Network Distributed File System in User Space

Ivan Voras, Mario Žagar
Faculty of Electrical Engineering & Computing, University of Zagreb,
Unska 3, 10000 Zagreb, Croatia
{ivan.voras, mario.zagar}@fer.hr

Abstract: *File systems have traditionally been implemented in the operating system's kernel to ensure maximum possible speed and integration with the rest of the operating system, and this was true even for network file systems such as NFS. However, available CPU power on mainstream architectures continues to increase daily at a rate which is not closely followed by speed of computer network equipment. When considering development of network-distributed file systems today it becomes clear that speed improvements offered by pure kernel-side implementations are no longer significant given the bandwidth and latencies of computer networks. Recent efforts in enabling user-space file system implementations on free / open source Unix-like operating systems have made it possible to create a solution for distributing file system data over computer networks entirely in user-space. In this work we present such a solution – the Trivially Distributed File System.*

Keywords: file systems, distributed computing, network file systems, file systems in user space, storage

1. Introduction

Management of computer data as files in a suitable file system is, next to data processing, the most important task of a computer system. The existence of various types and implementations of local file systems, some of which are comparatively recent (like Reiser4 [1]) show that this is still a very active area of research and improvement.

As computer networks became ubiquitous, the importance of accessing files located on remote computer systems has become more and more important. First file sharing applications were based on interactive protocols such as the File Transfer Protocol (FTP) [2] which required users to login and explicitly initiate a file transfer. Each application wishing to make use of remote file resources needed its own network support code and it was soon clear that what is needed is a system that offers transparent access to remote files in a way that is indistinguishable from accessing local files. The most popular implementation of this principle is the Network File System (NFS) [3] developed at Sun Microsystems, Inc. The NFS is strictly a client-server protocol, concerned with providing multiple clients with data stored on a single server. While very successful at the task it is meant to do, it is not applicable in situations where data is located (and must remain synchronised) on many servers at the same time.

Several network-distributed file system have been created to address this issue, including Sprite [4], AFS and Coda [5], InterMezzo [9], GFS [6] and others. Each of the referenced file systems is implemented with at least some of its components in kernel space. The terms “distributed” and “replicated” file systems are used as interchangeable in most previous works in this area, though a “distributed” file system does not imply it is automatically “replicated” (i.e. a “distributed” file system can hold different files or directory trees on different servers, while a “replicated” file system holds exact copy of data on each server).

This work presents an experimental solution for a replicated file system implemented entirely in user-space. This file system is currently informally called Trivially Distributed File System (TDFS), though it is actually a replicated file system. This name might change in the future as more features are added to the system. TDFS was implemented and tested on the FreeBSD operating system, but is portable to other operating systems.

2. Overview of Different Implementations of Distributed File Systems

In order to justify and explain the design of TDFS, we'll present examples of previous work done in this area.

2.1. Historical Network File Systems

The Network file system (NFS) must be mentioned as it is the first widely deployed transparent network file system [7]. It has evolved over time, acquiring features needed to support technological progress in networks and operating systems. Though Version 4 is the latest standard version, and has been for some time, most sites currently implement Version 3 (also called NFSv3). Because of its wide deployment and support (both by operating systems and by applications), features offered by NFSv3 can be considered a golden standard to which other network file systems can be compared.

NFS is essentially a stateless Remote Procedure Call (RPC) based protocol, meaning that every request a client makes is in the form of a procedure call that calls code implemented on the remote server, and every such call carries all data needed to satisfy the request and return a response. NFS uses standard Unix RPC mechanism (also developed at Sun Microsystems), which handles cross-platform compatibility issues by using External data representation format (XDR) for serialising data over the network. Almost all later network file systems implement similar RPC mechanisms.

NFS clients make no assumptions about server state and any caching of data is “unofficial” and with limited implicit semantics such as “call result can be reused if it's less than 30 seconds old.” NFS clients do not get notifications from servers (do not behave as RPC servers). Benefits of this approach are relatively easy implementation, excellent performance in the average case and easy error recovery (clients only have to wait until servers recover). Although NFS was designed to closely follow Unix file semantics, with minimum deviations needed for reasonable performance, file locking was introduced only as a separate service outside the main protocol. Usually, NFS is used to export an existing directory tree on the server over the network.

