# Structure and Performance of the Direct Access File System

Kostas Magoutis, Salimah Addetia, Alexandra Fedorova, Margo I. Seltzer
*Division of Engineering and Applied Sciences, Harvard University*
Jeffrey S. Chase, Andrew J. Gallatin, Richard Kisley, Rajiv G. Wickremesinghe
*Department of Computer Science, Duke University*
Eran Gabber
*Lucent Technologies - Bell Laboratories*

## Abstract

The Direct Access File System (DAFS) is an emerging industrial standard for network-attached storage. DAFS takes advantage of new user-level network interface standards. This enables a *user-level file system* structure in which client-side functionality for remote data access resides in a library rather than in the kernel. This structure addresses longstanding performance problems stemming from weak integration of buffering layers in the network transport, kernel-based file systems and applications. The benefits of this architecture include lightweight, portable and asynchronous access to network storage and improved application control over data movement, caching and prefetching.

This paper explores the fundamental performance characteristics of a user-level file system structure based on DAFS. It presents experimental results from an open-source DAFS prototype and compares its performance to a kernel-based NFS implementation optimized for zero-copy data transfer. The results show that both systems can deliver file access throughput in excess of 100 MB/s, saturating network links with similar raw bandwidth. Lower client overhead in the DAFS configuration can improve application performance by up to 40% over optimized NFS when application processing and I/O demands are well-balanced.

## 1 Introduction

The performance of high-speed network storage systems is often limited by client overhead, such as memory copying, network access costs and protocol overhead [2, 8, 20, 29]. A related source of inefficiency stems from poor integration of applications and file system services; lack of control over kernel policies leads to problems such as double caching, false prefetching and poor concurrency management [34]. As a result, databases and other performance-critical applications often bypass file systems in favor of raw block storage access. This sacrifices the benefits of the file system model, including ease of administration and safe sharing of resources and data. These problems have also motivated the design of radical operating system structures to allow application control over resource management [21, 31].

The recent emergence of commercial *direct-access transport* networks creates an opportunity to address these issues without changing operating systems in common use. These networks incorporate two defining features: *user-level networking* and *remote direct memory access* (RDMA). User-level networking allows safe network communication directly from user-mode applications, removing the kernel from the critical I/O path. RDMA allows the network adapter to reduce copy overhead by accessing application buffers directly.

The Direct Access File System (DAFS) [14] is a new standard for network-attached storage over direct-access transport networks. The DAFS protocol is based on the Network File System Version 4 protocol [32], with added protocol features for direct data transfer using RDMA, scatter/gather list I/O, reliable locking, command flow-control and session recovery. DAFS is designed to enable a *user-level file system client*: a DAFS client may run as an application library above the operating system kernel, with the kernel's role limited to basic network device support and memory management. This structure can improve performance, portability and reliability, and offer applications fully asynchronous I/O and more direct control over data movement and caching. Network Appliance and other network-attached storage vendors are planning DAFS interfaces for their products.

This paper explores the fundamental structural and performance characteristics of network file access using a user-level file system structure on a direct-access transport network with RDMA. We use DAFS as a basis for exploring these features since it is the first fully-specified file system protocol to support them. We describe DAFS-based client and server reference implementations for an open-source Unix system (FreeBSD) and report experimental results, comparing DAFS to a zero-copy NFS implementation. Our purpose is to illustrate the benefits and tradeoffs of these techniques to provide a basis for informed choices about deployment of DAFS-based systems and similar extensions to other network file protocols, such as NFS.

Our experiments explore the application properties that determine how RDMA and user-level file systems affect performance. For example, when a workload is balanced (i.e., the application simultaneously saturates the CPU and network link) DAFS delivers the most benefit compared to more traditional architectures. When workloads are limited by the disk, DAFS and more traditional network file systems behave comparably. Other workload factors such as metadata-intensity, I/O sizes, file sizes, and I/O access pattern also influence performance.

An important property of the user-level file system structure is that applications are no longer bound by the kernel's policies for file system buffering, caching and prefetching. The user-level file system structure and the DAFS API allow applications full control over file system access; however, the application can no longer benefit from shared kernel facilities for caching and prefetching. A secondary goal of our work is to show how *adaptation libraries* for specific classes of applications enable those applications to benefit from improved control and tighter integration with the file system, while reducing or eliminating the burden on application developers. We present experiments with two adaptation libraries for DAFS clients: Berkeley DB [28] and the TPIE external memory I/O toolkit [37]. These adaptation libraries provide the benefits of the user-level file system without requiring that applications be modified to use the DAFS API.

