Newer Is Sometimes Better: An Evaluation of NFSv4.1

Ming Chen,[1] Dean Hildebrand,[2] Geoff Kuenning,[3]
mchen@cs.stonybrook.edu, dhildeb@us.ibm.com, geoff@cs.hmc.edu
Soujanya Shankaranarayana,[1] Bharat Singh,[1] and Erez Zadok[1]
{soshankarana, bhasingh, ezk}@cs.stonybrook.edu
[1]Stony Brook University, [2]IBM Research—Almaden, and [3]Harvey Mudd College
Appears in the proceedings of the 2015 ACM SIGMETRICS conference

ABSTRACT

The popular Network File System (NFS) protocol is 30 years old. The latest version, NFSv4, is more than ten years old but has only recently gained stability and acceptance. NFSv4 is vastly different from its predecessors: it offers a stateful server, strong security, scalability/WAN features, and callbacks, among other things. Yet NFSv4's efficacy and ability to meet its stated design goals had not been thoroughly studied until now. This paper compares NFSv4.1's performance with NFSv3 using a wide range of micro- and macro-benchmarks on a testbed configured to exercise the core protocol features. We (1) tested NFSv4's unique features, such as delegations and statefulness; (2) evaluated performance comprehensively with different numbers of threads and clients, and different network latencies and TCP/IP features; (3) found, fixed, and reported several problems in Linux's NFSv4.1 implementation, which helped improve performance by up to 11×; and (4) discovered, analyzed, and explained several counter-intuitive results. Depending on the workload, NFSv4.1 was up to 67% slower than NFSv3 in a low-latency network, but exceeded NFSv3's performance by up to 2.9× in a high-latency environment. Moreover, NFSv4.1 outperformed NFSv3 by up to 172× when delegations were used.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Performance Attributes

General Terms

Measurement, Performance

Keywords

Network File System; NFS; NFSv4.1

1. INTRODUCTION

Since its introduction, the Network File System (NFS) protocol has become a highly popular network-storage solution. Over 90% of enterprise storage capacity is served by network-based storage, and NFS represents a significant proportion of that total [29, 32]. Faster networks, the proliferation of virtualization, and the rise of cloud computing all contribute to continued increases in NFS deployments [1].

The continuous development and evolution of NFS has been critical to its success. 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 [26]; (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 multiple data servers and thus greatly improves performance and scalability [14, 26].

V4.1 became ready for production deployment only a couple of years ago [10, 21]. 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 it 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 [21], in a wide range of environments using representative workloads.

This paper has four contributions: **(1)** a comprehensive comparison of the performance of V3 and V4.1 in low- and high-latency networks, using a wide variety of micro- and macro-workloads; **(2)** insightful performance analysis that sheds light on how underlying system components (networking, RPC, and local file systems) influence NFS's performance; **(3)** a deep analysis of the performance effect of V4.1's unique features (statefulness, sessions, delegations, etc.) in its Linux implementation; and **(4)** fixes to Linux's V4.1 implementation that improve its performance by up to 11×.

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;

SIGMETRICS'15, June 15–19, 2015, Portland, OR, USA.

ACM 978-1-4503-3486-0/15/06 ...$15.00.
http://dx.doi.org/10.1145/2745844.2745845.

The rest of this paper is organized as follows. Section 2 describes our benchmarking methodology. Sections 3 and 4 discuss the results of data- and metadata-intensive workloads, respectively. Section 5 explores NFSv4's delegations. Section 6 examines macro-workloads using Filebench. Section 7 overviews related work. We conclude in Section 8, and discuss this study's limitations and future work in Section 9.

2. METHODOLOGY

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. The storage configuration was able to achieve read throughputs of up to 860MB/s. 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 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.

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. 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 took too long just to initialize. 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 paper.

We benchmarked four kinds of workloads:

1. Data-intensive micro-workloads that test the ability of NFS to maximize network and storage bandwidth (Section 3);
2. Metadata-intensive micro-workloads that exercise NFS's handling of file metadata and small messages (Section 4);
3. Micro-workloads that evaluate delegations, which are V4's new client-side caching mechanism (Section 5); and
4. Complex macro-workloads that represent real-world applications (Section 6).

3. DATA-INTENSIVE WORKLOADS

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

3.1 Random Read

We begin with a workload where five NFS clients randomly read a 20GB file with a given I/O size. 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.