Next logical step deserving mention is the Andrew File System (AFS) [8], which provides explicit controls for data caching on the client side. It is also implemented as a RPC protocol, but this time clients responding to RPC calls from server in order to perform cache invalidation. In AFS, files are fetched from remote server into local caches on first use, so that most of file operations (most importantly, *read* and *write* operations) are performed on local data. Modified portions of files are transferred back to the server (only) when the file is closed. Local data cache is stored in special format in local file system and is persistent across reboots of both client and server systems. Implicitly, the only file locking semantics supported are those that work on entire files. The AFS is not considered a replicated file system (though it supports simplified read-only replication), but it is a distributed file system, where certain servers hold certain directory trees. Clients can request files from any server and are internally redirected to the proper server.

2.2. Network-distributed and Replicated file Systems

The Coda file system [5] started as an advancement of AFS and introduced several new key features: server replication, disconnected operation and resolution of diverging replicas with bandwidth adaptation. It extends the set of client and server RPC calls with those used for managing disconnected operation (i.e. when the connection between client and a server is not possible) and synchronising data between multiple servers. Both issues are addressed by using modification journals which contain timestamped and versioned records of file system operations, which are replayed when needed on the systems they are needed.

InterMezzo file system is implemented as a sort of stacked file system on top of ext2 file system [9] (i.e. it does not define its own file system format but depends closely on ext2 on which it replicates directory structures among the servers). It presents a somewhat different approach than the previously described file systems in that it tries to combine resolving most of file system requests from local cache and at the same time provide on-the-fly updates to server data. It started as an improvement to Coda.

Global File System (GFS) uses a specific approach, relying on high-speed network storage systems (Storage Area Networks – SANs). It implements inter-client synchronisation operations directly on storage in a way similar to that used by operating systems in RAM. In this way, clients are not directly aware of each other but implicitly communicate using storage [6].

All distributed file systems are influenced by Unix-like file semantics, most notably that readers immediately see changes to files made by writers, the “last write wins,” and that metadata updates should be synchronous to help recovery

in case of system crash. But to achieve speed, some systems (like AFS and Coda) modify and/or relax these semantics, which make them unsuitable for demanding applications such as databases.

3. Design and Implementation of TDFS

Both design and implementation of TDFS are influenced by two factors:

1. Availability and completeness of support for user space file systems provided by the FUSE project
2. Existing implementations of distributed file systems

Additionally, much thought was given in making the design as simple and robust as possible, in order to facilitate extensibility in the future.

3.1. The FUSE project: File Systems in User Space

Traditional operating system design makes a division between the "kernel" of the system, which is responsible for interfacing with hardware devices, provides networking, file system and other low-level services and "user space" where user applications execute. Typically the kernel is "monolithic," meaning that all its services share a single memory address space and execute at elevated hardware privilege level, and strictly divided from user space, in which each application has its own address space.

File systems are traditionally implemented in kernel space to take advantage of low overhead of accessing internal kernel services such as storage device drivers (typically this is to avoid a *context switch* between user space and kernel space) and to make themselves transparently accessible to any and all user space processes.

FUSE project aims to provide the means by which file systems can be written in user space and exposed via the kernel to rest of the operating system in a transparent way (i.e. not distinguishable from other file systems). The infrastructure to achieve this consists of two parts: operating system specific kernel module and portable user space libraries, which interface as depicted in Fig. 1. The FUSE system was developed on Linux (on which it has become a standard feature of the kernel) but has recently been ported, together with a new kernel module, to FreeBSD [10].

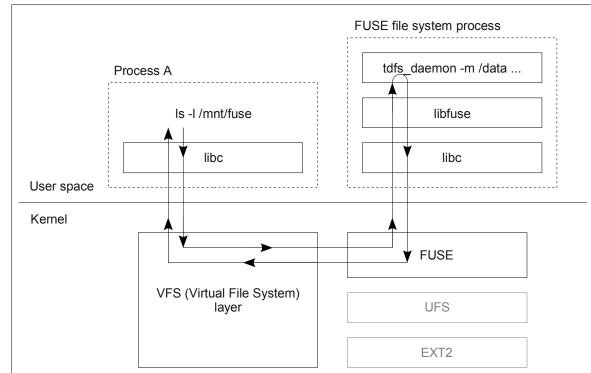

Figure 1. Path of a simple file system request

FUSE currently supports all major file system functionalities except file locking. Additionally, most functionalities are exposed on a very high level, making them efficient in terms of eliminating redundant interaction between file system implementation and kernel code (and thus context switches). Both the main FUSE library and the kernel module are currently in development and new features are constantly being implemented.

