USENIX Association

Proceedings of FAST '03: 2nd USENIX Conference on File and Storage Technologies

San Francisco, CA, USA
March 31–April 2, 2003

The Direct Access File System

Matt DeBergalis, Peter Corbett, Steve Kleiman, Arthur Lent
Dave Noveck, Tom Talpey, Mark Wittle
Network Appliance, Inc.

Abstract

The Direct Access File System (DAFS) is a new, fast, and lightweight remote file system protocol. DAFS targets the data center by addressing the performance and functional needs of clusters of application servers. We call this the *local file sharing* environment. File access performance is improved by utilizing Direct Access Transports, such as InfiniBand, Remote Direct Data Placement, and the Virtual Interface Architecture. DAFS also enhances file sharing semantics compared to prior network file system protocols. Applications using DAFS through a user-space I/O library can bypass operating system overhead, further improving performance. We present performance measurements of an IP-based DAFS network, demonstrating the DAFS protocol's lower client CPU requirements over commodity Gigabit Ethernet. We also provide the first multiprocessor scaling results for a well-known application (GNU gzip) converted to use DAFS.

1 Introduction

With the advent of the Virtual Interface Architecture [11], InfiniBand [1], and the Remote Direct Data Placement (RDDP) protocol [13], messaging transports that support Remote Direct Memory Access (RDMA) operations are moving from the realm of esoteric, single-vendor implementations into the mainstream of commodity technology. The DAFS Collaborative was launched in 2000 with the mission of developing a network file access protocol, the Direct Access File System (DAFS) [9], that takes full advantage of such *Direct Access Transports* (DATs), building on the semantics of existing network file system protocols. DAFS employs many concepts and semantics of NFSv4 [12], which in turn is derived from earlier versions of NFS and incorporates some semantic features of CIFS [30]. Unlike NFSv4, DAFS includes semantics for which there were no pre-existing APIs. The Collaborative defined the DAFS API to access them.

The design of DAFS targets two goals. The first is to enable low-latency, high-throughput, and low-overhead data movement between file system clients and servers. Part of this goal is to minimize the demands placed on the client CPU by the protocol. DAFS addresses network throughput and latency concerns by using fast DAT technologies built upon RDMA capability. It minimizes client CPU utilization by using the direct data placement and operating system bypass capabilities of the transports and by placing a premium on client CPU cycles at all stages of the protocol. By minimizing client CPU utilization, DAFS clients can potentially experience better performance accessing files via DAFS than they can by accessing files through their local file systems.

The second goal of DAFS is to include the necessary semantics to ensure reliable, shared access in a clustered system. Because it is targeted at highly scalable workloads within data center environments, support for clustered clients is essential. DAFS contains many features that enable sharing of files among clients, while allowing cooperating clients to limit access to shared files by outsiders.

1.1 Motivation

Files can be accessed in three ways: from local disk, from a remote shared disk, and over a network file system. Traditionally, local file systems [26, 28] accessing local disk storage have provided the highest performance access to file resident data. However, they do not solve the problem of sharing data sets or storage among a number of computers. Cluster file systems, such as Frangipani [20, 34], GPFS [29], and GFS [32], allow multiple clients to access a sin-

gle remote shared disk pool, by coordinating management and control of the shared file system among the clients. These systems access shared disks using transports such as iSCSI and Fibre Channel [31] that are designed for data storage, so they enjoy bulk data transfer performance on par with local disk. The price paid is relatively complex distributed data and metadata locking protocols, and complex failure handling. These systems also demand client software homogeneity.

Network file systems [16, 27] permit heterogenous clients to share data on remote disks using more traditional networks. This characteristic comes at the expense of relatively poor performance compared to local file systems, due in part to the high CPU and latency overhead of the network protocol layers, and the limited bandwidth of the transports. Network file systems avoid much of the complexity of cluster file system implementations by using messaging protocols to initiate operations that are performed on the server. The implementation of data sharing, fault isolation, and fault tolerance is simplified compared to cluster file systems because data and metadata modifying operations are localized on the server. Network file systems are typically less tightly integrated than cluster file systems, requiring a lower degree of locality and cooperation among the clients. We believe that there is a role for a high-performance network file system in tightly coupled environments, because of the inherent advantages of simple data sharing, interoperable interfaces, and straightforward fault isolation.

