TFS: A RDMA-based high performance networked filesystem

Tarushii Goel
tarushii@mit.edu

1 Introduction

Networked filesystems allow multiple clients to access shared storage over a network, enabling data sharing and collaboration across distributed systems. Traditional networked filesystems like NFS and SMB use TCP/IP for communication, which introduces significant CPU overhead and latency due to data copying and protocol processing. Modern high-performance networks like Infiniband offer Remote Direct Memory Access (RDMA), allowing direct memory access between machines without CPU involvement, presenting an opportunity to build more efficient distributed storage systems.

TFS is inspired by the design of 3FS [2], a high-performance distributed file system that uses RDMA for inter-node communication, and Google's GFS [1], which pioneered the use of chain replication for distributed storage.

TFS is particularly well-suited for read-heavy workloads where low latency and high throughput is critical, such as:

- **Scientific Computing**: Large-scale data analysis and visualization where researchers need fast access to shared datasets
- **Media Streaming**: Content delivery networks that serve large files to many concurrent readers
- **Database Backups**: Storing and retrieving database snapshots where read performance is crucial

2 Architecture

2.1 Overview

TFS consists of three main components:

1. **Metadata Service**: Implemented using ZooKeeper, maintains file system metadata including inode information and chain state.
2. **Chunkservers**: Serve get and put requests for file chunks. Chunks are stored as files on disk.
3. **Clients**: Interact with both the metadata service and chunkservers to perform file system operations, sending write requests to the HEAD node and read requests to any node in the chain. There are two client implementations: a lower-level RPC-based interface and a higher-level FUSE filesystem client.

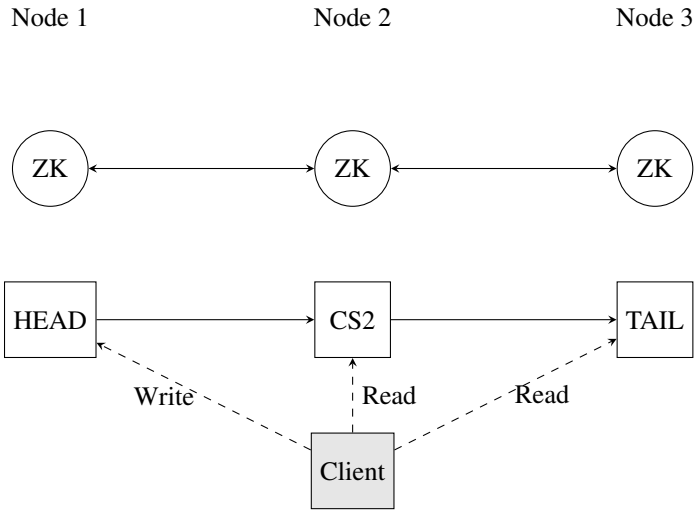

Figure 1: TFS Architecture

2.2 Chain Replication

TFS employs a chain replication architecture with n nodes, each running a ZooKeeper instance for metadata management and a chunkserver process for data storage. Write requests are directed to the HEAD node, which propagates updates through the chain, while read requests can be served by any

node. This design allows reads to be parallelized across nodes, improving read scalability.

In chain replication, write requests follow a strict sequence:

1. The client sends a write request to the HEAD node
2. The HEAD node applies the update and forwards it to the next node in the chain
3. Each subsequent node applies the update and forwards it until it reaches the TAIL
4. The TAIL node commits the update and sends an acknowledgment back up the chain, each node committing the update in reverse order until the HEAD node receives the acknowledgment.

The implementation details and explanation of consistency guarantees are discussed in the CRAQ paper [3], which this design is based on.

This design provides several benefits:

- **Strong Consistency**: Since writes are only acknowledged after reaching the TAIL, all nodes have the same data
- **Read Scalability**: Read requests can be served by any node, distributing the read load
- **Fault Tolerance**: If a node fails, the chain can be reconfigured to bypass it

2.2.1 Chain Configuration

The ZooKeeper metadata service maintains the chain configuration. Each chunkserver opens a session with ZooKeeper and creates a ephemeral file indicating its location in the chain. When the chunkserver is disconnected, the session is closed and the file is deleted. The chunkserver uses a heartbeat/lease mechanism to keep the session alive. If the current lease expires before a new lease is granted, the chunkserver assumes that it has been removed from the chain and stops serving GET/PUT requests.

