Network File System (NFS)

Vedant Acharya (23110010), Haarit Chavda (23110077),
Harshil Shah (23110132), Jeet Joshi (23110148), Akshat Shah (23110293)
Advisor - Naveen Tiwari 24210068

Introduction

In modern computing systems, sharing data seamlessly across multiple machines is a basic requirement. Network File System (NFS) is one of the earliest and most popular distributed file systems that provides transparent access to remote files over a network as if they were local. Originally developed by Sun Microsystems in the 1980s, NFS introduced the concept of a client–server file system in which clients can mount remote directories on the server and perform standard file operations such as open, read, write, and close, using the familiar POSIX interface. This abstraction allows users and applications to access files on remote servers without being aware of their physical location or storage details.

The motivation behind this project was to understand the working principles of NFS by implementing a simplified version of it. The main goals were to design a **request–response** communication protocol between client and server, implement efficient **client-side caching** mechanisms, and enable concurrent request processing on the server side to handle multiple clients simultaneously. The project implements the client side using the **Filesystem in Userspace (FUSE)** framework. This allows users to execute terminal commands such as **cat, touch, nano, cp**, and **mv** directly on the mounted NFS directory, while internally routing all these operations through our custom-built NFS client and server layers.

The designed NFS operates using an opcode-based request–response structure, where each filesystem operation corresponds to a specific opcode. The client constructs a request message containing the opcode and relevant fields depending on the operation. For example, in a READ operation, the request includes the file descriptor, offset, and data length. The client then sends this request to the server over a TCP socket. The server decodes the request, performs the required operation on its local filesystem based on the opcode, and sends a structured response back to the client containing the status and data (for a read).

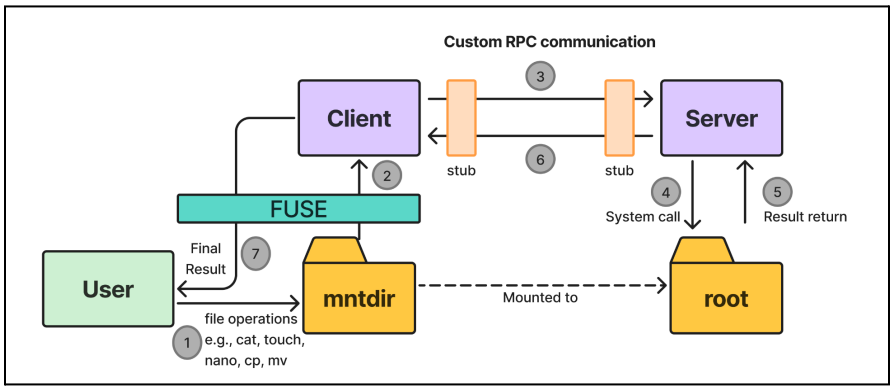


Figure 1: System Architecture

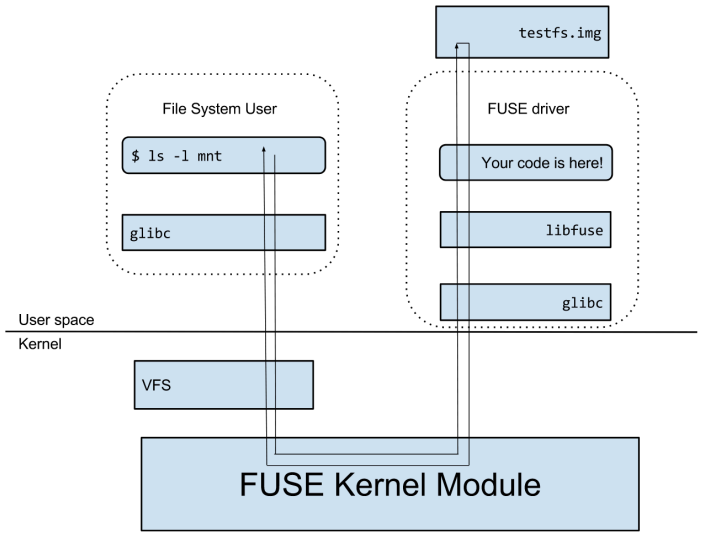
By combining socket programming, concurrency, and FUSE integration, the project provides a hands-on understanding of the challenges involved in achieving performance, consistency, and transparency in network-based file access. The following sections describe the overall architecture, implementation of the client and server modules, and the performance evaluation of the system.