3.2. Design of TDFS

The TDFS is implemented as two user space daemons called the "master" and the "slave" daemon. The master daemon is a FUSE file system implementation (FUSE provides a "*backend*" service to user space applications) and it provides a virtual device which is mounted into the file system hierarchy in the usual way. A master daemon connects to one or more slave daemons running on remote hosts (or, for purpose of testing, the local host). Communication between master and slave daemons is conducted using standard TCP/IP facilities provided by the operating system.

The guiding idea behind TDFS is that it should act as a "stacked" file system, offering network replication of data stored in arbitrary file systems from a system running the master daemon to the ones running slave daemons. To achieve high speed as well as simplicity of operation, all read requests are satisfied from the file system copy which is local to the particular daemon. In this respect, it is similar to InterMezzo system, but it is not dependant on type of the underlying file system. All considerations about file

caching are left to the usual mechanisms of the operating system.

Other architectural goals of TDFS are that it should natively support data compression over the network, that it should follow Unix file semantics as closely as possible (especially those referencing data consistency), and that the number of slaves connected to a master should not be limited by design.

3.3. Implementation of TDFS

Internally the FUSE system is multi-threaded: several (10 by default) file system requests can be processed at the same time in the user space, and this design influences the way master daemons work. In addition to FUSE threads (created to respond to system requests coming from kernel), there are two more threads that are used to manage the network protocol and system state. Thread management and synchronisation is implemented using POSIX threads [11] package (*pthreads*) provided by the operating system.

As a single slave daemon can only be connected to one master daemon in the experimental implementation, the slave daemons are single threaded. By way of network protocol built on TCP the master daemon sends messages containing high-level descriptions of file system operations to each connected slave. Slaves use this information to replay file system operations in a local file system tree. These local file system trees (slave replicas) can be independently accessed for read-only operations by local applications, making full use of data caching facilities provided by the operating system.

TDFS uses a relatively simple network protocol, similar to that of NFS in the aspect that one side of the client-server pair (in this case, the "slave" daemon) is as simple as possible and doesn't initiate operations (rather, it only responds to the messages arriving from the master daemon). Individual threads of the master daemon and a single slave daemon operate in lock-step configuration: when a kernel request is handled to a master daemon thread, the thread sends a message block to slave daemons and blocks execution until all of them return execution status. A value indicating success is returned to the kernel only if the local operation and all remote operations have executed successfully.

The experimental implementation supports one compression method, provided by *liblzf* library [12] chosen for its remarkable speed and decent compression ratio.

4. Comparison to Existing Solutions

Some features present in TDFS are similar to those of NFS, others are also found in more advanced systems such as the InterMezzo. Like NFS, the TDFS has a simple lock-step protocol in which one side of communication channel is subordinate (doesn't originate requests). Similar to InterMezzo, TDFS is a pure stacked file system that has to be mounted on top of existing file systems, but unlike it, TDFS is independent of local file system types on master and slave machines.

The one notably lacking feature in TDFS is file locking support, because no such support is provided in the FUSE system. While on the master server the locking is handled by the local operating system infrastructure, there are no locking messages in the network protocol, and the slave daemons are not aware of file locking events that happen on the master. Though in theory a multi-master operation could be achieved by setting up both master and slave daemons to run on each of several servers, missing support for file locking presents a major problem for this mode of operation.

The underlying FUSE infrastructure offers several tunable properties that can influence performance and overall behaviour of the file system implementation. Most notable of these are options for metadata caching, enabling of asynchronous read operations and specifiable maximum size for *write()* requests (*max_write*). The TDFS supports those optimisations by design and their usage can make a difference in performance. In particular, the *max_write* option directly influences the maximum message size in network communication with the slave daemon.

Currently, the protocol used by TDFS is unoptimised, and lacks several features that make NFS and other network-aware file systems perform better (such as caching, read-ahead and coalescing of small consecutive requests [7]). Even so, the speed achieved in regular usage can rival those systems when compression is turned on (because it can greatly reduce network traffic). The experimental implementation also lacks internal synchronisation mechanisms, and relies on external utilities such as *rsync*[13].