Based upon almost two decades of experience with the network file system model, it was apparent to the participants in the DAFS Collaborative that a new network file system protocol based on RDMA-capable Direct Access Transports would be most useful as a step beyond NFSv4. This need was recognized before NFSv4 was finalized, but after NFSv4 was largely defined. Unlike NFSv4, DAFS was targeted specifically at the data center, incorporating what was called the *local sharing* model as its design point. Hence DAFS assumes a higher degree of locality among the clients than NFSv4, and is designed to enable a greater degree of cooperation among those clients and the applications running on them. The intent is to enhance the support for clustered applications, while maintaining the simple fault isolation model characteristics of the network file system model.

One design alternative would be to leverage the existing ONC RPC [33] framework that underlies NFS by converting it to use RDMA for data transfer [18]. This approach would enable direct data placement, but the existing framework would not allow several other optimizations, including full asynchrony, session-oriented authentication, request throttling, and resource control. While it is likely that a reasonable modification to ONC RPC to accommodate these goals could be done, it would involve fairly major changes in the RPC framework and interface. Our goal was high-performance file access, not a general purpose RPC mechanism.

A goal of DAFS is to maximize the benefit of using a DAT. To accomplish this, it was not sufficient to simply replace the transport underneath the existing RPC based NFS protocol with an RDMA-aware RPC layer. DAFS enables clients to negotiate data transfers to and from servers without involving the client CPU in the actual transfer of data, and minimizes the client CPU required for protocol overhead. DATs offload much of the low-level network protocol processing onto the Network Interface Card (NIC), and the interfaces to these network adapters, such as DAPL [10], are callable from user space and do not require transitions into the kernel. Thus, it is possible to reduce the client CPU load of the network file system to just the costs of marshalling parameters for remote operations, and unpacking and interpreting the results. Data can be placed directly in the applications' memory by the NIC without additional data copies. DAFS further reduces the overhead of data transfer by incorporating batch and list I/O capabilities in the protocol.

To support clustered clients, DAFS extends the semantics of NFSv4 in the areas of locking, fencing, and shared key reservations. It does not require that the cluster file system implementations built using DAFS fully support POSIX file access semantics, but does provide sufficient capability to implement such semantics if desired.

The remainder of the paper is as follows. Section 2 provides some background on Direct Access Transports, the network technology required for DAFS. Section 3 introduces the core of the Direct Access File System, focusing on the rationale behind significant design decisions. Section 4 follows with the DAFS API, the standard interface to a user-space DAFS client. Section 5 demonstrates the suitability of DAFS for local file sharing by providing some performance results. Finally, the paper concludes with a summary of our work.

2 Direct Access Transports

Direct Access Transports are the state of the art in faster data transport technology. The high-performance computing community has long used memory-to-memory interconnects (MMIs) that reduce or eliminate operating system overhead and permit direct data placement [3, 4, 5, 15]. The Virtual Interface Architecture standard in the mid 1990's was the first to separate the MMI properties and an API for accessing them from the underlying physical transport. Currently, InfiniBand and RDDP are positioned as standard commodity transports going forward.

The DAT Collaborative has defined a set of requirements for Direct Access Transports, as well as the *Direct Access Provider Layer* (DAPL) API as a standard programming interface [10] to them. The DAT standard abstracts the common capabilities that MMI networks provide from their physical layer, which may differ across implementations. These capabilities include RDMA, kernel bypass, asynchronous interfaces, and memory registration. Direct memory-to-memory data transfer operations between remote nodes (RDMA) allow bulk data to bypass the normal protocol processing and to be transferred directly between application buffers on the communicating nodes, avoiding intermediate buffering and copying. Direct application access to transport-level resources (often referred to as *kernel bypass*) allows data transfer operations to be queued to network interfaces without intermediate operating system involvement. Asynchronous operations allow efficient pipelining of requests without additional threads or processes. Memory registration facilities specify how DAT NICs are granted access to host memory regions to be used as destinations of RDMA operations.