System Architecture

Figure 1 shows the overall system architecture of our simplified version of NFS, called LNFS. First, the user mounts the **mntdir** to the server's **root** directory using the FUSE library. Every filesystem call made within this mounted directory is routed from FUSE to our client.cpp program, which acts as a client stub. It serializes and sends the request structure to the server over a TCP socket based on our custom RPC design. The server.cpp stub receives the request, deserializes it, identifies the opcode, and executes the corresponding system call on its local root directory. The result is then serialized into a structured response and sent back to the client. The client.cpp program deserializes the response and forwards it to FUSE, which finally presents the result to the user. The following sections explain the working of FUSE and the request–response structure of the LNFS protocol in detail.

A. What is Fuse and how does it work??

Fuse 3 (often written as FUSE3 or libfuse3) is the current major version of Filesystem in Userspace, a Linux interface that lets user-space programs implement filesystems and mount them without writing kernel code. It consists of a kernel module plus a user-space library (libfuse3) and tools like fusermount3 and mount.fuse3 to manage FUSE mounts.



https://notes.yxy.ninja/OS/File-System/FUSE

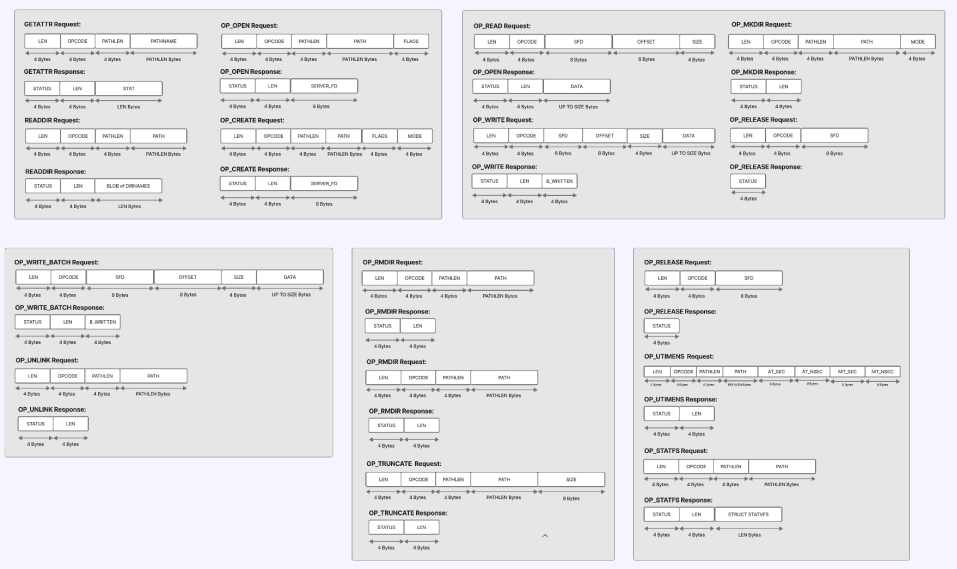
Applications make system calls like open, read, write, getattr, and readdir on paths that are mounted using FUSE. The Linux VFS recognizes the mount and hands those operations to the FUSE kernel module. The FUSE kernel module serializes each request and places it on a queue exposed via the character device /dev/fuse for a user-space daemon to handle, then sends the daemon's reply back up to the application.

B. Request-response structure

We referred to the NFSv2 RFC-1094 for designing our request and response protocols. We haven't kept the same structures but made them simpler, retaining 13 out of 17 operations that are sufficient and necessary for NFS to work with FUSE. The following table shows each operation name, its opcode, and the fields included in the request and response structures.

