On one source of latency in NFSv4 client

Dmitry Irtegov
Novosibirsk State University
630090 Pirogova 1, Novosibirsk,
Russia
fat@nsu.ru

Pavel Belousov
Novosibirsk State University
630090 Pirogova 1, Novosibirsk,
Russia
firix-pavel@yandex.ru

Alexander Fal
Novosibirsk State University
630090 Pirogova 1, Novosibirsk,
Russia
falalexandr007@gmail.com

Alexey Fedosenko
Novosibirsk State University
630090 Pirogova 1, Novosibirsk,
Russia
fedos-alexey@yandex.ru

ABSTRACT

Due to VFS architecture limitation, Linux NFSv4 and 4.1 client cannot join RPC requests into compounds even in cases when it is allowed by protocol specification. This leads to the high sensitivity to the network latency and loss of performance on metadata-intensive operations, especially on workloads when many small files are opened. Similar issue exists in other Unix-like kernels. We propose a modification to VFS API that resolves this issue. We have a demo implementation of modified VFS and NFS client that shows measurable improvement of latency and general throughput on synthetic metadata-intensive tests, even with standard NFS servers.

CCS CONCEPTS

• Networks → Network performance evaluation → Network performance analysis • Networks → Network protocols → Network File System (NFS) protocol

KEYWORDS

shared storage, latency, compound RPC, Linux, VFS

ACM Reference format

D.Irtegov, P.Belousov, A.Fal, A.Fedosenko. 2017. On one source of latency in NFSv4 client. In Proceedings of CEE-SEC(R)'2017, St.Petersburg, Russia, 9 pages.

1 INTRODUCTION

File system performance cannot be adequately described by the single parameter. For some applications, most important parameter might be the sustained throughput when writing to a single file, for other applications a read throughput or random access speed can be more important, etc. For some applications, like web servers, most important performance parameter is the latency when working with large number of small files. This is one of the reasons why there are so many different file systems and so many technologies and approaches for network file storage.

When choosing a storage for load-balanced web cluster [1] we found that SAN technologies like iSCSI have the best latency on most types of workload, but do not provide shared storage (no concurrent read-write access to a single LUN). NFS provides shared storage but has relatively high latency and other popular network file systems, like CIFS, have even higher latency. There also exists a class of solutions known as cluster file systems [2, 3], but they also have their own limitations.

The fact that NFS has high latency compared to SAN on same hardware is confirmed by many researchers [4, 5]. But there is no generally accepted explanation of this. Authors of [4] state than NFS is slow on metadata-intensive operations but do not try to find a root cause of this. Authors of [5] assume that this is probably connected to POSIX semantics.

We identified one source of latency in Linux NFSv4 client associated with file lookup operation. Most operations that are called metadata-intensive in paper [4] actually do a file lookup, so this is probably the important part of general latency.

It is possible to improve a latency of the file lookup, but it requires a change not only in NFS client kernel module, but also in kernel VFS framework. We did this change and have an experimental implementation of improved NFS that demonstrates measurable better latency on metadata-intensive workloads. We also found that similar issue exists in other Unix-like kernels (Solaris and FreeBSD), so our approach can be useful for improving NFS performance in these kernels too.

2 NFS PROTOCOL

2.1 Brief history of NFS

Network File System protocol was originally developed by Sun Microsystems in 1980s. A decade later, Sun Microsystems handled the development of the protocol to IETF. NFSv4 and current NFSv4.1 protocol specifications have status of IETF standards [6, 7].

NFS is supported by practically all popular general-purpose operating systems, including Unix-like systems (Unix System V, *BSD, MacOS, Linux), Microsoft Windows, IBM z/OS. It is also widely supported by SAN/NAS vendors.

NFS is a RPC (Remote Procedure Call) protocol, more or less directly mappable to the POSIX file operations. Actually it maps to RPC not the POSIX system calls (open(2), read(2), write(2), etc), but internal calls made by kernel VFS (Virtual File System) framework. Historically, VFS framework in SunOS and early versions of NFS were developed concurrently and had some convergent evolution. VFS framework in Linux is not directly related to SunOS (different licenses explicitly forbid sharing the code between these kernels), but its architecture shows the signs of a significant indirect influence.

NFSv4 offered significant changes compared to previous version 3. These changes are related mostly to performance and security. List of performance-related changes includes (but is not limited to):

1. Stateful operation, contrasting to previous attempts to make the protocol stateless
2. Mandatory use of the TCP transport protocol
3. Cache-coherence capabilities known as "Delegation"
4. Compound RPC – a feature allowing to send several RPC requests in one batch.

Implementing these features offered real performance benefits over NFSv3 under most benchmarks and real-world workloads, so all vendors who support NFSv4 recommend using this version when possible. However, many research works [4, 5, 8] demonstrated a complex picture of NFSv4 performance, showing unexplained delays and low throughput under many types of workload.

Probably most interesting is comparison of NFS to SAN technologies, especially iSCSI. iSCSI can be run on the same hardware as NFS, so head-to-head comparison is possible.

2.2 iSCSI and cluster file systems

iSCSI [9] is a standard protocol for storage access networks, supported by most operating systems and SAN/NAS vendors. It can be described as RPC, but these RPC correspond not to filesystem API, but to block device driver calls. Historically, SCSI was a protocol used in peripheral bus interconnects, and some parts of this protocol were standardized long before the term RPC was invented. iSCSI uses TCP connection for transporting SCSI commands. It allows to use Ethernet network hardware so it is generally considered as a cheaper alternative to FiberChannel SANs.

iSCSI is used to provide a disk image (so called SCSI LUN), not a file system. On the server this image might actually be a file. Many implementations of iSCSI servers also can provide raw block devices, disk partitions or LVM volumes as LUNs. After attaching to the LUN, client sees it as a [virtual] block device and can use it as any other type of block device, for placing file systems, LVM physical volumes, swap partitions, etc.

Modern operating systems assume that they have exclusive access to block devices, so most uses of iSCSI LUNs do not allow to use it as shared resource. For example, a typical filesystem driver heavily depends on consistency of on-disk data, but has no locks and semaphores to ensure this consistency during concurrent access to the disk.