DAPL specifies a messaging protocol between established endpoints, and requires that the receiving end of a connection post pre-allocated buffer space for message reception. This model is quite similar to traditional network message flow. DATs also preserve message ordering on a given connection. The DAFS request-response protocol uses DAPL messaging primitives to post requests from client to server and receive their responses on the client. DAFS also uses RDMA to transmit bulk data directly into and out of registered application buffers in client memory. RDMA can proceed in both directions: an RDMA *write* allows host A to transfer data in a local buffer to previously exported addresses in host B's memory, while an RDMA *read* allows A to transfer data from previously exported addresses in B's memory into a local buffer. In the DAFS protocol, any RDMA operations are initiated by the server. This convention means that a client file *read* may be accomplished using RDMA *writes*. Conversely, a client may write to a file by instructing the server to issue RDMA reads.

Any storage system built using a DAT network can be measured in terms of its throughput, latency, and client CPU requirements. Server CPU is not typically a limiting factor, since file server resources can be scaled independently of the network or file system clients. DATs may be built using a variety of underlying physical networks, including 1 and 2 Gbps Fibre Channel, 1 Gbps and 10 Gbps Ethernet, InfiniBand, and various proprietary interconnects. The newer 10Gbps interconnects (e.g. 4X InfiniBand and 10Gbps Ethernet) approach the limits of today's I/O busses. In addition, multiple interfaces may be trunked. As such, network interface throughput is not a performance limiting factor.

We break storage latencies down into three additive components: the delay in moving requests and responses between the client machine and the transport wire, the round trip network transit time, and the time required to process a request on the server. In the case of disk-based storage, the third component is a function of the disks themselves. Being mechanical, their access times dominate the overall access latency. Caching on the file server mitigates the latency problem by avoiding the disk component entirely for a fraction of client requests. Asynchrony or multi-threading can hide I/O request latency by allowing the application to do other work, including pipelining of additional requests while an I/O request is in progress.

The remaining metric is the client CPU overhead associated with processing I/O. If the CPU overhead per I/O operation is high, then for typical record sizes (2–16 KB), the client can spend a significant fraction of its available CPU time handling I/O requests. Additionally, client CPU saturation can impact throughput even for bulk data-movement workloads that can otherwise tolerate high latencies. The most significant performance advantages of DAFS stem from its attention to minimizing client CPU requirements for all workload types.

There are several major sources to CPU overhead,

including operating system, protocol parsing, buffer copying, and context switching. Modern TCP/IP offload engines and intelligent storage adapters mitigate some of the protocol parsing and copying overhead, but do little to reduce system overhead. Without additional upper layer protocol support within the offload engines, incoming data will not be placed in its final destination in memory, requiring a data copy or page flip [7]. It is here that DAFS shines; only direct data placement avoids the costly data copy or page flip step in conventional network file system implementations. Previous results comparing DAFS direct data placement to offloaded NFS with page flipping show that the RDMA-based approach requires roughly 2/3 of the CPU cycles for 8KB streaming reads, improving to less than 1/10 of the cycles as the block size increases beyond 128KB [23, 24].

3 The DAFS Protocol

This section presents the Direct Access File System architecture and wire protocol. It begins with an overview of the DAFS design, followed by three subsections that cover performance improvements, file sharing semantics targeting the requirements of local file sharing environments, and security considerations.

The DAFS protocol uses a session-based client-server communication model. A DAFS session is created when a client establishes a connection to a server. During session establishment, a client authenticates itself to the server and negotiates various options that govern the session. These parameters include message byte-ordering and checksum rules, message flow control and transport buffer parameters, and credential management.