Operation	**Op code**	**Description**	**Request Payload**	**Response Payload**
OP_GETATTR	1	Get file attributes (metadata)	Pathname	stat structure (attributes)
OP_READDIR	2	Read the contents of a directory	Pathname	Blob of directory entries
OP_OPEN	3	Open a file	Pathname, Flags	File descriptor (FD)
OP_READ	4	Read data from a file	FD, Offset, Size	Data
OP_WRITE	5	Write data to a file	FD, Offset, Size, Data	Bytes written
OP_CREATE	6	Create a new file	Pathname, Flags, Mode	File descriptor (FD)
OP_UNLINK	7	Remove a file	Pathname	Empty
OP_MKDIR	8	Create a new directory	Pathname	Empty
OP_RMDIR	9	Remove a directory	Pathname	Empty
OP_TRUNCATE	10	Truncate or resize a file	Pathname, Size	Empty
OP_UTIMENS	11	Update file timestamps	Pathname, Access time (s and ns), Modified Time (s and ns)	Empty
OP_STATFS	12	Get filesystem statistics	Pathname	statvfs structure
OP_RELEASE	13	Close a file and release resources	File Descriptor (FD)	Empty
OP_WRITE_BATCH	14	Perform batched write operations	FD, Offset, Size, Data	Bytes written

To send a request, the client first sends the total length (payload length + 4 bytes for the opcode) to the server. The server then creates a buffer of that size. Next, the client sends the opcode followed by the payload. The server processes this request and, based on the opcode, calls the respective POSIX system call and first sends back a status code, based on which the client determines whether the request succeeded or failed. If the request is successful, the client then reads the response payload size sent by the server, allocates a buffer accordingly, and finally receives the result data it requested. The image below shows the detailed field structure along with its sizes for each operation.



Link for high-resolution PDF - LNFS.pdf

Design Choices

A. Chunk Size For Large Files

Sending very large files in a single transfer is inefficient. It can lead to several issues, including high memory usage, increased risk of network timeouts or failures, and longer delays that impact overall system responsiveness. To address these concerns, we break down large files into smaller, manageable chunks.

It also improves network utilization and allows parallel processing of chunks, reducing server load and preventing bottlenecks. Choosing an appropriate chunk size balances between minimizing overhead (too small chunks increase management cost) and avoiding resource exhaustion (too large chunks strain memory and bandwidth).

B. Caching for Files - LRU

File data is cached on the client in fixed-size blocks, managed using a Least Recently Used (LRU) replacement policy. When the cache reaches its maximum capacity, the least recently accessed blocks are evicted to make room for new data, ensuring that frequently or recently accessed blocks remain available in memory.

Advantages

1. Improves performance by keeping frequently or recently accessed file blocks readily available, reducing access times and network requests.
2. Efficiently manages limited cache memory, automatically discarding data that is least likely to be used again.
3. Particularly effective for workloads with strong temporal locality, such as sequential or re-access patterns.

Disadvantages

1. Cache effectiveness drops for access patterns that are random or larger than the available cache size, leading to frequent evictions and cache misses.
2. Requires additional client memory, which might be limited on some devices.
3. LRU management logic adds some implementation complexity and computational overhead.

C. Caching For Get Attr

To minimize repetitive overhead, file attribute requests (such as size, permissions, and modification times) are cached on the client. Since the operating system frequently queries file metadata—often through multiple getattr calls on the same open file - caching these responses reduces the need for repeated network communication with the server, resulting in lower latency and improved system performance.

D. Logic For Write-Back

On the client side, each open file is assigned its own write buffer. When a write request is made, the system checks whether the new data is sequential to what is already stored in the buffer. If the request is sequential, the data is appended directly to the buffer. If it is not, the buffer is flushed to the server to maintain consistency, and the new data then starts a new buffer sequence. Whenever the size of the buffer exceeds a predetermined threshold, the buffer is automatically flushed, sending all accumulated changes to the server in a single batch operation for efficiency.

Advantages

1. Significantly reduces the frequency of network transmission by batching multiple small writes, resulting in higher throughput and lower latency.
2. Improves overall application performance, since write operations are confirmed as soon as they are stored in the local cache and not delayed by network round-trips.