Many benchmarks and real-world experience show that iSCSI offers better performance than any known network file system on the same hardware. So iSCSI is displacing network file systems in cases when sharing is not necessary or read-only sharing is OK. It is used for diskless computers, for extension of disk space on servers with limited local disk space, for disk images of virtual machines, etc.

There are numerous attempts to overcome a main limitation of all SAN technologies, inability to work as shared storage. These attempts have no generally accepted common name; we will call them cluster file systems. Examples are Lustre [3], OCFS2 [10], Ceph [2]. These file systems use shared or replicated SAN device and build a distributed locking protocol on top of the device itself or using out-of-band network connections. These filesystems have to deal with distributed lock problem which has no known universal solution [11]. For the purpose of our comparison, we must note, that all these systems are built on top of SAN, and, therefore, cannot work faster than SAN.

Head-to-head comparison of NFS and iSCSI shows that read and write operation of these protocols have similar performance characteristics [4, 5]. It is interesting to note that in the field there is widely used solution to store virtual machine disks and other virtual block storage on NAS servers. All modern Unix-like systems have a feature that allows converting a file to a virtual block device, called loopback device. Loopback devices situated on NFS NAS have high throughput and low latency, similar to SAN on same hardware. However, loopback device images cannot be shared, for same reasons SAN devices cannot.

Where NFS starts to lose is opening the files. Most significant difference can be noticed on workloads that involve opening and reading many small files. The general verdict, repeated in many research works, is that NFS is slow on metadata-intensive operation. Metadata in this context mean system data: file attributes, file allocation on disk, free space, etc.

These tests also show that NFS is much "chattier" protocol, using much more RPC calls than iSCSI for doing similar operations. Again, this is most noticeable when working with small files

This is very strange, because NFS uses high-level operations to work with the metadata. Let's consider an NFS and iSCSI client looking up a file in a directory. Let's consider that parent directory is in the cache, but target directory is not, so we know a directory inode number and file name, but have no directory data and metadata in memory.

On iSCSI, filesystem driver must, in worst case:

1. read and parse inode table metadata to find a relevant extent of inode table
2. read and parse directory inode record
3. read and parse significant part of directory data to find a file directory entry and file inode number
4. again read and parse inode table metadata, because file inode most likely is located in other extent of inode table
5. read file inode record

Note that we need to parse every piece of data to find location of the next piece of data, so read-ahead and command queuing probably will not really help. So, every request must be handled by separate SCSI command.

In comparison, NFS client must:

1. Do a LOOKUP RPC with parent directory handle and target directory name. If successful, this RPC returns a handle to target directory.
2. Do a LOOKUP RPC with target directory handle and file name. This RPC returns a handle to a file.

So, we can do a file lookup in just two RPC calls. All inode table metadata and directory data lookups are local to the server. In practice, NFS clients also read directory and file attributes, but this can be done in a single compound request with a LOOKUP itself, and modern NFSv4 clients actually use compounds for this. Also, NFSv4 protocol specification allows to join all requests related to operations 1 and 2 in single compound RPC.

So, in theory, NFS must do 1 compound RPC calls when iSCSI must do 5 or more separate SCSI commands. This theory obviously contradicts the practice.

2.3 Compound RPC

In modern networks, time of executing a single or compound RPC request by the server usually is small, compared to network RTT (Round Trip Time, see Fig. 1). Our measurement show that even on low end hardware, time of server processing of most RPC requests is smaller than RTT. Measurements on Fig.1 were taken on a single switch 1Gb/s Ethernet network (full specs of hardware used in our tests is presented in section 3).

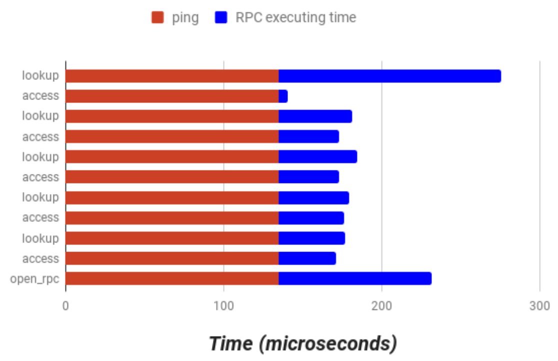

Figure 1: Time of NFS RPC requests compared to ping time. Ping time is used as RTT estimation.

Network RTT consists of time of signal propagation along the cable, and delays in active network equipment (routers, switches, etc). From the protocol standpoint, it is pure waste of time. It is not the same as a network latency, but closely related to it.

Many modern network protocols, including HTTP 2.0, iSCSI, SMB2 and NFSv4 provide command queuing, compound requests and similar means to send requests in batches, to reduce the influence of latency and RTT.

However, if we look on actual RPC requests made by real NFS client when opening the file, we see the sequences of calls, presented in Appendix 1 for Linux, Appendix 2 for Solaris and Appendix 3 for FreeBSD. All these request sequences were produced when an userland process issued a request to open a file file_tests/nfscc/tests/multi_test/multi_test.py on the NFS share.

It is easy to note that all tested clients use rather complex compound sequences (Linux and FreeBSD send 5 RPC in single compound, Solaris sends 9 RPC) for every path component. But all clients send requests for every path component as a separate compound. They do not attempt to join them into a single compound.

Also, it is interesting to note that all these clients send two compound requests per pathname component, one to verify the parent directory and second to actually do a lookup. When doing a compound lookup first step would be redundant.

Simple test shows that this can be the source of the significant latency. We measured the time of open(2) system call with long hierarchical pathnames on NFS and iSCSI on the same hardware, and found than NFS has linear dependency between number of hierarchy levels and open time (see Fig. 2). Similar results were reported by other authors [4, 5].

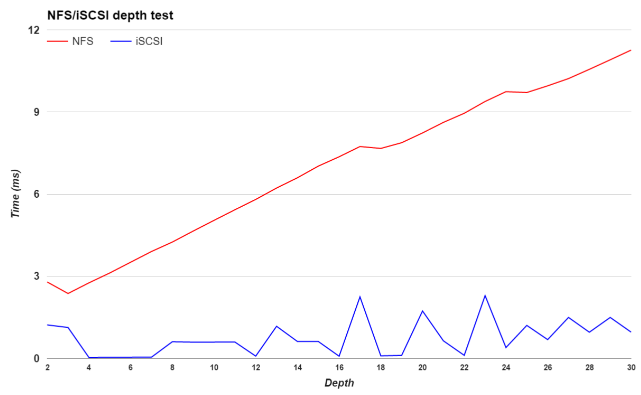

Figure 2: Time of open(2) syscall relative to pathname depth. Measurements were taken on CentOS 7 client and server with CentOS 3.10 kernel build.