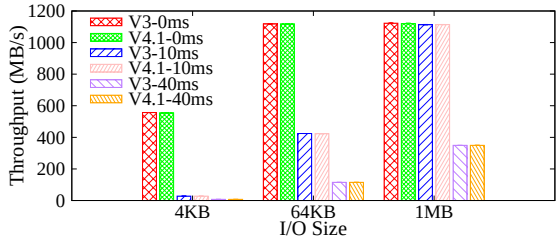


Figure 1: Random-read throughput with 16 threads and different network delays (varying I/O size).

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 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 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.

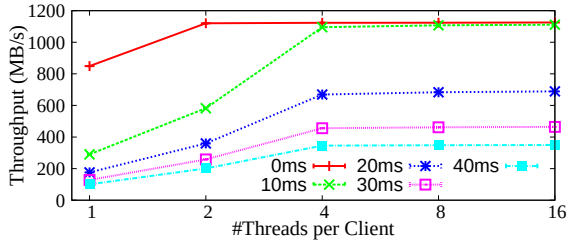


Figure 2: Random-read throughput with 1MB I/O size, default 2MB TCP maximum buffer size, and different network delays (varying the number of threads per client).

However, when we increased the network delay further, from 10ms to 40ms, we could not saturate the 10GbE network (Figure 2) even if we added more threads and used larger I/O sizes. As shown in Figure 2, the curves for 20ms, 30ms, and 40ms reached a limit at 4 threads. We found that this limit was caused by the NFS server's maximum TCP buffer sizes (`rmem_max` and `wmem_max` size), which restricted TCP's congestion window (i.e., the amount of data on the wire). To saturate the network, the `rmem_max` and `wmem_max` sizes must be larger than the network's 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 paper.

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.

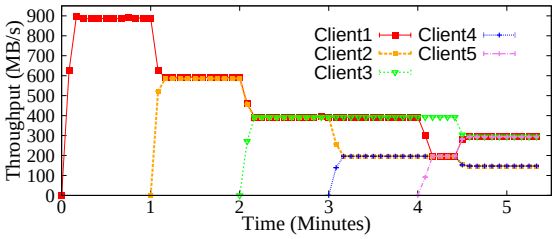


Figure 3: Sequential-read throughputs of individual clients when they were launched one after the other at an interval of one minute. Results of one run of experiments.

The winner-loser pattern was unexpected given that all the five clients had the same hardware, software, and settings, and were performing the same operations. Initially, we suspected that the pattern was caused by the order in which the clients launched the workload. To test that hypothesis, we repeated the experiment but launched the clients in a controlled order, one additional client every minute. However, the results disproved any correlation between experiment launch order and the winners. Figure 3 shows that Client2 started second but ended up as a loser, whereas Client5 started last but became a winner. Figure 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 [25]. 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 [25]. 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. How-

ever, 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`.)

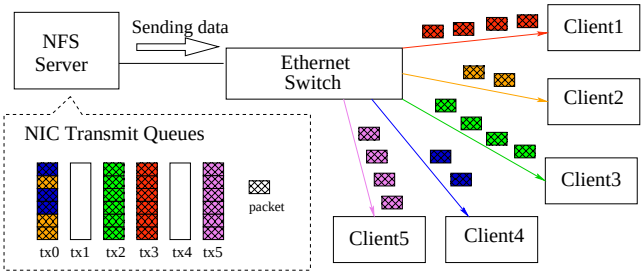


Figure 4: Illustration of Hash-Cast.

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 4.

Hash-Cast explains the performance anomalies illustrated in Figure 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 [11]. 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 [22]. 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 paper we used SFQ with a single `tx-queue`. The random-read results shown in Section 3.1 also used SFQ.

3.3 Random Write

The random-write workload is the same as the random-read one discussed in Section 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 [29], 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.

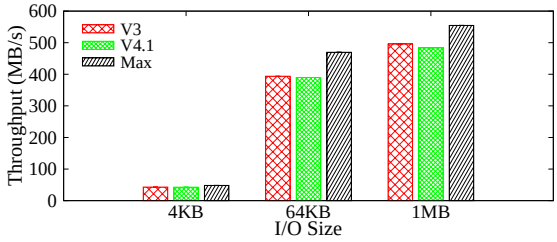


Figure 5: Random-write throughput in a zero-delay network.