Disadvantages

1. The data in the buffer may be lost if the client fails.

E. Multiple Client Side Connections

On the client side, a connection pool is established whose size is determined by a global constant (e.g., POOL_SIZE). During initialization, many persistent network connections are opened to the server and stored in a shared list. Each time the send_and_recv() routine is invoked—for example, via FUSE

operations such as read or write—it searches for an available (unused) socket in the pool and allocates it for network communication. This mechanism resembles a producer-consumer setup: after the transaction completes, the socket is marked as available and returned to the pool for subsequent use.

By enabling multiple parallel connections to the server, this approach eliminates bottlenecks inherent to classic single-connection blocking communication, allowing concurrent FUSE operations and enhancing the system's overall throughput and responsiveness.

F. Logic for Read Ahead

The client implements a read-ahead strategy to enhance performance during file reads. When a read request is made, the system first checks if the needed data is present in the local cache. If it is missing, the client fetches not only the required data from the server but also adjacent data blocks, anticipating that future reads will likely request contiguous data. This additional data is proactively stored in the cache based on the assumption of sequential access patterns.

Advantages

1. Reduces latency for sequential reads by making anticipated data immediately available from the cache.
2. Decreases the number of network requests for subsequent reads, thereby improving throughput and overall read performance, especially under high-latency network conditions.
3. Benefits workloads with heavy streaming or sequential I/O, allowing applications to retrieve large blocks of data efficiently.

Disadvantages

1. Ineffective or wasteful for highly random access patterns, as prefetched data may never be used, leading to cache pollution and unnecessary network usage.
2. Increased memory consumption on the client, since extra data is retained in the cache even if it's not accessed.
3. Risk of evicting useful cached data if the read-ahead window is set too aggressively, which can result in more cache misses for other files.

G. Server Side Thread Pool

The server uses a queue-based thread pooling mechanism to achieve multi-threading efficiently. A fixed number of worker threads are created at initialization of the server. When any request is made to the server, the underlying task of the request is placed into a shared queue from which worker threads fetch and execute tasks concurrently. Each thread continuously waits for new tasks and processes them until the pool is stopped.

Advantages

1. Enables concurrent task execution, improving throughput and responsiveness.
2. Reuses worker threads, avoiding the overhead of frequent thread creation
3. Provide controlled concurrency
4. Ensure thread-safe task handling using mutexes and conditional variables

Disadvantages

1. Requires careful synchronization to avoid a race condition
2. Long-running tasks can block threads, causing load imbalance
3. If the queue grows large, new tasks may experience reduced responsiveness and increased latency.

H. Reader-Writer Lock

The server employs a reader-writer locking mechanism to manage concurrent access to files safely and efficiently. This mechanism ensures that multiple clients can read from the same file simultaneously, while write operations remain strictly exclusive. It maintains data consistency and prevents race conditions when multiple threads try to access or modify shared files.

In this approach, each file or directory in the system is associated with its own logical lock. When a read operation is performed, a shared lock is acquired, allowing other reads to proceed concurrently. However, when a write operation occurs, an exclusive lock is obtained, ensuring that no other read or write operations can occur on that file until the modification is complete. This design allows high parallelism for read-heavy workloads while maintaining strict control during write operations.

Advantages

1. Enables multiple concurrent read operations on the same file without interference.
2. Ensures exclusive access during write operations, preserving file integrity.
3. Provides fine-grained synchronization by locking only the files being accessed rather than the entire system.
4. Prevents data inconsistency and corruption caused by simultaneous writes or read-write conflicts.
5. Integrates seamlessly with the server's multi-threaded request handling, maintaining safe parallel execution.

Disadvantages

1. Requires careful coordination to avoid deadlocks between read and write requests.
2. Introduces additional locking overhead, especially under frequent write contention.
3. The lock management structure may grow if many unique files are accessed concurrently.
4. Slightly increases complexity compared to a single global lock due to per-file synchronization handling.

Results and Benchmarking

The following shows a read and write throughput comparison between our and different Linux NFS versions. We measure read and write throughput using the fio command for a 50 MB file.