There is no requirement in the NFS protocol specification to do lookups in separate compounds. Section 8.7.1 of RFC 7530 even has an example of multilevel compound lookup RPC and description how it must be processed.

Other researchers also noticed this. Authors of the work [5] assume that this is somehow connected with the POSIX semantics. As we will see, this cannot be the case. Indeed, POSIX system call to open a file, open(2), accepts hierarchical pathname as a parameter [12]. So, the POSIX-compliant kernel has all information needed to form a compound lookup request.

To understand why Linux and other modern Unix-like kernels do not use compound lookups, we must discuss internal structure of the kernel, specifically VFS layer.

2.4 Linux Virtual File System framework

The Virtual File System (also known as the Virtual Filesystem Switch) is the software layer in the kernel that provides the filesystem interface to userspace programs. It also provides an abstraction within the kernel which allows different filesystem implementations to coexist [13].The VFS handles all userland file-related system calls and file requests from other kernel modules. It translates these requests into the calls to a specific filesystem driver. It calls functions of the driver, so from the filesystem driver perspective it is a framework.

When processing open(2) and other system calls that take a pathname as a parameter, VFS parses the name to components (directory names and file name). Then it tries to lookup every component. First, it consults a dentry cache. In a case of cache miss, it calls a lookup() function provided by a filesystem driver. The driver must consult on-disk metadata and fill two structures, dentry and inode.

Dentry structure describes a directory entry. Dentry can be positive (for names corresponding to existing files, directories and other filesystem objects) or negative (for names that do NOT correspond to existing objects). Negative dentries are cached because many Unix programs do searches in lists of directories. For example, execvp(2) syscall does a search of executable file in PATH environment variable.

Positive dentry has a pointer to inode structure. Inode contains metadata describing the filesystem object. It has standard attributes returned by stat(2), attributes required by VFS layer and, probably, other information that the FS driver considers worth keeping in memory. Most important, inode structure contains pointers to functions that must be called when VFS performs an operation on the filesystem object. Nomenclature of these functions is different for different types of objects.

Regular files and character special files (device drivers) have read() and write() functions, directories have lookup() function, etc. For directories and regular files, these functions are provided by the filesystem driver.

It is important to note that to do a file or directory lookup, VFS must have in memory the inode of the parent directory. Having the inode, VFS just calls its lookup() function.

Tree of kernel function calls made during open(2) syscall is presented in Fig. 3.

Inodes and dentries are cached, i.e. they are not deleted after the use. While the file is opened, the kernel must keep its inode in memory. When the file is closed, the inode is kept in the cache until cache manager reaps it. Cache manager uses heuristic strategies to determine which structures to reap, depending on amount of available memory, nature of the object, statistic of the requests to this object and other parameters.

Linux NFS client is a VFS filesystem driver, and follows the VFS semantics. We see one important difference between POSIX and Linux VFS semantics. POSIX standard does not specify how hierarchical file lookups should be made, and does not forbid compound lookups, at least not explicitly. VFS specifies that hierarchical pathname must be split to components and every component must be handled separately.

We did our measurements on Linux 3.10 kernel. In newer kernels, numerous enhancements in VFS and NFS client were made, including support for NFSv4.1 and directory delegations, but this aspect of the VFS API and architecture did not change.

VFS implicitly forbids joining NFS LOOKUP RPC to compounds by two reasons. First, to make a next lookup() call, VFS must have an inode structure from the previous call. So it cannot wait while NFS driver collects several lookup requests and sends them to the server. Second, NFS driver does not know the context of the lookup() call. It does not even know is it the last lookup in the chain. So even if it could collect delayed lookup requests, it could not know when to stop.

We can only speculate why this was overlooked. Indeed, doing all lookups separately simplifies an FS driver, which is generally good. Second, it simplifies handling of some situations, like symlinks and pathnames that cross filesystem boundaries. Third, VFS API mostly stabilized in 1990s, when NFS was at version 3 and compound requests were not available. Fourth, when VFS stabilized, latency to bandwidth ratio in local networks was different, so the advantage of the compound requests was not so significant.

3 PROPOSED CHANGE TO VFS FRAMEWORK

3.1 Chain_lookup

We propose to add a new operation to Linux VFS framework API for directory inodes. We call it chain_lookup(). VFS has concept of optional operations in inode structure. If some operations are not available in the specific driver, VFS can use some generic procedure to implement the required operation. In our case, if the driver does not provide chain_lookup() function, VFS will use standard walk_component() function and lookup() driver call. So our change does not require modification of other filesystem drivers.

It is important to note that Linux VFS already has some NFS-specific modifications and driver functions that implemented only in NFS. Our modification can be useful not only for NFSv4, but for other network file systems that support compound requests (like SMBv2) and for local and virtual file systems that have global directory index, like Reiserfs.

As the name suggests, chain_lookup() accepts the list of dentry structures corresponding to components of parsed pathname. In the case of NFSv4, this function forms compound request, sends it to the server, receives the answer and fills inodes for valid filesystem objects. If some of the pathname components do not exist, corresponding dentry structures are converted to negative dentries.

The general logic of the modified VFS framework remains similar to the original one. The VFS parses the pathname to components, links them to a chain and consults the dentry cache. If the cache record is valid, it removes the corresponding component from the chain and continues until it finds obsolete or nonexistent cache record. Then it passes the rest of the chain to chain_lookup() function.