Figure 5 shows the random-write throughput when we varied the I/O size in the zero-delay network. V4.1 and V3 achieved around the same throughput, but both were significantly slower than the maximum performance of our SSD RAID (measured on the server side without NFS). Neither V4.1 nor V3 achieved the maximum throughput even with more threads. We initially suspected that the lower throughputs were caused by the network, but the throughput did not improve when we repeated the experiment directly on the NFS server over the loopback device. Instead, we found the culprit to be the `O_SYNC` flag, which has different semantics in `ext4` than in NFS. The POSIX semantics of `O_SYNC` require all meta-data updates to be synchronously written to disk. On Linux's local file systems, however, `O_SYNC` is implemented so that only the actual file data and the meta-data directly needed to retrieve that data are written synchronously; other meta-data remains buffered. Since our workloads used only in-place writes, which updated the file's modification time but not the block mapping, writing an `ext4` file did not update meta-data. In contrast, the NFS implementation more strictly adheres to POSIX, which mandates that the server commit both the written data and "all file system metadata" to stable storage before returning results. Therefore, we observed many meta-data updates in NFS, but not in `ext4`. The overhead of those extra updates was aggravated by `ext4`'s journaling of meta-data changes on the server side. 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. Clients initially realized high throughput (over 1GB/s) since all data was buffered in

their caches. Once the number of dirty pages passed a threshold, the throughput dropped to near zero as the clients began flushing those pages to the server; this process took up to 3 minutes depending on the I/O size. After that, the write throughput became high again, until caches filled—and the bursty pattern then repeated.

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 single-threaded sequential-write throughputs were lower than the corresponding multi-threaded random-write throughputs because our all-SSD storage backend favors multi-threaded I/O accesses due to the SSD's internal parallelism [2]. For sequential writes, the `O_SYNC` behavior we discussed in Section 3.3 had an even stronger effect if the backend storage used HDDs, because the small disk writes generated by the meta-data updates and the associated journaling broke the sequentiality of NFS's writes to disk. We measured a 50% slowdown caused by this effect when we used HDDs for our storage backend instead of SSDs [8].

4. METADATA-INTENSIVE WORKLOADS

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

4.1 Read Small Files

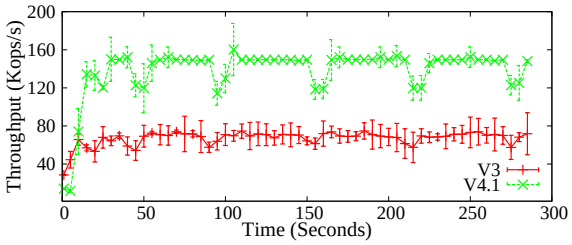


Figure 6: Throughput of reading small files with one thread in a zero-delay network.

We pre-allocated 10,000 4KB files on the NFS server. Figure 6 shows the results of the 5 clients randomly reading entire files repeatedly for 5 minutes. The throughputs of both V3 and V4.1 increased quickly during the first 10 seconds and then stabilized once the clients had read and cached all files. V4.1 started slower than V3, but outperformed V3 by 2× after their throughputs stabilized. We observed that V4.1 made 8.3× fewer NFS requests than V3 did. The single operation that caused this difference was GETATTR, which accounted for 95% of all the requests performed by V3. These GETATTRs were being used by the V3 clients to revalidate their client-side cache. However, V4.1 rarely made any requests once its throughput had stabilized. Further investigation revealed that this was due to 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 Section 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.

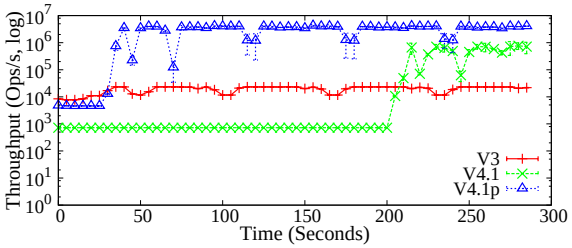


Figure 7: Throughput of reading small files with 16 threads in a 10ms-delay network ($\log_{10}$).

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 7 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 6), 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_queue`s 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 [26]; 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 [26] 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 7. 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 that of V3 (thanks to 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 6 and 7, which corresponds to the `actimeo` mount option. When this timer expires, client-cached metadata must again be retrieved from the server, temporarily lowering throughput.