NFS Version	Read throughput (KiB/s)	Write throughput (KiB/s)
NFSv3	6603	1535
NFSv4	8878	1506
LNFSv1 (ours)	2580	1270

NFSv4 delivers the highest read throughput (8878 KiB/s), followed by NFSv3 (6603 KiB/s), while our LNFSv1 achieves significantly lower read performance (2580 KiB/s),Write throughput shows less variation across versions, with all around 1.2 - 1.5 MiB/s, indicating similar write efficiency. Overall, LNFSv1 lags behind standard NFS versions, particularly in read performance.

The lower read throughput in LNFSv1 can be attributed to several factors:

1. **User-space implementation:** Our design uses the FUSE framework, which introduces additional context switches between kernel and user space, increasing latency.
2. **Simplified RPC design:** Unlike optimized RPC mechanisms in NFSv3 and NFSv4, our custom TCP-based request–response protocol lacks pipelining and batching, reducing parallelism.
3. **Limited caching optimization:** Although LNFSv1 includes read-ahead and write-back caching, these mechanisms are simpler and less adaptive than those in production NFS implementations.

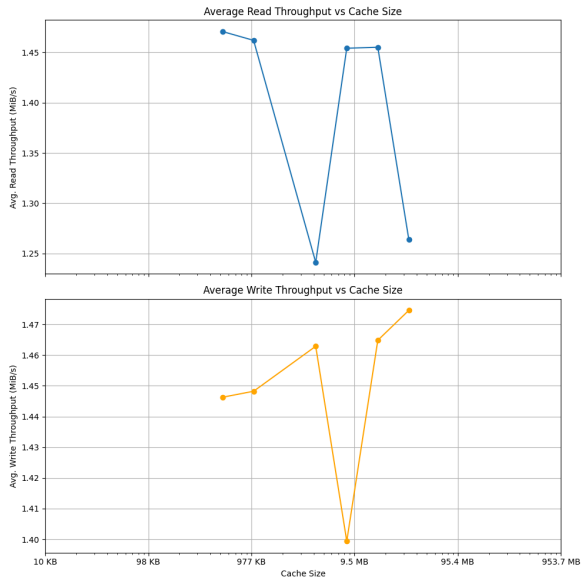


Fig 01

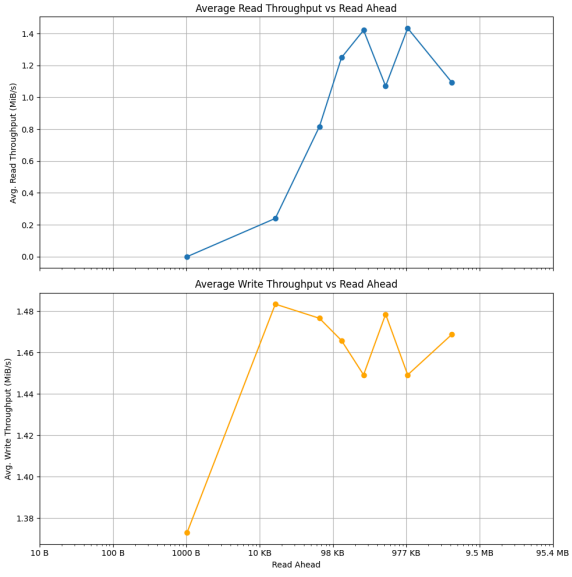


Fig 02

Average Throughput vs Read Ahead Size (Fig 2)
Increasing the read-ahead size allows more of the file to be cached in the local filesystem's cache, leading to more frequent cache hits. As a result, the average read throughput increases with a larger read-ahead size, eventually reaching a saturation point limited by the network bandwidth. Conversely, write throughput is mostly unaffected by read-ahead size, showing only minor fluctuations (0.1–0.3 KiB/s) caused by network conditions.

Average Throughput vs cache size (Fig 1)
Increasing the cache size has little effect on read throughput, which remains nearly constant with variations around ±0.2 KiB/s. This is expected because the test uses a fixed 10 MiB file size, which fits within the cache. Also, these metrics depend on how FIO makes the read and write calls.

This behavior aligns with known effects of read-ahead optimization that preloads data to improve sequential read performance, while write operations are independent of this parameter and primarily influenced by other factors