Once connected, messages between a client and server are exchanged within the context of a session. DAFS uses a simple request-response communication pattern. In DAPL, applications must pre-allocate transport buffers and assign them to a session in order to receive messages. Figure 1 shows a typical arrangement of these transport buffers, called *descriptors*, on both a client and file server. Messages are sent by filling in a *send descriptor* and posting it to the transport hardware. The receiving hardware will fill in a pre-allocated *receive descriptor* with the message contents, then dequeue the descriptor and inform the consumer that a message is available. Descriptors may contain multiple segments; the hardware will gather send segments into one message, and scatter an incoming message into multiple receive segments.

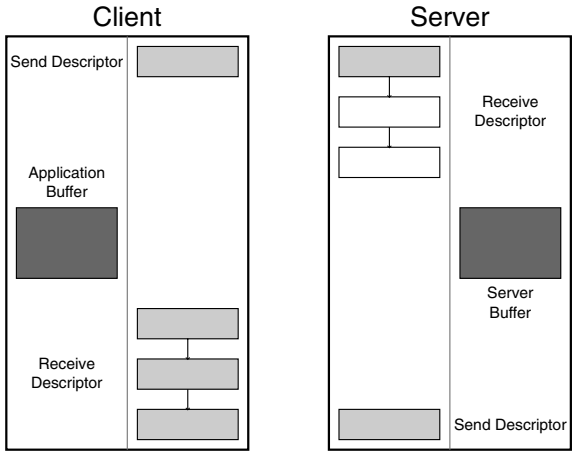

Figure 1: Client and server descriptor layout. The faint line in each system separates the provider library from its consumer, which contains an application data buffer. This and subsequent diagrams will use dark shading to show bulk data, light shading for protocol meta-data, and empty boxes for unused descriptor segments. In this diagram the receive descriptors are shown to have multiple segments; the transport hardware will scatter incoming messages into the segment list.

DAPL provides no flow control mechanism, so a DAFS server manages its own flow control credits on each open DAFS session. When a DAFS connection is established, the client and server negotiate an initial number of request credits; a client may only issue a request to a server if it has a free credit on an open session. Credit replenishment is not automatic; the server chooses when to allocate new credits to its clients.

DAFS semantics are based on the NFSv4 protocol. NFSv4 provides a broad set of basic file system operations, including file and directory management (*lookup*, *open*, *close*, *create*, *rename*, *remove*, *link*, *readdir*), file attribute and access control (*access*, *getfh*, *getattr*, *setattr*, *openattr*, *verify*, *secinfo*, *setclientid*), and data access (*read*, *write*, *commit*). All of these DAFS operations behave like their NFSv4 analogues. Most DAFS messages are small, so servers can allocate minimal transport resources waiting for incoming client requests. Instead of being included as part of the request or response,

most bulk data is transferred using RDMA reads and writes. These RDMA operations are always initiated by the server, after a client sends a *handle* to a particular application buffer along with a request for the server to write data to (DAFS read) or read data from (DAFS write) that buffer. Always initiating the RDMA on the server minimizes the cross host memory management issues in a request-response protocol.

3.1 Performance Improvements

This subsection outlines key DAFS protocol features, with emphasis on how they improve file access performance and reduce client CPU utilization.

Message format DAFS simplifies the composition and parsing of messages by placing the fixed size portion of each message at the beginning of the packet buffer, aligned on a natural boundary, where it may be composed and parsed as fixed offset structures. Clients also choose the byte order of header information, moving the potential cost of byte swapping off of the client.

Session-based authentication Since DAFS communication is session-based, the client and server are authenticated when a session is created, rather than during each file operation. DAFS relies on the notion that a DAT connection is private to its two established endpoints; specific DAT implementation may guarantee session security using transport facilities. DAFS sessions also permit a single registration of user credentials that may be used with many subsequent operations. This reduces both the basic CPU cost of file operations on the server, and the latency of operations that require expensive authentication verification. Section 3.3 describes the DAFS authentication mechanisms in greater detail.