4.2 File Creation

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

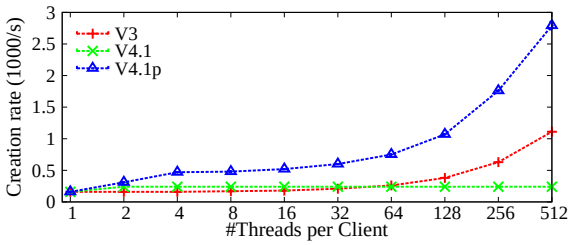


Figure 8: Rate of creating empty files in a 10ms-delay network.

Figure 8 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 8, 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 Section 4.1, its performance did not improve at all once the thread count increased beyond two. With the `seqid` problem fixed, V4.1p scaled much better, with an average boost of 34% per step. With 16–512 threads, V3's scalability improved significantly, and it achieved a high average performance boost of 44% per step; V4.1p also scaled well with an average boost of 40% per step.

V4.1p always outperformed the original V4.1, by up to 11.6× with 512 threads. Therefore, for the rest of this paper, 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

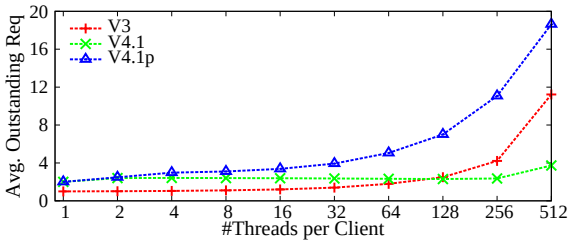


Figure 9: Average number of outstanding requests when creating empty files in a 10ms-delay network.

than 4 threads, and by less than 27% with fewer threads; thus, they all had the same trends. As shown in Figure 8, 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 9, which closely resembles Figure 8 in overall shape. This suggests that V4.1p performed faster than V3 because the V4.1p clients sent more requests to the server at one time. We examined the client code and discovered that V3 clients use synchronous RPC calls (rpc_call_sync) to create files, whereas V4.1p clients use asynchronous calls (rpc_call_async) that go through a work queue (nfsiod_workqueue). We believe that the asynchronous calls are the reason why V4.1p had more outstanding requests: the long network delay allowed multiple asynchronous calls to accumulate in the work queue and be sent out in batch, allowing networking algorithms such as TCP Nagle to efficiently coalesce multiple RPC messages. Sending fewer but larger messages is faster than sending many small ones, so V4.1p achieved higher rates. Our analysis was confirmed by the `mountstats` data, which showed that V4.1p's OPEN requests had significantly longer queuing times (up to 30×) on the client side than V3's CREATEs. (V3 uses CREATEs to create files whereas V4.1p uses OPENs.) Because V3's server is stateless, all its mutating operations have to be synchronous; otherwise a server crash might lose data. V4, however, is stateful and can perform mutating operations asynchronously because it can restore states properly in case of server crashes [23].

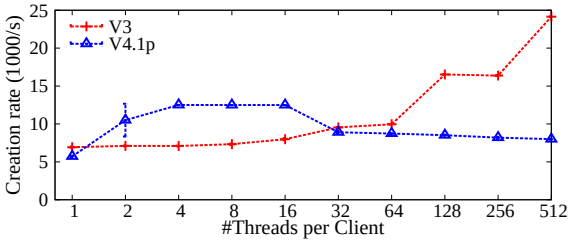


Figure 10: Rate of creating empty files in a zero-delay network.

In the zero-delay network, there was not a consistent winner between the two NFS versions (Figure 10). 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 num-

ber of outstanding requests closely resembles Figure 10. 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 11, waiting began at 32 threads, which is also where V4.1p's performance began dropping (Figure 10). 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.)

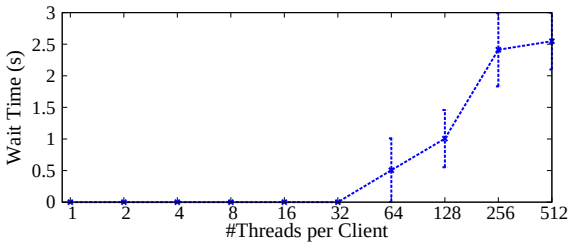


Figure 11: Average waiting time for V4.1p's session slots of ten experimental runs. Error bars are standard deviations.

