows. In our test of a sequential read workload, we frequently observed a *winner-loser pattern* among the clients, for both NFSv3 and NFSv4.1, exhibiting the following three traits: (1) the clients formed two clusters, one with high throughput (winners) and one 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; a winner in one experiment might became a loser in another.

Initially, we suspected that the winner-loser pattern was caused by the order in which the clients launched the workload. To test that hypothesis, we repeated the experiment but launched the

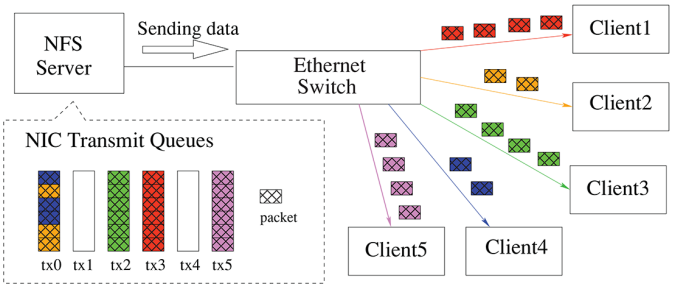

Figure 3: Illustration of Hash-Cast

clients in a controlled order, one additional client every minute. However, the results disproved any correlation between experiment launch order and winners. Figure 2 shows that Client2 started second but ended up as a loser, whereas Client5 started last but became a winner. Figure 2 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 caused by the server's use of its multi-queue 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 [7]. Many modern NICs have multiple sets of `tx-queues` and `rx-queues` to allow networking to scale with the number of CPU cores, and to facilitate better NIC virtualization [7]. 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 outstanding 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.

The winner-loser pattern is caused by uneven hashing of TCP flows to `tx-queues`. In our experiments, the server had five flows (one per client) and a NIC with six `tx-queues`. If two flows were hashed into one `tx-queue` and the rest into three others, then the two flows sharing a `tx-queue` got half the throughput of the other three because all `tx-queues` were transmitting at the same rate. We call this phenomenon (i.e., hashing unevenness causing a winner-loser pattern of throughput) *Hash-Cast* (see Figure 3).

Hash-Cast explains the performance anomalies illustrated in Figure 2. 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 has the highest probability, around 69%.

To eliminate hashing unfairness, we used only a single `tx-queue` and then 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. Most importantly, SFQ used 127 software queues so that hash collisions were much less probable compared to using only six. To further alleviate the effect of collisions, we set SFQ's hashing perturbation rate to 10 seconds, 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 queueing 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. We have also found Hash-Cast to be related to Bufferbloat [2].

Does Statefulness Make NFSv4.1 Slow?

The biggest difference that distinguishes NFSv4.1 from NFSv3 is statefulness. Both NFSv2 and NFSv3 were designed to be stateless (i.e., the server does not maintain clients' states). A stateless server is easy to implement, but it precludes stateful tasks such as mounting, file locking, quota management, etc. Consequently, NFSv2/v3 pushed these tasks onto standalone services running on separate (and sometimes ad hoc) ports, causing maintenance problems (especially when clients need to access these ports through firewalls). Being stateful, NFSv4.1 can consolidate all its services to run on single well-known port 2049, which simplifies configuration and maintenance. However, a stateful protocol introduces overhead messages to maintain state. We ran tests to characterize this overhead.

Our tests validated that the stateful NFSv4.1 is more talkative than NFSv3. Figure 4 shows the number of requests made by NFSv4.1 and NFSv3 when we ran the Filebench File-Server workload for five minutes. For both the zero and 10 ms-latency networks, NFSv4.1 achieved lower throughput than NFSv3 despite making more requests. In other words, NFSv4.1 needs more communication with the server per file operation. In Figure 4, more than one third of NFSv4.1's requests are `OPEN` and `CLOSE`, which are used to maintain states. We also observed in