I/O operations For most bulk data operations, including *read, write, readdir, setattr,* and *getattr*, DAFS provides two types of data transfer operations: *inline* and *direct*. Inline operations provide a standard two-message model, where a client sends a request to the server, and then after the server processes the request, it sends a response back to the client. Direct operations use a three-message model, where the initial request from the client is followed by an RDMA transfer initiated by the server, followed by the response from the server. The transport layer implementation on the client node participates in the RDMA transfer operation without interrupting the client. In general, clients use inline data transfers for small requests and direct data transfers for large requests.

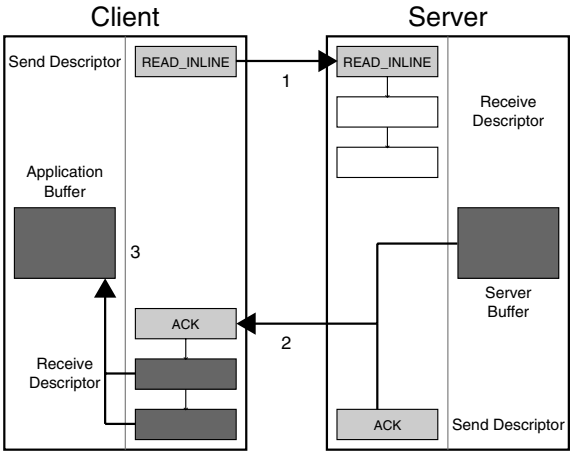

Figure 2: A DAFS inline read.

Figure 2 shows a typical inline read. In step 1, the client sends the `READ_INLINE` protocol message to the server. That message lands in a pre-allocated receive buffer, and in this case only requires a single segment of that buffer. In step 2, the server replies with a message containing both an acknowledgement and the bulk read data. The server uses a gathering post operation to avoid copying the bulk data. The client receives this message into its receive descriptor. Here, the message fills multiple segments. Finally, in step 3, the client copies the bulk data from the filled descriptor into the application's buffer, completing the read.

The DAFS protocol provides some support for replacing the copy in step 3 with a page flip by adding optional padding of inline read and write headers. For an inline read, the client requests the necessary padding length in its read request. In the case shown in Figure 2, the client would specify a padding such that the read inline reply plus the padding length exactly fills the first posted segment, leaving the bulk data aligned in the remaining segments.

Direct operations, on the other hand, offer the chief benefits of CPU-offload from the RDMA operation itself, and the inherent copy avoidance due to direct

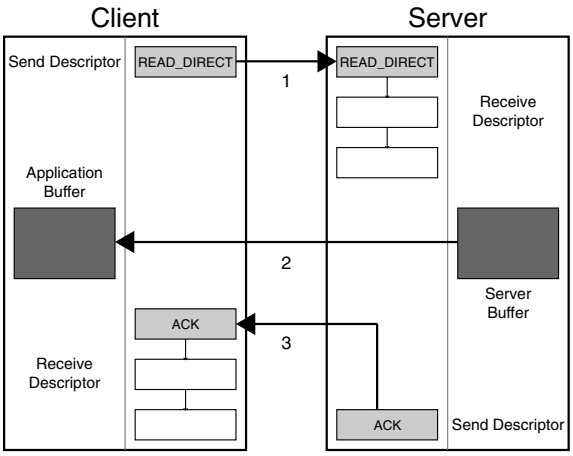

Figure 3: A DAFS direct read.

data placement (DDP) resulting from the separation of the request operation header (transferred in pre-allocated transport buffers) from request data (transported memory-to-memory via the RDMA transfer). The direct operations can also be used to transfer unusually large amounts of file meta-data (directory contents, file attributes, or symlink data). Figure 3 shows the three steps of a direct read. As with an inline read, the process begins with the client posting a short message, `READ_DIRECT` in this case, to the server. This message includes a list of memory descriptors that contain all the required information for the server to initiate an RDMA write directly into the client's application buffer. In step 2, the server gathers together the bulk data and issues an RDMA write operation to the transport. The client receives no information about this operation, so the transaction concludes with step 3, where the server sends a small acknowledgement message to the client. Since DATs preserve message ordering, the server can issue that acknowledgement immediately after the RDMA write; it need not wait for the RDMA to complete. Once the client receives the acknowledgement message, it is assured that the application buffer is filled in.