4.1. On Performance

To explore performance of TDFS several benchmarks were conducted on two computers connected end to end into a private 100Mbit/s network. The master computer was a Pentium M laptop (1.5GHz clock speed) and the slave was an Athlon 64 desktop machine (1.8GHz clock speed). All benchmark data presented here are average of 5 benchmark runs.

TDFS performance is dependant mostly on three distinct factors:

1. Kernel to user space communication latency in FUSE system
2. Network transmission latency
3. Master-slave protocol overhead

Hard drive performance is not included because practical measurements indicate that latencies from hard drive access are dwarfed by those listed above.

While latencies in communication between kernel services and user space are relatively high (compared to latencies of communication between parts of the same application or between kernel services), CPU power available today makes this a non-issue for all but highly performance-critical systems (either embedded or high-powered).

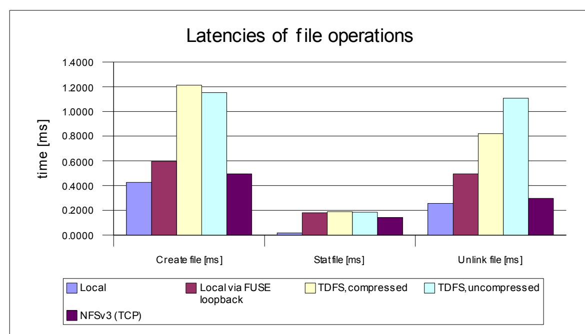

Figure 2. Speed of common file operations

Benchmark results presented in Fig. 2. indicate that, while still significant, these latencies are much smaller than those introduced by network and protocol overhead. Even with network and protocol overhead, speed is still well in the same order of magnitude with that of local access. NFS is doing particularly well in this benchmark because it has the ability to buffer and coalesce metadata requests into larger requests.

Latencies introduced by network communication can be improved greatly by introducing high-speed network equipment (e.g. Gigabit Ethernet or better). Unfortunately, no such equipment was available for testing TDFS so all tests were conducted on 100Mbit/s Ethernet.

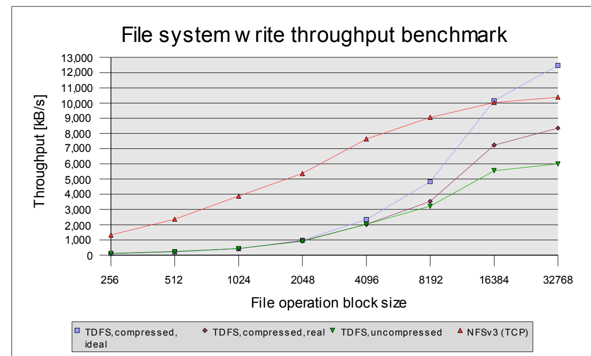

Figure 3. Write throughput at various block sizes

Because of the unbuffered lock-step operation of TDFS network protocol, benchmark results for small block sizes (presented in Fig. 3) are notably worse than that of NFS. Rresults for the largest block sizes are better than NFS because of data compression in the protocol. For the large block sizes, uncompressed throughput is less than 2x worse than that of NFS. The “ideal” and “real” marks distinguish between best-case compression (blocks of zeroes, trivially compressible) and real-case (data from a text file, compressed to 50%-75% of original size) data for TDFS.

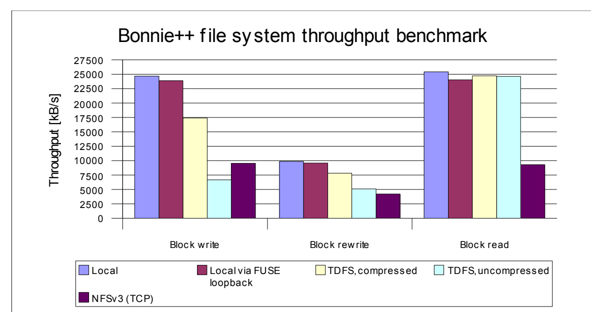

Figure 4. Bonnie++ benchmarks

Bonnie++ is an industry-standard file system benchmark used to benchmark ideal performance in a uniform and repeatable way. Contrasted to benchmarks from Fig. 2 and Fig 3. (made with a custom utility) this benchmark is conducted with full file system buffering (the default configuration for server computer systems).