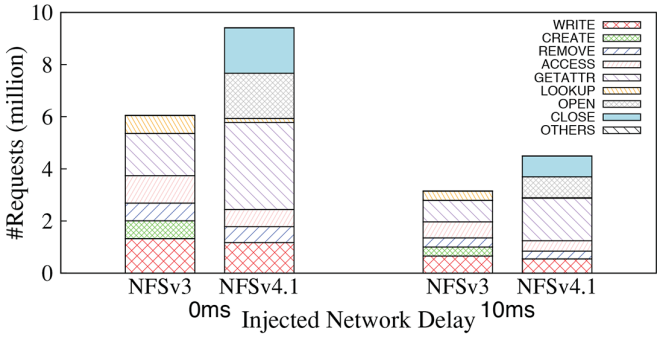

Figure 4: Number of NFS requests made by the File Server workload. Each NFSv4.1 operation represents a compound procedure. For clarity, we omit trivial and rare operations in the same compound (e.g., `PUTFH`, `SEQUENCE`, etc.).

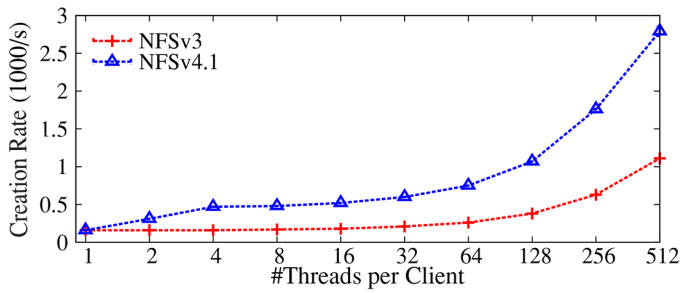

Figure 5: Rate of creating empty files in a 10 ms-delay network

other tests (particularly with a single thread and high network latency) that NFSv4.1's performance was hurt by its verbosity.

Verbosity is the result of NFSv4.1's stateful nature. To combat that effect, NFSv4.1 provides compound procedures, which can pack multiple NFS operations into one RPC. Unfortunately, compounds are not very effective: 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 multi-operation compounds is difficult.

Nevertheless, statefulness also helps performance. Figure 5 shows the speed of creating empty files in the 10 ms-delay network: NFSv4.1 increasingly outperformed NFSv3 as the number of threads grew. This is because of NFSv4.1's asynchronous RPC calls, which allow the networking layer (TCP Nagle) to coalesce multiple RPC messages. Sending fewer but larger messages is faster than sending many small ones, so NFSv4.1 achieved higher performance. Because NFSv3 is stateless, all its metadata-mutating operations have to be synchronous; otherwise a server crash might lose data. NFSv4.1, however, is stateful and can perform metadata-mutating operations asynchronously because it can restore states properly in case of server crashes.

What Is the Impact of Sessions?

Sessions are a major feature of NFSv4.1; they offer Exactly-Once Semantics (EOS). To work around network outages, NFS has evolved from UDP-only (NFSv2), to both-UDP-and-TCP (NFSv3), and now to TCP-only (NFSv4.1). However, using TCP does not solve all problems. Should a TCP connection be completely disrupted, RPC requests and replies might be lost and need to be retried once another connection is established. An NFS request might be executed more than once if (1) it was received and executed by the server, (2) its reply to the client got lost, and (3) the client retried after reconnecting. This causes problems for non-idempotent operations such as `rename`.

To solve this problem, an NFS server has a Duplicate Requests/Reply Cache (DRC), which saves the replies of executed requests. If the server finds a duplicate in the DRC, it simply resends the cached reply without re-executing the request. However, when serving an older client, the server cannot know how many replies might be lost, because clients do not acknowledge them. This means that the DRC is effectively unbounded in size.