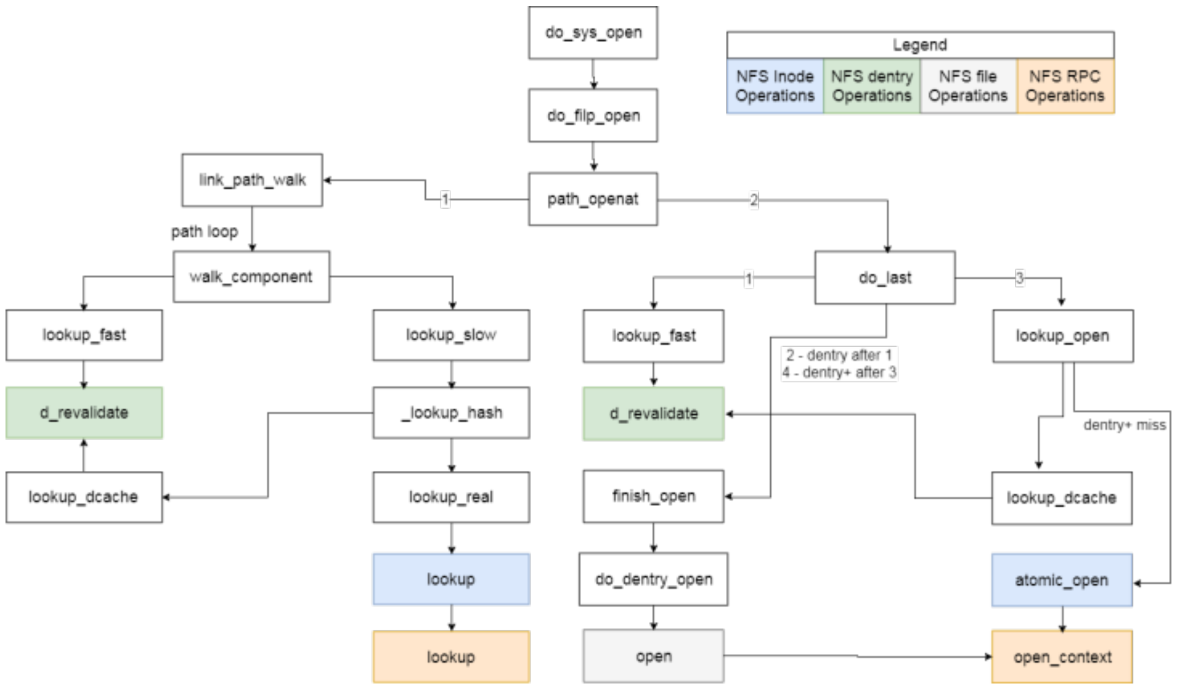

Figure 3: VFS call tree for open(2) syscall (Linux 3.10 kernel)

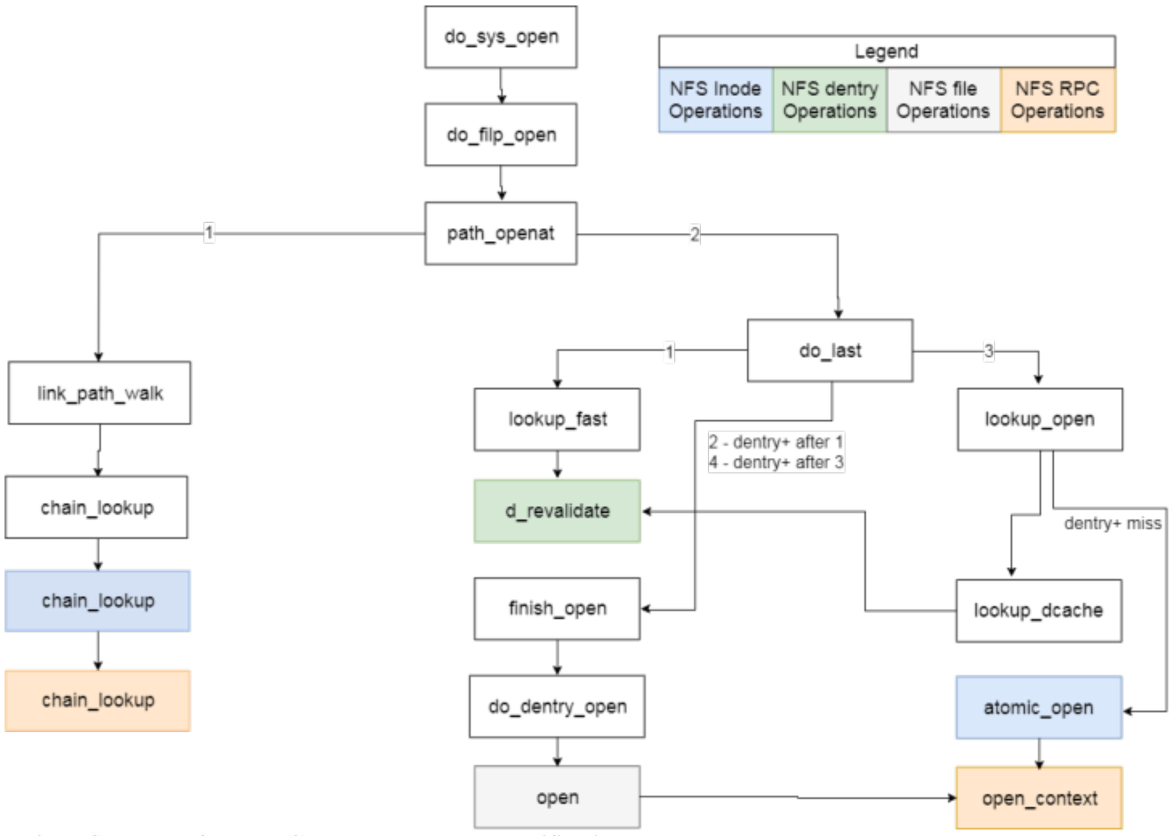

Figure 4: VFS call tree for open(2) syscall, proposed modification

The modified call tree for open(2) is presented in Fig. 4.

According to NFS specification, individual RPCs of the compound request are processed according to their order until some of the calls fails. Then the server returns results of all successful requests and an error code for the failed request. This is exactly the behavior we need for a pathname lookup. If the path component does not exist, its child components cannot exist too.

There are several cases that must be handled to make this work in real-world scenarios. These are: mount points, symlinks, '.' and '..' directories and obsolete cached dentries.

3.1.1 Mount points

In Unix-like systems, filesystems are connected to directory tree by issuing a mount(1M) command or corresponding system call. This command logically attaches the root directory of the target filesystem to some directory of the previously mounted file system. This directory is called a 'mount point'. There is a dedicated "Root" filesystem which is mounted at system boot in unusual way. It has mount point with name '/'.