Results presented in Fig. 4 nicely emphasize main assets of TDFS in an ideal environment: that read requests are satisfied from local files, and that built-in network protocol compression can have a large influence on performance. In the first benchmark *bonnie++* writes large blocks filled with zero bytes which can be compressed extensively, so the achieved throughput in the "TDFS, compressed" run is much higher than what the physical network bandwidth offers. The third benchmark performs large block reads, which are satisfied locally and the results are again much higher than network bandwidth allows. In the middle, the second benchmark repeatedly reads and writes the same blocks of data, and again gets optimised by local reads and compressed network writes. This benchmark exposes the worst case for NFS as both reads and writes must pass through the network.

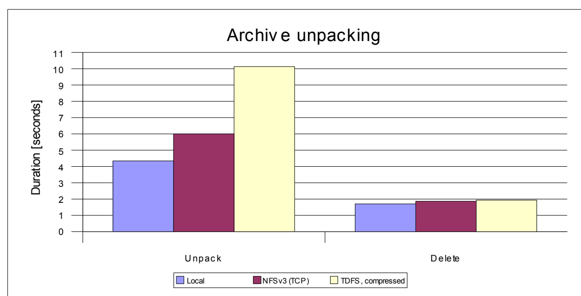

Figure 5. Archive unpacking speed

Finally, Fig. 5 shows the results of a real-world benchmark, unpacking of a source-code archive which, unpacked, contains 35MB of various text and source code files which vary in size from couple of bytes to over 1MB. This benchmark shows that the presence of many small files is bad for TDFS's performance, but that even in such pessimistic case TDFS is less than 2x slower than NFS.

5. Conclusion

This work has explored the idea of creating a distributed, network replicated file system entirely in user space of a computer operating system. Though file systems have traditionally been implemented strictly as kernel services because of speed considerations, the rise in available computing power has made a user space implementation possible and feasible. An experimental file system (named Trivially Distributed File System – TDFS) was implemented and benchmarked to show the potential performance of such systems. Results are optimistic and show that even with a trivial network protocol and only basic optimisations, performance is adequate even for a moderately demanding environment. Further work is expected to turn this experimental implementation into a reliable and fully featured solution.

6. References

[1] Hans Reiser: *Reiser4 File system*, online publication, http://www.namesys.com/v4/v4.html

[2] J. Postel, J. Reynolds, *File Transfer Protocol (FTP)*, IETF RFC Document #959, 1985.

[3] Sun Microsystems: *NFS: Network File System Protocol Implementation*, IETF RFC Document #1094, 1989.

[4] J. Ousterhout, A. Cherenson, F. Douglis et al, *The Sprite Network Operating System*, IEEE Computer, pp. 23–36, February 1988.

[5] M. Satyanarayanan, *Coda: A Highly Available File System for a Distributed Workstation Environment*, Proceedings of the Second IEEE Workshop on Workstation Operating Systems, 1989.

[6] S. Soltis, G. Erickson, K. Preslan et al, *The Global File System: A File System for Shared Disk Storage*, IEEE Transactions on Parallel and Distributed Systems, October 1997.

[7] P. J. Braam: *File Systems for Clusters from a Protocol Perspective*, Second Extreme Linux Topic Workshop, Monterey CA, 1999.

[8] J. Howard, M. Kazar, S. Menees et al, *Scale and Performance in a Distributed File System*, ACM Transactions on Computer Systems, vol. 6, issue 1, p. 51-81, 1988.

[9] P. J. Braam, Philip A. Nelson: *Removing Bottlenecks in Distributed Filesystems: Coda & InterMezzo as Examples*, Proceedings of Linux Expo 1999.

[10] C. Henk: *Fuse4BSD*, online publication, http://fuse4bsd.creo.hu/

[11] A. Josey, chair, *POSIX Threads*, IEEE Standard 1003.1, 2004 Edition

[12] M. Lehmann: *LibLZF: fast compression library*, online publication, http://www.goof.com/pcg/marc/liblzf.html

[13] A. Tridgell: *Efficient Algorithms for Sorting and Synchronisation*, Ph.D. thesis, Australian National University, 1999.