The layout of this paper is as follows. Section 2 summarizes the trends that motivated DAFS and user-level file systems and sets our study in context with previous work. Section 3 gives an overview of the salient features of the DAFS specifications, and Section 4 describes the DAFS reference implementation used in the experiments. Section 5 presents two example adaptation libraries, and Section 6 describes zero-copy, kernel-based NFS as an alternative to DAFS. Section 7 presents experimental results. We conclude in Section 8.

## 2 Background and Related Work

In this section, we discuss the previous work that lays the foundation for DAFS and provides the context for our experimental results. We begin with a discussion of the issues that limit performance in network storage systems and then discuss the two critical architectural features that we examine to attack performance bottlenecks: direct-access transports and user-level file systems.

### 2.1 Network Storage Performance

Network storage solutions can be categorized as Storage-Area Network (SAN)-based solutions, which provide a block abstraction to clients, and Network-Attached Storage (NAS)-based solutions, which export a network file system interface. Because a SAN storage volume appears as a local disk, the client has full control over the volume's data layout; client-side file systems or database software can run unmodified [23]. However, this precludes concurrent access to the shared volume from other clients, unless the client software is extended to coordinate its accesses with other clients [36]. In contrast, a NAS-based file service can control sharing and access for individual files on a shared volume. This approach allows safe data sharing across diverse clients and applications.

Communication overhead was a key factor driving acceptance of Fibre Channel [20] as a high-performance SAN. Fibre Channel leverages network interface controller (NIC) support to offload transport processing from the host and access I/O blocks in host memory directly without copying. Recently, NICs supporting the emerging iSCSI block storage standard have entered the market as an IP-based SAN alternative. In contrast, NAS solutions have typically used IP-based protocols over conventional NICs, and have paid a performance penalty. The most-often cited causes for poor performance of network file systems are (a) protocol processing in network stacks; (b) memory copies [2, 15, 29, 35]; and (c) other kernel overhead such as system calls and context switches. Data copying, in particular, incurs substantial per-byte overhead in the CPU and memory system that is not masked by advancing processor technology.

One way to reduce network storage access over-

head is to offload some or all of the transport protocol processing to the NIC. Many network adapters can compute Internet checksums as data moves to and from host memory; this approach is relatively simple and delivers a substantial benefit. An increasing number of adapters can offload all TCP or UDP protocol processing, but more substantial kernel revisions are needed to use them. Neither approach by itself avoids the fundamental overheads of data copying.

Several known techniques can remove copies from the transport data path. Previous work has explored copy avoidance for TCP/IP communication (Chase et al. [8] provide a summary). Brustoloni [5] introduced *emulated copy*, a scheme that avoids copying in network I/O while preserving copy semantics. IO-Lite [29] adds scatter/gather features to the I/O API and relies on support from the NIC to handle multiple client processes safely without copying. Another approach is to implement critical applications (e.g., Web servers) in the kernel [19]. Some of the advantages can be obtained more cleanly with combined data movement primitives, e.g., *sendfile*, which move data from storage directly to a network connection without a user space transfer; this is useful for file transfer in common server applications.

DAFS was introduced to combine the low overhead and flexibility of SAN products with the generality of NAS file services. The DAFS approach to removing these overheads is to use a direct-access transport to read and write application buffers directly. DAFS also enables implementation of the file system client at user level for improved efficiency, portability and application control. The next two sections discuss these aspects of DAFS in more detail. In Section 6, we discuss an alternative approach that reduces NFS overhead by eliminating data copying.

## 2.2 Direct-Access Transports

Direct-access transports are characterized by NIC support for remote direct memory access (RDMA), user-level networking with minimal kernel overhead, reliable messaging transport connections and per-connection buffering, and efficient asynchronous event notification. The Virtual Interface (VI) Architecture [12] defines a host interface and API for NICs supporting these features.