For latency-sensitive operations where transport costs dominate the request latency, the inline two-message mechanism may provide reduced overall latency, even though it requires a copy or page flip. For bulk data transfer, though, the direct data movement operations are preferred. Our user-space DAFS client always issues direct reads, since the client CPU costs are lower and the transport cost of an RDMA write is similar to a message post. Our client issues small writes as inline operations, though, since an RDMA read requires a transport-level acknowledgement and therefore has higher latency than an inline post.

By providing a small set of direct operations in addition to read and write, DAFS promotes the use of smaller inline buffers, thereby promoting additional operation concurrency. Effectively, DAFS is designed to use a large number of small pre-allocated buffers for most operations involving meta-data, and a variable number of variably sized RDMA buffers for bulk data transfer. For this reason, it is useful to determine the smallest useful buffer size. Since DAFS operations are generally issued in an unpredictable order, pre-allocated buffers are sized uniformly. In practice, since most meta-data operations require less than 1 KB of buffer space, the key trade-off in buffer size is determining whether read and write operations will be performed inline or direct. For a given memory resource cost, the number of concurrent requests can be increased dramatically by maintaining a small buffer size, and performing read and write requests using direct operations.

Reduced client overhead All RDMA transfers are initiated by the DAFS server. For file write operations, this means that the server has access to the write request header before the RDMA read transfer from the client begins. It can allocate an appropriately located, sized, and aligned buffer before initiating the transfer. This transfer should proceed as a zero-copy operation on both the client and server.

For file read operations, the server can initiate the RDMA write transfer to the client as soon as the data is available on the server. Since the address of the destination buffer on the client is contained in the read operation header, this operation should proceed as a zero-copy operation on both the client and server. Since DAPL preserves message ordering, the server may send the read response message without waiting for an RDMA completion. If the RDMA transfer fails for any reason, the connection will be broken and the response message will not arrive at the client.

DAFS is further designed to take advantage of the characteristics of Direct Access Transports to address the overheads from the operating system, protocol parsing, buffer copying, and context switching, leaving the client free to run application code. Some of these techniques are discussed in Section 4, which covers the user-space DAFS client API. The

DAFS protocol enables asynchronous data access, further lowering client overhead. Where the option exists, DAFS is somewhat biased towards reducing the client overhead even if it slightly increases file server overhead. One example of this tradeoff is permitting clients to use their native byte order.

Batch I/O facility DAFS provides a batch I/O operation, analogous to list I/O in some operating systems, that lets the client combine a number of read and write transfer requests, each accessing distinct buffers, files, and file offset ranges via RDMA transfers. The batch semantics match well with high-performance I/O APIs like MPI–IO [14]. Batch requests may specify memory scatter/gather regions on the client and file scatter/gather regions on the server. These regions need not align. Conceptually, the data on each side of the wire maps into a contiguous buffer for the transfer, though the transport may implement the scatter operation directly. Clients maintain control over the I/O pipeline and buffer resources by specifying the size of each batch of completion notifications. Clients may request synchronous or asynchronous notifications. In the former case, the server returns a single reply once the entire batch has completed. In the latter, the server asynchronously notifies the client as results complete; the client can advise the server how many completions should be grouped together before sending a completion notification for that group.