For all active mount points, VFS creates dentry and inode structures with special attributes. These structures are locked in the cache and never reaped by cache manager. So we must consult the cache, and, if the dentry corresponds to the mount point, we must check for this and return the control to VFS.

It is interesting to note that mount point can be not empty before mounting. The files and subdirectories of the mount point remain on disk but become inaccessible. So it is possible to imagine a scenario when we form a compound lookup and it returns valid inodes for all components, but we must rejects some of these inodes because they are below a mount point. For the driver it means only that it must check every pathname component for being a mount point, not just first and last one.

3.1.2 Symlinks

Symlinks or symbolic links are the files of the special type. Instead of the data blocks, these files contain a text string. This string is interpreted as a (relative or absolute) name of another filesystem object. It is said that symlink points to this object. Symlinks can point to files, directories, any other named filesystem objects and also to names that do not exist. Pointed objects can be placed in the same filesystem as the symlink or in other filesystems.

Because symlinks can point to other filesystems, they cannot be handled on the FS driver level. When finding a symlink, the driver must stop pathname processing and return control to VFS. It is easy to implement, because NFS LOOKUP RPC returns the error when applied to a name relative to the symlink. We just need to properly handle this error.

3.1.3 '.' and '..' directory entries

In POSIX-compliant file systems, every directory must contain two special entries, '.' and '..'. Entry '.' points to the directory itself, and '..' points on its parent directory. On-disc structure of directories in common Unix file systems, such as UFS or ext3/ext4. actually do contain these entries. NFS servers also return valid metadata for these names. On other filesystems, the FS driver must somehow imitate their existence.

The '.' entries in most cases can be removed from the pathname, and this is what we do.

The '..' directory cannot be handled on the filesystem driver level, because it can point above the mount point, to the other filesystem. If it is placed after a symlink, it can point to completely unrelated filesystem, and we cannot know this until we find that this pathname component is a symlink. In current implementation, we never include '..' directories in compound lookups. Instead, we split a pathname to parts before and after a '..' component, and handle them separately. This probably not the best solution from the latency perspective, but this allowed us to implement a correct filesystem behavior.

3.1.4 Obsolete cached dentries

Network file system clients must consider a situation when the filesystem is changed by other client or a process running on the server. This cannot happen with local file systems, because all changes to a local FS come through the same VFS layer and the same cache. For example, a directory corresponding to cached dentry and inode can be renamed or removed.

To deal with this situation, VFS has NFS-specific logic that calls d_revalidate() driver function on dentry cache hits (this call is visible on Fig. 3). This result in validation RPC requests before every LOOKUP RPC. These RPC sequences are noticeable in Appendices 1-3.

For NFSv4 and 4.1, this is redundant, because this protocol version has NFS4ERR_STALE error code signaling that given entity does not exist on the server.

So, instead of validating a cache entry before any access we can try the operation and do a revalidation only after receiving an NFS4ERR_STALE result. In worst case (when we need revalidation), this produces same number or RPC requests as current implementation. In best case, when the revalidation is not necessary, we avoid one RPC compound.

This approach does not work with negative dentries. The VFS never does any file operations on cached negative dentries, so it never can get NFS4ERR_STALE message. But negative dentry can become obsolete.

Consider the following scenario. The web server on a client node checks for existence of the file .htaccess in a directory. VFS gets negative dentry and caches it. Then the other client node creates .htaccess file. Until VFS revalidates negative dentry, the web server will think the file still does not exist.

This sound like the reason for doing negative dentry revalidation on every request. We tried to do this and got a significant performance hit, because real web servers do many failed file lookups per every successful one.

In current implementation, we mark negative dentry with timestamps and revalidate them on a first hit after a given timeout. This approach has obvious drawbacks. Hopefully, directory delegation in NFSv4.1 would allow us to better handle this situation.

3.1.5 Other possible issues

We tested our implementation only against Linux CentOS 7 (kernel version 3.10) server. It is possible that other NFS servers will reject compound lookups, or, worse, compound lookup would trigger previously unknown bugs in the server code. Production-ready implementation must take this to consideration. It must have a mount option that disables new behavior for a specific server or share and reverts to old behavior.

Our implementation currently does not properly handle a situation when the server rejects too long compound. According to the protocol specification, in this case a server could either process part of the compound and return the error on the rest of it (we found that Linux NFS kernel server does this) or reject all compound. First case can be handled by current implementation, we just need to add a proper handling to NFS4ERR_RESOURCE (Error Code 10018) error that must be returned. The second case is more complicated. Probably, the best solution would be to add a mount option setting a limit on compound length for a given mount point.

3.1 Results

Times of open(2) syscall on the modified VFS/NFSv4 show significant improvement relative to stock kernel (see Fig. 5). It can be said that the Fig. 5 graph shows unrealistically deep directory hierarchies that never occur in real life. However, even on 2-5 levels of hierarchy the performance gain is still significant.

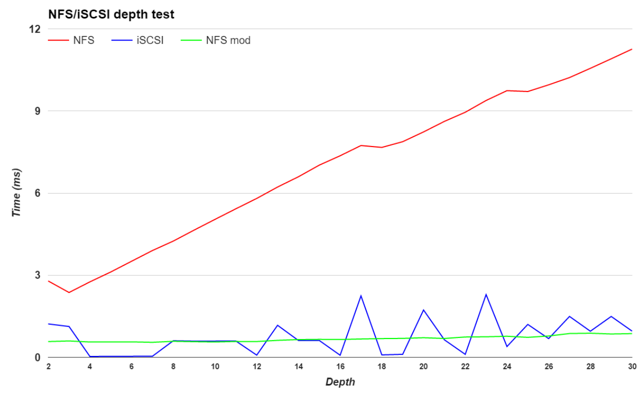

Figure 5: Time of open(2) syscall relative to pathname depth, including data for modified NFS client.

Impact of this change for real-world applications heavily depends on a type of the workload. For applications accessing large files, like virtual machine images or multimedia files, the performance gain might be insignificant. We did the measurement on a web server application that works with small files and takes a significant performance hit on stock NFS client.