Direct-access transports enable *user-level networking* in which the user-mode process interacts directly with the NIC to send or receive messages

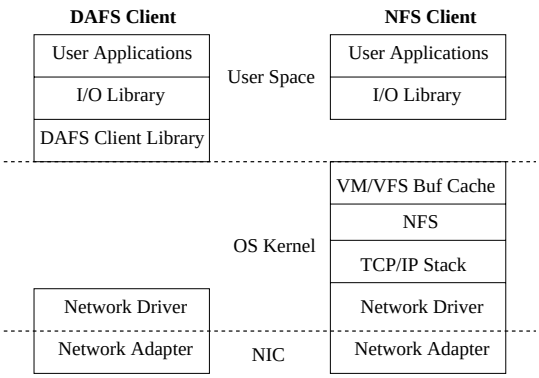


Figure 1: User-level vs. kernel-based client file system structure.

with minimal intervention from the operating system kernel. The NIC device exposes an array of connection descriptors to the system physical address space. At connection setup time, the kernel network driver maps a free connection descriptor into the user process virtual address space, giving the process direct and safe access to NIC control registers and buffer queues in the descriptor. This enables RDMA, which allows the network adapter to reduce copy overhead by accessing application buffers directly. The combination of user-level network access and copy avoidance has a lengthy heritage in research systems spanning two decades [2, 4, 6, 33, 38].

The experiments in Section 7 quantify the improvement in access overhead that DAFS gains from RDMA and transport offload on direct-access NICs.

## 2.3 User-Level File Systems

In addition to overhead reduction, the DAFS protocol leverages user-level networking to enable the network file system structure depicted in the left-hand side of Figure 1. In contrast to traditional kernel-based network file system implementations, as shown in the right side of Figure 1, DAFS file clients may run in user mode as libraries linked directly with applications.

While DAFS also supports kernel-based clients, our work focuses primarily on the properties of the user-level file system structure. A user-level client yields additional modest reductions in overhead by removing system call costs. Perhaps more importantly, it can run on any operating system, with no special kernel support needed other than the NIC driver itself. The client may evolve independently of the operating system, and multiple

client implementations may run on the same system. Most importantly, this structure offers an opportunity to improve integration of file system functions with I/O-intensive applications. In particular, it enables fully asynchronous pipelined file system access, even on systems with inadequate kernel support for asynchronous I/O, and it offers full application control over caching, data movement and prefetching.

It has long been recognized that the kernel policies for file system caching and prefetching are poorly matched to the needs of some important applications [34]. Migrating these OS functions into libraries to allow improved application control and specialization is similar in spirit to the *library operating systems* of Exokernel [21], *protocol service decomposition* for high-speed networking [24], and related approaches. User-level file systems were conceived for the SHRIMP project [4] and the Network-Attached Secure Disks (NASD) project [18]. NFS and other network file system protocols could support user-level clients over an RPC layer incorporating the relevant features of DAFS [7], and we believe that our results and conclusions would apply to such a system.

Earlier work arguing against user-level file systems [39] assumed some form of kernel mediation in the critical I/O path and did not take into account the primary sources of overhead outlined in Section 2.1. However, the user-level structure considered in this paper does have potential disadvantages. It depends on direct-access network hardware, which is not yet widely deployed. Although an application can control caching and prefetching, it does not benefit from the common policies for shared caching and prefetching in the kernel. Thus, in its simplest form, this structure places more burden on the application to manage data movement, and it may be necessary to extend applications to use a new file system API. Section 5 shows how this power and complexity can be encapsulated in prepackaged I/O *adaptation libraries* (depicted in Figure 1) implementing APIs and policies appropriate for a particular class of applications. If the adaptation API has the same syntax and semantics as a pre-existing API, then it is unnecessary to modify the applications themselves (or the operating system).

# 3 DAFS Architecture and Standards

The DAFS specification grew out of the DAFS Collaborative, an industry/academic consortium led by Network Appliance and Intel, and it is presently undergoing standardization through the Storage Networking Industry Association (SNIA).

The draft standard defines the *DAFS protocol* [14] as a set of request and response formats and their semantics, and a recommended procedural *DAFS API* [13] to access the DAFS service from a client program. Because library-level components may be replaced, client programs may access a DAFS service through any convenient I/O interface. The DAFS API is specified as a recommended interface to promote portability of DAFS client programs. The DAFS API is richer and more complex than common file system APIs including the standard Unix system call interface.

The next section gives an overview of the DAFS architecture and standards, with an emphasis on the transport-related aspects: Sections 3.2 and 3.3 focus on DAFS support for RDMA and asynchronous file I/O respectively.

## 3.1 DAFS Protocol Summary

The DAFS protocol derives from NFS Version 4 [32] (NFSv4) but diverges from it in several significant ways. DAFS assumes a reliable network transport and offers server-directed command flow-control in a manner similar to block storage protocols such as iSCSI. In contrast to NFSv4, every DAFS operation is a separate request, but DAFS supports request chaining to allow pipelining of dependent requests (e.g., a name *lookup* or *open* followed by file *read*). DAFS protocol headers are organized to preserve alignment of fixed-size fields. DAFS also defines features for reliable session recovery and enhanced locking primitives. To enable the application (or an adaptation layer) to support file caching, DAFS adopts the NFSv4 mechanism for consistent caching based on *open delegations* [1, 14, 32].