Batch I/O also allows the client to advise a maximum latency target for the batch, allowing a DAFS server to perform write-gathering and other local throughput optimizations while maintaining client control of the overall throughput. This technique is well suited for prefetching and asynchronous cleaning of dirty buffers in sophisticated client buffer managers like those in database engines. The latency of operations in this context is often secondary to efficiency of the batch, so applications that can tolerate high latencies can profit overall from batching requests and receiving asynchronous, batched results. If, after issuing a high-latency batch, a client finds it necessary to wait for a specific request to complete, it may issue an *expedite* operation to request that the server immediately process the specified request.

Cache hints DAFS client *cache hints* allow the file server to make better use of its own cache. Cache hints are advisory. Clients can specify file-level policy and access pattern information, similar to *madvise()*, as well as provide positive and negative byte-range hints with each read and write operation. In the local file sharing environment, network latencies and DAFS CPU requirements are low enough that server cache can be considered an extension of the client or application cache. Hints provide a mechanism in a highly tuned system for the client and server to better manage cache hierarchy. In cases like large database deployments where the client cache dwarfs the server cache, hints may allow the server to function as an effective metadata or victim cache instead of uselessly shadowing a fraction of the application's local cache.

Chaining DAFS *chaining* is similar to compound operations in NFSv4. NFSv4 defines compound requests to allow multiple file operations in a single network exchange as a way to reduce the combined latency of a series of operations. DAFS defines chained operations with the same semantics as NFSv4, but transfers the component operations separately. This approach limits the size of the preallocated buffers required to receive messages on both the client and server, and it preserves the benefits of a very simple packet layout for marshaling and parsing requests.

Combined with a user-space DAFS client, chaining permits client engines to implement complex operations that map to sequences of DAFS protocol requests. A DAFS client can then issue such requests without intervening between each protocol operation, reducing application impact while still preserving the failure characteristics of the equivalent series of operations.

3.2 New File Sharing Features

DAFS adds specific semantic enhancements to NFSv4 for high-performance, reliable clustered applications. These semantic features generally fall into two categories: shared access, and recovery semantics. Shared access semantics include a fencing mechanism to support application sharing within clusters and a shared key mechanism to arbitrate per-file access among cooperating applications. Recovery semantics include additional locking mechanisms and exactly-once failure semantics that aid recovery

Rich locking semantics File locking is important for cooperating applications accessing a common file store. These applications may crash while holding locks or otherwise fail to properly release their locks before terminating. Limited to existing locking APIs in POSIX and Windows, NFSv4 provides time-based lease expiration on locks, allowing other clients to access the resource without requiring client recovery. However, in these situations, the state of the previously locked file is suspect: presumably the lock was obtained in order to create atomicity across multiple operations, but since the lock was not explicitly released, the series of operations may be only partially completed, possibly leaving the file in an inconsistent state. Some recovery action may be necessary before making use of the data in the file. DAFS includes richer locking semantics that address this shortcoming and provide greater utility to applications.

DAFS persistent locks provide notification of an error condition following abnormal (lease expiration) release of a lock. Earlier *lockd*-based locks persisted following the failure of a client holding a lock, until either the client recovered or manual administrative intervention was performed. Further, due to UNIX file locking semantics, NFS clients in practice release locks immediately upon any termination of the owning process. Persistent locks provide notification of abnormal lock release to any subsequent client attempting to lock the file, until the lock is specifically reset. DAFS autorecovery locks provide an alternative safeguard against data corruption by associating data snapshot and rollback mechanisms with the act of setting a lock or recovering from an abnormally released lock.

Cluster fencing In clustered application environments, cluster membership is governed by a cluster manager that ejects nodes suspected of misbehaving or having crashed. Such a manager requires a mechanism whereby cluster members that are ejected from the cluster can be prevented from accessing shared file storage. The DAFS *fencing* operations manage client fencing access control lists associated with files or entire file systems that are under the control of the cluster application. The cluster manager populates the fencing list with cluster members' names, allowing multiple cluster applications to run on the same nodes independently of each other. Updates to a fencing list cause the DAFS server to complete any in-progress operations on the file(s) and update the appropriate access control list before responding to the fence request. Subsequent file operations from cluster members whose access privilege has been removed are denied by the DAFS server. The session-based architecture maps very conveniently to the fencing model.