For historical reasons, because at the beginning of our research we assumed that NFS is slow because of inefficient caching, we set up a test configuration where we measured performance of a single dynamic web page loaded repeatedly. This workload is not very realistic but provides best conditions for any caching strategy.

The tests were performed on identical machines with the following hardware:

1. Intel Pentium(R) Dual Core CPU E5200@2.50GHz
2. Intel 82566 DC Gigabit NIC
3. Segate Barracuda 7200.10 ST3160815AS
4. 2x1 GB RAM DDR2 800 MHz, swap 2 GB
5. CentoOS 7 (Linux 3.10 kernel)
6. NETGEAR prosafe 16 port Gigabit Switch model JGS516 IEEE 1000BaseT

The workload was Apache2 web server with mod_php, serving Joomla! CMS application. SQL server for Joomla! was installed on the same machine as Apache, and the database was placed on local HDD. PHP, static files and data files of Joomla! were placed on different types of NAS/SAN storage on an identical computer connected to the web server by a single switch. The Apache Jmeter tool running on a third computer was used to repeatedly open a Joomla! start page.

This type of workload is more complex than it seems, because start page of Joomla! is a dynamic web page generated by server-side PHP script. Opening of this page involves opening or checking existence of ~200 files, including PHP libraries, .css, JavaScript and picture files, .htaccess files, etc. Latencies when opening or calling stat(2) on these files have cumulative impact on the page open time.

In table 1 hot-cache sustained speed measurements for different types of storage are presented (first sample taken on cold cache was discarded in all datasets). It can be seen than stock NFS has ~20% performahce hit compared to block storages, and modified version of the NFS is similar to block storage.

Table 1: Joomla! hot-cache page open time on different storage types

Storage	Average time, ms	80% line, ms
iSCSI	111	100
NFS	130	142
NFS loop	115	108
NFS mod	108	117

Another way to produce filesystem workload are filesystem benchmarks. Postmark is a filesystem benchmark developed by NetApp. Unfortunately, original project page on the NetApp website is not available, we used the version downloaded from site [14]. We choose this benchmark because it was used in work [4] and because it could run tests on small (1000 bytes to 20 kb) files, which is one of the worst cases for NFS.

Below are presented the postmark config and the test results for stock and modified NFS client. For both types of the client, the best dataset from three runs was selected.

```
Config.txt:
The base number of files is 10000
Transactions: 100000
Files range between 1000 bytes and 24.41 kilobytes in size
Working directory: /root/postmark-1.51/mnt/postmark
Block sizes are: read=10.00 kilobytes, write=10.00 kilobytes
Biases are: read/append=9, create/delete=-1
Not using Unix buffered file I/O
Random number generator seed is 42
Report format is verbose.
```

```
NFS stock:
Time:
  48 seconds total
  33 seconds of transactions (3030 per second)
Files:
  10000 created (208 per second)
      Creation alone: 10000 files (1000 per second)
      Mixed with transactions: 0 files (0 per second)
  89906 read (2724 per second)
  10089 appended (305 per second)
  10000 deleted (208 per second)
      Deletion alone: 10000 files (2000 per second)
      Mixed with transactions: 0 files (0 per second)
Data:
  1332.15 megabytes read (27.75 megabytes per second)
  169.57 megabytes written (3.53 megabytes per second)
NFS mod:
Time:
      40 seconds total
      25 seconds of transactions (4000 per second)
Files:
      10000 created (250 per second)
              Creation alone: 10000 files (1000 per second)
              Mixed with transactions: 0 files (0 per second)
      89906 read (3596 per second)
      10089 appended (403 per second)
      10000 deleted (250 per second)
              Deletion alone: 10000 files (2000 per second)
              Mixed with transactions: 0 files (0 per second)
Data:
      1332.15 megabytes read (33.30 megabytes per second)
      169.57 megabytes written (4.24 megabytes per second)
```

4 CONCLUSIONS

We identified a source of latency in popular types of NFSv4 clients. Then we proposed a method of removing it by modifying (extending) a VFS API and implemented this modification. The modified version of NFSv4 kernel client for Linux shows significant improvement in the execution time of open(2) system call and in other system calls that work with hierarchical pathnames (stat(2), access(2), etc). Modified version of the client shows measurable improvement on system and application-level benchmarks, especially on workloads using small files.

Our approach can be applied to other Unix-like NFS clients and to other network file system clients for protocols supporting compound requests, such as SMBv2.

Our approach (modifying kernel framework API) has obvious drawbacks, but also obvious advantages compared to alternatives. For example, authors of [5] propose to improve NFS latencies by implementing an userland library for NFS file access. This requires modification of all applications that work with the files. On other hand, change in kernel NFS client would be available to all applications without any change in the application or library code.

A APPENDICES

The following sequences of RPC calls were recorded by rpcdebug(8) utility when different types of Unix-like clients executed the command

cat file_tests/nfscc/tests/multi_test/multi_test.py

Only requests related to first two pathname components are presented.