The DAFS specification is independent of the underlying transport, but its features depend on direct-access NICs. In addition, some transport-level features (e.g., message flow-control) are defined within the DAFS protocol itself, although they could be viewed as a separate layer below the file service protocol.

## 3.2 Direct-Access Data Transfer

To benefit from RDMA, DAFS supports *direct* variants of key data transfer operations (*read, write, readdir, getattr, setattr*). Direct operations transfer

directly to or from client-provided memory regions using RDMA *read* or *write* operations as described in Section 2.2.

The client must register each memory region with the local kernel before requesting direct I/O on the region. The DAFS API defines primitives to *register* and *unregister* memory regions for direct I/O; the *register* primitive returns a region descriptor to designate the region for direct I/O operations. In current implementations, registration issues a system call to pin buffer regions in physical memory, then loads page translations for the region into a lookup table on the NIC so that it may interpret incoming RDMA directives. To control buffer pinning by a process for direct I/O, the operating system should impose a resource limit similar to that applied in the case of the 4.4BSD *mlock* API [26]. Buffer registration may be encapsulated in an adaptation library.

RDMA operations for direct I/O in the DAFS protocol are always initiated by the server rather than a client. For example, to request a DAFS direct write, the client's write request to the server includes a region token for the buffer containing the data. The server then issues an RDMA *read* to fetch the data from the client, and responds to the DAFS write request after the RDMA completes. This allows the server to manage its buffers and control the order and rate of data transfer [27].

### 3.3 Asynchronous I/O and Prefetching

The DAFS API supports a fully asynchronous interface, enabling clients to pipeline I/O operations and overlap them with application processing. A flexible event notification mechanism delivers asynchronous I/O completions: the client may create an arbitrary number of *completion groups*, specify an arbitrary completion group for each DAFS operation and poll or wait for events on any completion group.

The asynchronous I/O primitives enable event-driven application architectures as an alternative to multithreading. Event-driven application structures are often more efficient and more portable than those based on threads. Asynchronous I/O APIs allow better application control over concurrency, often with lower overhead than synchronous I/O using threads.

Many NFS implementations support a limited form of asynchrony beneath synchronous kernel I/O APIs. Typically, multiple processes (called *I/O daemons* or *nfsiods*) issue blocking requests for sequential block read-ahead or write-behind. Unfortunately, frequent *nfsiod* context switching adds overhead [2]. The kernel policies only prefetch after a run of sequential reads and may prefetch erroneously if future reads are not sequential.

## 4 DAFS Reference Implementation

We have built prototypes of a user-level DAFS client and a kernel DAFS server implementation for FreeBSD. Both sides of the reference implementation use protocol stubs in a DAFS SDK provided by Network Appliance. The reference implementation currently uses a 1.25 Gb/s Giganet cLAN VI interconnect.

### 4.1 User-level Client

The user-level DAFS client is based on a three-module design, separating transport functions, flow-control and protocol handling. It implements an asynchronous event-driven control core for the DAFS request/response channel protocol. The subset of the DAFS API supported includes direct and asynchronous variants of basic file access and data transfer operations.

The client design allows full asynchrony for single-threaded applications. All requests to the library are non-blocking, unless the caller explicitly requests to wait for a pending completion. The client polls for event completion in the context of application threads, in explicit polling requests and in a standard preamble/epilogue executed on every entry and exit to the library. At these points, it checks for received responses and may also initiate pending sends if permitted by the request flow-control window. Each thread entry into the library advances the work of the client. One drawback of this structure is that pending completions build up on the client receive queues if the application does not enter the library. However, deferring response processing in this case does not interfere with the activity of the client, since it is not collecting its completions or initiating new I/O. A more general approach was recently proposed for asynchronous application-level networking in Exokernel [16].

### 4.2 Kernel Server

The kernel-based DAFS server [25] is a kernel-loadable module for FreeBSD 4.3-RELEASE that implements the complete DAFS specification. Using the VFS/Vnode interface, the server may export

DOCKET ALARM

# Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

## Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

## Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

## Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

## API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

**LAW FIRMS**
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

**FINANCIAL INSTITUTIONS**
Litigation and bankruptcy checks for companies and debtors.

**E-DISCOVERY AND LEGAL VENDORS**
Sync your system to PACER to automate legal marketing.