We note that Figure 11 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 [28]. 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 10), V4.1p's performance advantage over V3 was because of V4.1p's asynchronous calls (the same as explained above); V4.1p's OPENs had around 4× longer queuing time, which let multiple requests accumulate and be sent out in batch. This queuing time was not caused by the lack of available session slots (Figure 11). This was verified by evaluating the use of a single thread, in which case V4.1p performed 17% slower than V3 because V4.1p's OPEN requests are more complex and took longer to process than V3's CREATE (see Section 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 10).

4.3 Directory Listing

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

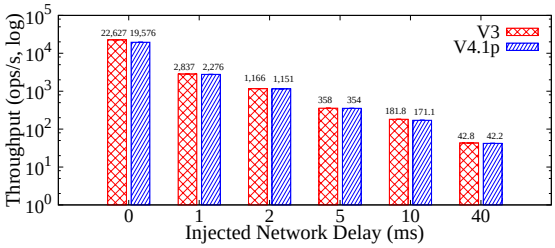


Figure 12: Directory listing throughput ($\log_{10}$).

This workload is read-only, and showed behavior similar to that of reading small files (Section 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 12 (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.

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.)

5. NFSV4 DELEGATIONS

In this section, we discuss delegations, an advanced client-side caching mechanism that is a key new feature of NFSv4. Caching is essential to good performance in any system, but in distributed systems like NFS it gives rise to consistency problems. V2 and V3 explicitly ignored strict consistency [7, 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. While holding the delegation, a client need not revalidate its attributes or contents. 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 [26] and rarely concurrent [16]. Thus, they can boost performance most of the time, but can also hurt performance in the 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 [12].

5.1 Granting a Delegation

An open delegation is granted when a client opens a file with an appropriate flag. However, clients must not assume that a delegation will be granted, because that choice is up to the server. If a delegation is rejected, the server can explain its decision via flags in the open reply (e.g., lock contention, unsupported delegation type). Even if a delegation is granted, the server is free to recall it at any time via the back channel. 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.

5.2 Delegation Performance: Locked Reads

We previously showed the benefit of delegations in Figure 7, 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 [13] 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.

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 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).

Table 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 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 problem is caused by the timestamp granularity on the NFS server. Traditionally, NFS provides close-to-open cache consistency. 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 [29], media files, etc. The problem is worsened by two factors: (1) invalidation happens even when the client is simply acquiring read (not write) locks, and (2) a file's entire cache contents are invalidated even if the lock only applies to a single byte. In contrast, the NFS client with delegation was able to satisfy nine of the ten repeated READs from the page cache. There was no need to revalidate the cache at all because its validity was guaranteed by the delegation.

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

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

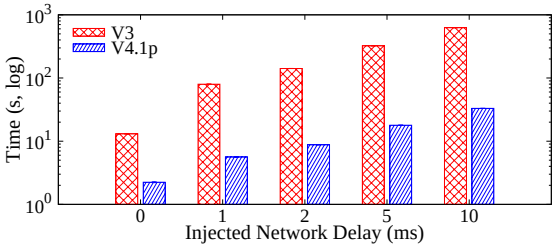


Figure 13: Running time of the locked-reads experiment ($\log_{10}$). Lower is better.

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 13, these reductions translate to a 6–19× speedup in networks with 0–10ms latency.

5.3 Delegation Recall Impact

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× 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 [3]. Also, because Linux does not support write delegations, Linux NFS clients do not have any dirty data (of delegated files) to write back to the server, and thus should be able to return delegations quickly. We believe it would be better to start with a much shorter timeout; if that turns out to be too small, the client will back-off quickly anyway since the timeout increases exponentially.

6. MACRO-WORKLOADS

We now turn to macro-workloads that mix data and meta-data operations. These are more complex than micro-workloads, but also more closely match the real world. Our study used Filebench's File Server, Web Server, and Mail Server workloads [15].

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 [30].