A.1 Linux 3.10 kernel (CentOS 7)

```
nfsv4 compound op #1/3: 22 (OP_PUTFH)
nfsd: fh_verify(28: 00070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff88007b56f080 opcnt 3 #1: 22: status 0
nfsv4 compound op #2/3: 3 (OP_ACCESS)
nfsd: fh_verify(28: 00070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff88007b56f080 opcnt 3 #2: 3: status 0
nfsv4 compound op #3/3: 9 (OP_GETATTR)
nfsd: fh_verify(28: 00070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff88007b56f080 opcnt 3 #3: 9: status 0
nfsv4 compound returned 0
nfsd_dispatch: vers 4 proc 1
nfsv4 compound op #1/4: 22 (OP_PUTFH)
nfsd: fh_verify(28: 00070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff88007b56f080 opcnt 4 #1: 22: status 0
nfsv4 compound op #2/4: 15 (OP_LOOKUP)
nfsd: fh_verify(28: 00070001 00000087 00000000 03fd0000 00000000
00000000)
nfsd: nfsd_lookup(fh 28: 00070001 00000087 00000000 03fd0000
00000000 00000000, file_tests)
nfsd: fh_compose(exp fd:03/135 share/file_tests, ino=33600139)
nfsv4 compound op ffff88007b56f080 opcnt 4 #2: 15: status 0
nfsv4 compound op #3/4: 10 (OP_GETFH)
nfsv4 compound op ffff88007b56f080 opcnt 4 #3: 10: status 0
nfsv4 compound op #4/4: 9 (OP_GETATTR)
[12355023.270083] nfsd: fh_verify(40: 81070001 00000087 00000000
03fd0000 00000000 00000000)
nfsv4 compound op ffff88007b56f080 opcnt 4 #4: 9: status 0
nfsv4 compound returned 0
nfsd_dispatch: vers 4 proc 1
nfsv4 compound op #1/3: 22 (OP_PUTFH)
nfsd: fh_verify(40: 81070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff88007b56f080 opcnt 3 #1: 22: status 0
nfsv4 compound op #2/3: 3 (OP_ACCESS)
fh_verify(40: 81070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff88007b56f080 opcnt 3 #2: 3: status 0
nfsv4 compound op #3/3: 9 (OP_GETATTR)
nfsd: fh_verify(40: 81070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff88007b56f080 opcnt 3 #3: 9: status 0
nfsv4 compound returned 0
nfsd_dispatch: vers 4 proc 1
nfsv4 compound op #1/4: 22 (OP_PUTFH)
```

```
nfsd: fh_verify(40: 81070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff880035990080 opcnt 4 #1: 22: status 0
nfsv4 compound op #2/4: 15 (OP_LOOKUP)
nfsd: fh_verify(40: 81070001 00000087 00000000 03fd0000 00000000
00000000)
nfsd: nfsd_lookup(fh 40: 81070001 00000087 00000000 03fd0000
00000000 00000000, nfscc)
nfsd: fh_compose(exp fd:03/135 file_tests/nfscc, ino=67191807)
nfsv4 compound op ffff880035990080 opcnt 4 #2: 15: status 0
nfsv4 compound op #3/4: 10 (OP_GETFH)
nfsv4 compound op ffff880035990080 opcnt 4 #3: 10: status 0
nfsv4 compound op #4/4: 9 (OP_GETATTR)
nfsd: fh_verify(40: 81070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff880035990080 opcnt 4 #4: 9: status 0
nfsv4 compound returned 0
```

A.2 Solaris (OpenIndiana 5.11)

```
nfsv4 compound op #1/9: 22 (OP_PUTFH)
nfsd: fh_verify(28: 00070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff88001dcab000 opcnt 9 #1: 22: status 0
nfsv4 compound op #2/9: 32 (OP_SAVEFH)
nfsv4 compound op ffff88001dcab000 opcnt 9 #2: 32: status 0
nfsv4 compound op #3/9: 15 (OP_LOOKUP)
nfsd: fh_verify(28: 00070001 00000087 00000000 03fd0000 00000000
00000000)
nfsd: nfsd_lookup(fh 28: 00070001 00000087 00000000 03fd0000
00000000 00000000, file_tests)
nfsd: fh_compose(exp fd:03/135 share/file_tests, ino=33600139)
nfsv4 compound op ffff88001dcab000 opcnt 9 #3: 15: status 0
nfsv4 compound op #4/9: 10 (OP_GETFH)
nfsv4 compound op ffff88001dcab000 opcnt 9 #4: 10: status 0
nfsv4 compound op #5/9: 9 (OP_GETATTR)
nfsd: fh_verify(40: 81070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff88001dcab000 opcnt 9 #5: 9: status 0
nfsv4 compound op #6/9: 31 (OP_RESTOREFH)
nfsv4 compound op ffff88001dcab000 opcnt 9 #6: 31: status 0
nfsv4 compound op #7/9: 17 (OP_NVERIFY)
nfsd: fh_verify(28: 00070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff88001dcab000 opcnt 9 #7: 17: status 0
nfsv4 compound op #8/9: 9 (OP_GETATTR)
nfsd: fh_verify(28: 00070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff88001dcab000 opcnt 9 #8: 9: status 0
nfsv4 compound op #9/9: 3 (OP_ACCESS)
nfsd: fh_verify(28: 00070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff88001dcab000 opcnt 9 #9: 3: status 0
nfsv4 compound returned 0
nfsd_dispatch: vers 4 proc 1
nfsv4 compound op #1/9: 22 (OP_PUTFH)
nfsd: fh_verify(40: 81070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff88001dcab000 opcnt 9 #1: 22: status 0
nfsv4 compound op #2/9: 32 (OP_SAVEFH)
nfsv4 compound op ffff88001dcab000 opcnt 9 #2: 32: status 0
nfsv4 compound op #3/9: 15 (OP_LOOKUP)
nfsd: fh_verify(40: 81070001 00000087 00000000 03fd0000 00000000
00000000)
nfsd: nfsd_lookup(fh 40: 81070001 00000087 00000000 03fd0000
00000000 00000000, nfscc)
nfsd: fh_compose(exp fd:03/135 file_tests/nfscc, ino=67191807)
nfsv4 compound op ffff88001dcab000 opcnt 9 #3: 15: status 0
nfsv4 compound op #4/9: 10 (OP_GETFH)
nfsv4 compound op ffff88001dcab000 opcnt 9 #4: 10: status 0
nfsv4 compound op #5/9: 9 (OP_GETATTR)
nfsd: fh_verify(40: 81070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff88001dcab000 opcnt 9 #5: 9: status 0
nfsv4 compound op #6/9: 31 (OP_RESTOREFH)
nfsv4 compound op ffff88001dcab000 opcnt 9 #6: 31: status 0
nfsv4 compound op #7/9: 17 (OP_NVERIFY)
nfsd: fh_verify(40: 81070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff88001dcab000 opcnt 9 #7: 17: status 10009
nfsv4 compound returned 10009
nfsd_dispatch: vers 4 proc 1
nfsv4 compound op #1/3: 22 (OP_PUTFH)
nfsd: fh_verify(40: 81070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff88007b56f080 opcnt 3 #1: 22: status 0
nfsv4 compound op #2/3: 3 (OP_ACCESS)
nfsd: fh_verify(40: 81070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff88007b56f080 opcnt 3 #2: 3: status 0
nfsv4 compound op #3/3: 9 (OP_GETATTR)
nfsd: fh_verify(40: 81070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff88007b56f080 opcnt 3 #3: 9: status 0
nfsv4 compound returned 0nfsv4 compound returned 0
```