NFSv4.1's sessions solve the problem by bounding the number of unacknowledged replies. The server assigns each session a number of slots, each of which allows one outstanding request. By reusing a session slot, a client implicitly acknowledges that the reply to the previous request using that slot has been received. Thus, the DRC size is bounded by the total number of session slots, making it feasible to keep DRC in persistent storage: for example, a small amount of NVRAM, which is necessary to achieve EOS in the face of crashes.

However, 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. In our tests, we observed that the lack of session slots can affect NFSv4.1's performance. In Figure 6, session slots became a bottleneck when the thread count increased above 64, and thus NFSv4.1 performed worse than NFSv3. Note that the number of session slots is a dynamic value negotiated between the client and server. However, Linux has a hard limit of 160 on the number of slots per session.

How Much Does Delegation Help?

A key new feature of NFSv4.1 is delegation, a client-side caching mechanism that allows cached data to be used without lengthy revalidation. Caching is essential to good performance, but in distributed systems like NFS it creates consistency problems. NFS clients need to revalidate their cache to avoid reading stale

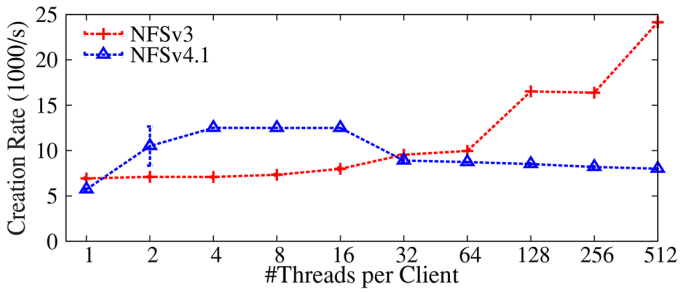

Figure 6: Rate of creating empty files in a zero-delay network

data. In practice, revalidation happens often, causing extra server load and adding delay in high-latency networks.

In NFSv4.1, the cost of cache validation is reduced by letting a server *delegate* a file to a particular client for a limited time. While holding the delegation, a client need not revalidate the file's attributes or contents. If any other clients want to perform conflicting operations, the server can recall the delegation using *callbacks*. Delegations are based on the observation that file sharing is infrequent and rarely concurrent [4]. Thus, they boost performance most of the time, but can hurt performance in the presence of concurrent and conflicting file sharing.

We studied *read delegations* (which are the only type currently supported in the Linux kernel). In Linux, a read delegation is granted if (1) the back channel to the client is working, (2) the client is opening the file with `O_RDONLY`, and (3) the file is not open for write by any client. The benefits of delegations appear in Figure 1, where they helped NFSv4.1 read small files around 172χ faster than NFSv3. In Figure 1, NFSv4.1 was simply reading from the local cache without any server communication at all, whereas NFSv3 had to send repeated `GETATTR`s for cache revalidation.

Read delegations can also improve the performance of file locking. We quantified the improvement by pre-allocating 1000 4 KB files in a shared NFS directory. Each client repeatedly opened each file, locked it, read the entire file, and then unlocked it. After ten repetitions the client moved to the next file.

Table 2 shows the number of operations performed by NFSv3 and by NFSv4.1 with and without delegation. Only NFSv4.1 shows `OPEN`s and `CLOSE`s because only NFSv4.1 is stateful. When delegations were on, NFSv4.1 used only 1000 `OPEN`s 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 NFSv4.1 did not use any `LOCK`s or `LOCKU`s because, with delegations, locks were also processed locally.

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: NFS operations performed by each client for NFSv3 and NFSv4.1 (with and without delegations). Each NFSv4.1 operation represents a compound procedure. For clarity, we omit trivial and rare operations in the same compound (e.g., `PUTFH`, `SEQUENCE`, `FSINFO`). NFSv3's `LOCK` and `LOCKU` (i.e., unlock) come from the Network Lock Manager (NLM).

Without a delegation (NFSv3 and NFSv4.1 with delegations off in Table 2), 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 invalidate the cache [1].