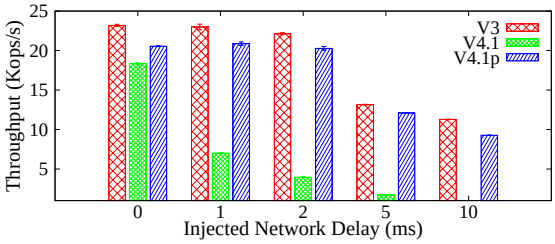


Figure 14: File Server throughput (varying network delay)

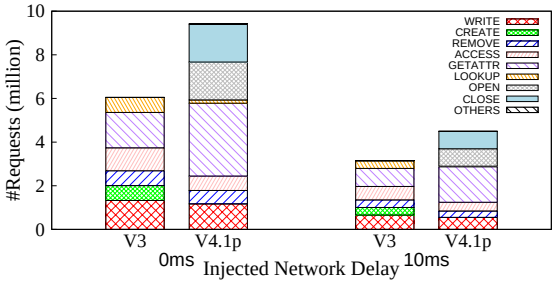


Figure 15: Number of NFS requests made by the File Server

As shown in Figure 14, 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 15), 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 (Section 4.1), where V4 was 40% slower than V3 with a single thread and a 10ms-delay network. Verbosity is the result of V4's stateful nature, and the V4 designers were aware of the issue. To reduce verbosity, V4 provides compound procedures, which pack multiple NFS operations into one message. However, compounds have not been implemented effectively in Linux (and other OSes): most contain only 2–4 often-trivial operations (e.g., SEQUENCE, PUTFH, and GETFH); and applications currently have no ability to generate their own compounds. We believe that implementing effective compounds is difficult for two reasons: (1) The POSIX API dictates a synchronous programming model: issue one system call, wait, check the result, and only then issue the next call. (2) Without transaction support, failure handling in compounds with many operations is fairly difficult.

In this File Server workload, even though V4.1p made a total of 56% more requests than V3, V4.1p was only 12% slower because its asynchronous calls allowed 40–95% more outstanding requests (as explained in Section 4.2). When we injected delay into the network (Figure 14), 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 14 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× throughput improvement. We conclude that our patch helps V4.1's performance in both micro- and macro-workloads, especially as network delays increase.

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.

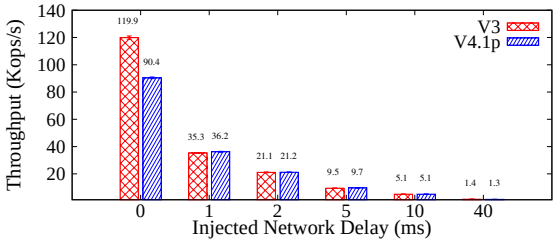


Figure 16: Web Server throughput (varying network delay).

Figure 16 shows the throughput with different network delays. V4.1p was 25% slower than V3 in the zero-delay network. The `mountstats` data showed that the average round-trip time (RTT) of V4.1p's requests was 19% greater than for V3. As the network delay increased, the RPC RTT became overshadowed by the delay, and V4.1p's performance became close to V3's and even slightly better (up to 2.6%). V4.1p's longer RTT was due to its complexity and longer processing time on the server side, as explained in Section 4.3. With longer network delays, V4.1p's performance picked up and matched V3's because of its use of asynchronous calls.

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

To characterize the bottleneck, we varied the number of threads in the workload and repeated the experiments with delegations both on and off. Figure 17 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 17 converged. With delegations, V4.1p was 2.2× faster than V3 when using one thread. However, V4.1p began to slow down with 4 threads, whereas V3 sped up and did not slow down until the thread number increased to 64. The

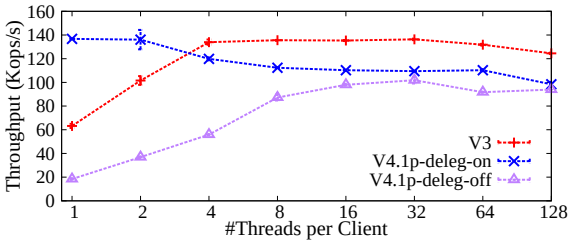


Figure 17: Web Server throughput in the zero-delay network (varying thread count per client).

eventual slowdown of both V3 and V4.1p was because the system became overloaded when the log-writing bottleneck was hit. However, V4.1p hit the bottleneck with fewer threads than V3 did because V4.1p, with delegations, only performed repeated WRITEs, whereas V3 performed ten GETATTRs (for cache revalidation) before each WRITE. With more than 32 threads, V4.1p's performance was also hurt by waiting for session slots (see Section 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.

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.