A.3 FreeBSD 11.0-RELEASE-p1

```
nfsv4 compound op #1/5: 22 (OP_PUTFH)
nfsd: fh_verify(28: 00070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff880035990080 opcnt 5 #1: 22: status 0
nfsv4 compound op #2/5: 9 (OP_GETATTR)
nfsd: fh_verify(28: 00070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff880035990080 opcnt 5 #2: 9: status 0
nfsv4 compound op #3/5: 15 (OP_LOOKUP)
nfsd: fh_verify(28: 00070001 00000087 00000000 03fd0000 00000000
00000000)
nfsd: nfsd_lookup(fh 28: 00070001 00000087 00000000 03fd0000
00000000 00000000, file_tests)
nfsd: fh_compose(exp fd:03/135 share/file_tests, ino=33600139)
nfsv4 compound op ffff880035990080 opcnt 5 #3: 15: status 0
nfsv4 compound op #4/5: 10 (OP_GETFH)
nfsv4 compound op ffff880035990080 opcnt 5 #4: 10: status 0
nfsv4 compound op #5/5: 9 (OP_GETATTR)
nfsd: fh_verify(40: 81070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff880035990080 opcnt 5 #5: 9: status 0
nfsv4 compound returned 0
nfsd_dispatch: vers 4 proc 1
nfsv4 compound op #1/3: 22 (OP_PUTFH)
nfsd: fh_verify(40: 81070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff880035990080 opcnt 3 #1: 22: status 0
nfsv4 compound op #2/3: 3 (OP_ACCESS)
nfsd: fh_verify(40: 81070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff880035990080 opcnt 3 #2: 3: status 0
nfsv4 compound op #3/3: 9 (OP_GETATTR)
nfsd: fh_verify(40: 81070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff880035990080 opcnt 3 #3: 9: status 0
nfsv4 compound returned 0
nfsd_dispatch: vers 4 proc 1
nfsv4 compound op #1/5: 22 (OP_PUTFH)
nfsd: fh_verify(40: 81070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff880035990080 opcnt 5 #1: 22: status 0
nfsv4 compound op #2/5: 9 (OP_GETATTR)
nfsd: fh_verify(40: 81070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff880035990080 opcnt 5 #2: 9: status 0
nfsv4 compound op #3/5: 15 (OP_LOOKUP)
nfsd: fh_verify(40: 81070001 00000087 00000000 03fd0000 00000000
00000000)
nfsd: nfsd_lookup(fh 40: 81070001 00000087 00000000 03fd0000
00000000 00000000, nfscc)
nfsd: fh_compose(exp fd:03/135 file_tests/nfscc, ino=67191807)
nfsv4 compound op ffff880035990080 opcnt 5 #3: 15: status 0
nfsv4 compound op #4/5: 10 (OP_GETFH)
nfsv4 compound op ffff880035990080 opcnt 5 #4: 10: status 0
nfsv4 compound op #5/5: 9 (OP_GETATTR)
nfsd: fh_verify(40: 81070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff880035990080 opcnt 5 #5: 9: status 0
nfsv4 compound returned 0
nfsd_dispatch: vers 4 proc 1
nfsv4 compound op #1/3: 22 (OP_PUTFH)
nfsd: fh_verify(40: 81070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff88007b56f080 opcnt 3 #1: 22: status 0
nfsv4 compound op #2/3: 3 (OP_ACCESS)
nfsd: fh_verify(40: 81070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff88007b56f080 opcnt 3 #2: 3: status 0
compound op #3/3: 9 (OP_GETATTR)
nfsd: fh_verify(40: 81070001 00000087 00000000 03fd0000 00000000
00000000)
nfsv4 compound op ffff88007b56f080 opcnt 3 #3: 9: status 0
compound returned 0
```

REFERENCES

[1] Dmitry Irtegov, Igor Knyazev, Julia Mallaeva, Sergey Oleynikov, Michael Rootman, and Dmitry Solovyev. 2014. About one approach to building low latency network file system. In Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia (CEE-SECR '14). ACM, New York, NY, USA,, Article 2, 9 pages. DOI=10.1145/2687233.2687248

[2] Sage A. Weil. Ceph: reliable, scalable, and high-performance distributed storage, Doctoral Dissertation, University of California at Santa Cruz Santa Cruz, CA, USA, 2007

[3] Torben P. Lustre File System: Demo Quick Start Guide, Lustre Group, 2009

[4] Peter Radkov, Li Yin, Pawan Goyal, Prasenjit Sarkar, Prashant Shenoy, A Performance Comparison of NFS and iSCSI for IP-Networked Storage, Proceedings of the 3rd USENIX Conference on File and Storage Technologies, March 31-31, 2004, San Francisco, CA

[5] Chen, M., Hildebrand, D., Nelson, H., Saluja, J., Subramony, A. S. H., & Zadok, E. (2017, February). vNFS: Maximizing NFS Performance with Compounds and Vectorized I/O. In FAST (pp. 301-314).

[6] Haynes, Tom, and David Noveck. "Network File System (NFS) version 4 Protocol." RFC 7530 (2015).

[7] Shepler, Spencer, David Noveck, and Mike Eisler. "NFS version 4 minor version 1." RFC 5661 (2010).

[8] Chen, Ming, et al. "Newer is sometimes better: An evaluation of NFSv4. 1." ACM SIGMETRICS Performance Evaluation Review. Vol. 43. No. 1. ACM, 2015.

[9] Satran, Julian, and Kalman Meth. "Internet small computer systems interface (iSCSI)." RFC 3720 (2004).

[10] Project OCFS2, General-purpose cluster file system. https://oss.oracle.com/projects/ocfs2/

[11] A. Tannenbaum, M. Van Steen, 2006. Distributed Systems: Principles and Paradigms (2nd Edition), Prentice-Hall, Inc. Upper Saddle River, NJ, USA.

[12] IEEE Std 1003.1, 2016 Edition

[13] Gooch, Richard and Engberg, Pekka, Overview of the Linux Virtual File System, https://www.kernel.org/doc/Documentation/filesystems/vfs.txt

[14] PostMark Test Profile: https://openbenchmarking.org/test/pts/postmark