Another major difference among the columns in Table 2 was the number of `GETATTR`s. In the absence of delegations, `GETATTR`s were used for two purposes: to revalidate the cache upon file open, and to update file metadata upon read. The latter `GETATTR`s were needed because the locking preceding the read invalidated both the data and metadata caches for the locked file.

In total, delegations cut the number of NFSv4.1 operations by over 29χ (from 88 K to 3 K). This enabled the original stateful and "chattier" NFSv4.1 (with extra `OPEN`, `CLOSE`, and `FREE_STATEID` calls) to finish the same workload using only 5% of the requests used by NFSv3. These reductions translate to a 6–193χ speedup in networks with 0–10 ms latency.

Nevertheless, users should be aware of delegation conflicts, which incur expensive cost. We tested delegation conflicts by opening a delegated file with `O_RDWR` from another client and observed that the open was delayed for as long as 100 ms because the NFS server needed to recall outstanding delegations upon conflicts.

We have also observed that read delegations alone are sometimes not enough to significantly boost performance because writes quickly became the bottleneck, and the overall performance was

thus limited. This finding calls for further investigation into how delegations can improve write performance.

Conclusions

We have presented a comprehensive performance benchmarking of NFSv4.1 by comparing it to NFSv3. Our study found that:

1. Linux's NFSv4.1 implementation is stable but suffers from some performance issues.
2. NFSv4.1 is more talkative than NFSv3 because of statefulness, which hurts NFSv4.1's performance and makes it slower in low-latency networks (e.g., LANs).
3. In high-latency networks (e.g., WANs), however, NFSv4.1 performed comparably to and even better than NFSv3, since NFSv4.1's statefulness permits higher concurrency through asynchronous RPC calls.
4. NFSv4.1 sessions can improve correctness while reducing the server's resource usage, but the number of session slots can be a scalability bottleneck for highly threaded applications.
5. NFSv4.1's read delegations can effectively avoid cache revalidation and improve performance, especially for applications using file locks, but delegation conflicts can incur a delay of at least 100 ms.
6. Multi-queue NICs suffer from the Hash-Cast problem and can cause unfairness among not only NFS clients but any data-intensive TCP flows.

Due to space limits, we have presented only the major findings and have omitted some details in explanation; please refer to our SIGMETRICS paper [1] for more results and further details.

Limitations and Future Work

Our study focused only on the performance of NFSv4.1; other aspects such as security (RPCSEC_GSS) and scalability (pNFS) should be interesting subjects to study in the future. Most of our workloads did not share files among clients. Because sharing is infrequent in the real world [4], it is critical that any sharing be representative. Finally, a different NFSv4.1 implementation than the Linux one might (and probably would) produce different results.

Acknowledgments

We thank Lakshay Akula, Vasily Tarasov, Arun Olappamanna Vasudevan, and Ksenia Zakirova for their help in this study. This work was made possible in part thanks to NSF awards CNS-1223239, CNS-1251137, and CNS-1302246.

References

[1] 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 SIGMETRICS 2015*, Portland, OR, June 2015, ACM. Forthcoming.

[2] 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.

[3] D. Hildebrand and P. Honeyman, "Exporting Storage Systems in a Scalable Manner with pNFS," in *Proceedings of MSST,* Monterey, CA, 2005, IEEE.

[4] A. Leung, S. Pasupathy, G. Goodson, and E. Miller, "Measurement and Analysis of Large-Scale Network File System Workloads," in *Proceedings of the USENIX Annual Technical Conference,* Boston, MA, June 2008, pp. 213–226.

[5] A. McDonald, "The Background to NFSv4.1," *;login:*, vol. 37, no. 1, February 2012, pp. 28–35.

[6] NFS: avoid nfs_wait_on_seqid() for NFSv4.1: http://www.spinics.net/lists/linux-nfs/msg47514.html.

[7] Scott Rixner, "Network Virtualization: Breaking the Performance Barrier," *Queue,* vol. 6, no. 1, January 2008, pp. 36–37 ff.