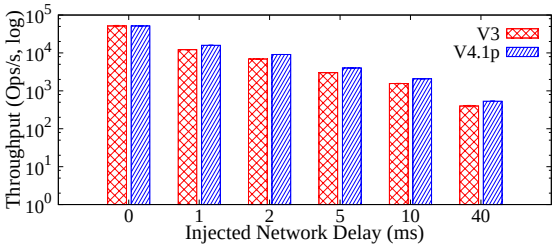
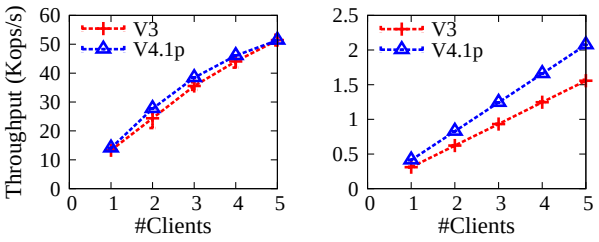


Figure 18: Mail Server throughput (varying network delay)

Figure 18 (note the log Y scale) presents the Mail Server throughput with different network delays. Without delay, V4.1p and V3 had the same throughput; with 1–40ms delay, V4.1p was 1.3–1.4× faster. Three factors affected V4.1p's performance: (1) V4.1p made more NFS requests for the same amount of work (see Section 6.1); and (2) V4.1p's operations were more complex and had longer RPC round-trip times (see Section 4.3); but (3) V4.1p made many asynchronous RPC calls and helped the networking algorithms coalesce RPC messages (see Section 4.1). Although the first two factors hurt V4.1p's performance, the third more than compen-

sated for them. Increasing the network delay did not change factor (1), but diminished the effect of (2) as the delay gradually came to dominate the RPC RTT. Longer network delays also magnified the benefits of factor (3) because longer round trips were mitigated by coalescing requests. Thus, V4.1p increasingly outperformed V3 (1.3–1.4×) as the delay grew. V4.1p's read delegations did not help in this workload because most of its activities write files (reads are largely cached). This again shows the potential benefit of write delegations, even though Linux does not currently support them.



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

Figure 19: Mail Server throughput (varying client count)

Figure 19 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.

7. RELATED WORK

NFS versions 2 and 3 are popular and have been widely deployed and studied. Wittle and Keith designed the LADDIS NFS workload and measured NFS's response time and throughput under various loads [31]. Based on LADDIS, the SPECsfs suites were designed to benchmark and compare the performance of different NFS server implementations [27]. Martin and Culler [20] 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 [9]; 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 [6] 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 [17].

Most prior studies [6, 9, 17, 20, 27, 31] 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 [4] implementation. Radkov et al. compared the performance of a prototype version of V4.0 and iSCSI in IP-networked storage [24]. Martin [19] compared the file operation performance between Linux V3 and V4.0; Kustarz [15] 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. [18] 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 [5]. Gulati et al. built a V4.0 cache proxy, also using delegations, to improve NFS's performance in WANs [13]. 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. 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 [12]. 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.

8. CONCLUSIONS

We have presented a comprehensive benchmarking study of Linux's NFSv4.1 implementation. 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×.

9. LIMITATIONS AND FUTURE WORK

This work has two limitations that can be addressed in the future: **(1)** Most of our workloads did not share files among clients. Because sharing is infrequent in the real world [26], 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.

Lastly, we believe that V4.1's compound procedures hold much promise but are woefully underutilized. 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.

Acknowledgments

We thank the anonymous SIGMETRICS reviewers for their valuable comments. We also 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.

10. REFERENCES

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

[3] 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.

[4] 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.

[5] 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.

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

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

[8] 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.

[9] 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.

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

[11] 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`.

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

[13] 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.

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

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

[16] 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 USENIX Annual Technical Conference*, pages 213–226, Boston, MA, June 2008.

[17] 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.

[18] 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.

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

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

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

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

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

[24] 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.

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

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

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

[28] V. Tarasov, S. Bhanage, E. Zadok, and M. Seltzer. Benchmarking File System Benchmarking: It *IS* Rocket Science. In *Proceedings of HotOS*, Napa, CA, May 2011.

[29] Vasily Tarasov, Dean Hildebrand, Geoff Kuenning, and Erez Zadok. Virtual machine workloads: The case for new benchmarks for NAS. In *Proceedings of USENIX FAST*, San Jose, CA, February 2013.

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

[31] 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.

[32] 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.