2.3 RDMA

TFS uses RDMA for inter-node communication. Each node is equipped with a 12 GB/s Infiniband socket, enabling high-throughput, low-latency communication between nodes. Each chunkserver establishes RDMA sessions with its neighboring nodes in the chain and with clients.

2.3.1 GET Operations

For GET operations, clients first register a memory buffer with their local NIC and obtain its address and access key. When requesting data, the client sends a GET request (over TCP/IP) to the server containing this address and key. The server then performs an RDMA write directly into the client's registered memory buffer, bypassing the CPU and operating system on both ends. This zero-copy approach significantly reduces latency and CPU overhead compared to traditional network I/O.

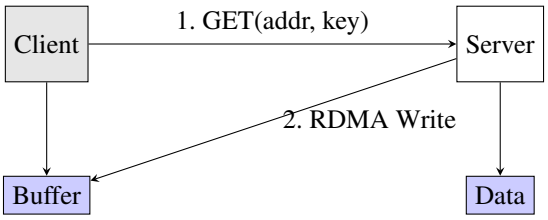

Figure 2: GET Operation Flow

It is possible that the request arrives while a PUT is in progress. To ensure consistency, the server will respond immediately with a STALE response. The client will then retry the GET request at the tail node.

2.3.2 PUT Operations

PUT operations in TFS are implemented as a series of RDMA reads through the chain. The client first registers the data to be written with its local NIC and sends a PUT request to the HEAD node containing the data's address and access key. The HEAD node then performs an RDMA read from the client's memory, followed by RDMA reads between each node in the chain until the data reaches the TAIL, where it is guaranteed to be succeed, because the TAIL is aware of all committed operations.

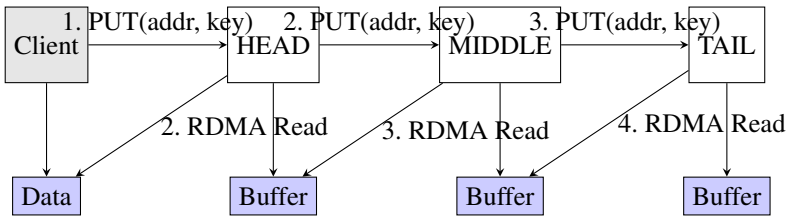

Figure 3: PUT Operation Flow

This design ensures that data flows efficiently through the chain while maintaining strong consistency. Each node in the chain performs an RDMA read from its predecessor, minimizing CPU involvement and network overhead. The TAIL node commits the write only after successfully receiving all data, ensuring that the entire chain has the same consistent state.

2.4 Filesystem Format

Each file is stored as a sequence of chunks. Each chunk is 10 MB in size. Each chunk has a file ID (which is also the inode

number) that indicates which file it belongs to, and a chunk ID that indicates its position in the file. Filesystem lookups involve determining the chunk ID from the offset into the file that the client is requesting, and sending a request with the appropriate file and chunk IDs. The filesystem format is a simple flat filesystem with a single directory. Multiple directories is an additional feature that could be easily added, but that I did not implement due to lack of interest.

3 Results

Throughput: I am able to fully saturate the 12 GB/s Infiniband socket. My simple test sends 1000 concurrent GET requests of 10 MB chunks, and receives the data in 0.85 seconds. For reference, sending the same amount of data over TCP/IP took 2.8 seconds on the same pair of machines with the same network configuration. This demonstrates the power of RDMA and cutting out the TCP/IP headers and data copying.

Latency: On the same 1000 request stress-test, the average latency of a GET request is 0.4 seconds. This is far below my initial goals of sub-10 milliseconds. When there is no server pressure, the request latency normalizes to 1-2 milliseconds. Since the requests are sent over TCP/IP, and only the actual chunk data is sent over RDMA, there is little scope for sub-millisecond latency. However, the design could be improved so that the latency does not vary as drastically with server pressure.

4 Code

The code is available at https://github.com/2022tgoel/TFS.

References

[1] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 20–43. ACM, 2003.

[2] DeepSeek Team. Fire-flyer file system. URL: https://github.com/deepseek-ai/3FS.

[3] Jeff Terrace and Michael J Freedman. Object storage on craq: High-throughput chain replication for read-mostly workloads. In *2009 USENIX Annual Technical Conference (USENIX ATC 09)*, San Diego, CA, jun 2009. USENIX Association. URL: https://www.usenix.org/conference/usenix-09/object-storage-craq-high-throughput-chain-replication-read-mostly-workloads.