The fence operation provides a serialization point for recovery procedures by the clustered application, without interfering with other files on the DAFS server or other client applications. The DAFS fencing mechanism is independent of standard file access controls, and is designed to support cooperating cluster members, similar to NFS advisory lock controls.

Shared key reservations NFSv4 allows share reservations, similar to those of CIFS, as part of the *open* operation. Together with an access mode of *read*, *write*, or *read-write*, a deny mode of *none*, *read*, *write*, or *read-write* may be specified to limit simultaneous access to a given file to those uses compatible with that of the process doing the open. DAFS enhances NFSv4 reservations by also providing *shared key reservations*. Any client opening a file can supply a shared key. Subsequent opens must provide the same key, or they will be excluded from accessing the file. This provides a similar capability to fencing at the file level. Key reservations have the scope of a file open. The duration of enforcement of a shared key reservation is from the time the first open specifies the key, to the time the file is finally closed by all clients that opened the file with that key.

Request throttling The use of credit-based message transfers managed on a per-client session basis allows the DAFS server to dedicate a fixed amount of resources to receiving client requests, and constantly redistribute them among many connected clients depending on their importance and activity level. Clients may affect this by requesting more credits as their workloads increase or returning unneeded credits to the server.

Exactly-once failure semantics To reach parity with local file systems, DAFS servers may implement exactly-once failure semantics that properly handle request retransmissions and the attendant problem of correctly dealing with retransmitted requests caused by lost responses. Retry timeouts do not offer this capability [17]. Our approach takes

advantage of the DAFS request credit structure to bound the set of unsatisfied requests. The server's response cache stores request and response information from this set of most recently received requests for use during session recovery. Upon session reestablishment after a transport failure, the client obtains from the server the set of last-executed requests within the request credit window. It can therefore reliably determine which outstanding requests have been completed and which must be reissued. If the server's response cache is persistent, either on disk or in a protected RAM, then the recovery procedure also applies to a server failure. The result is to provide true exactly-once semantics to the clients.

Other enhancements The DAFS protocol includes a variety of additional data sharing features, including improved coordination for file append operations between multiple clients and the ability to automatically remove transient state on application failure. DAFS supports two common examples of the latter: creation of *unlinked files* that do not appear in the general file system name space, and *delete on last close* semantics, which provide for a deleted file that is currently being accessed to have its contents removed only when it is no longer being accessed.

3.3 Security and Authentication

The DAFS protocol relies on its transport layer for privacy and encryption, using protocols like IPSEC, and so contains no protocol provision for encryption. Since DAFS clients may be implemented in user space, a single host may contain many clients acting on behalf of different users. DAFS must therefore include a strong authentication mechanism in order to protect file attributes which depend on identifying the principal. For this reason, authentication is available in several extensible levels. For more trusted deployments, DAFS provides simple clear-text username/password verification, as well as the option to disable all verification. For environments that require stronger authentication semantics, DAFS uses the GSS-API [21] to support mechanisms such as Kerberos V [19].

4 The DAFS API

In addition to the DAFS protocol, the DAFS Collaborative defined the DAFS API, which provides a convenient programmatic interface to DAFS. The API provides access to the high-performance features and semantic capabilities of the DAFS protocol. The DAFS API is designed to hide many of the details of the protocol itself, such as session management, flow control, and byte order and wire formats. To the protocol, the DAFS API adds session and local resource management, signaling and flow control, along with basic file, directory, and I/O operations. The DAFS API is intended to be implemented in user space, making use of operating system functions only when necessary to support connection setup and tear down, event management, and memory registration. Through the use of kernel-bypass and RDMA mechanisms, the primary goal of the DAFS API is to provide low-latency, high-throughput performance with a significant reduction in CPU overhead. That said, a DAFS client may also be written as a kernel-based VFS, IFS, or device driver. Applications written to the POSIX APIs need not be modified to use those flavors of DAFS clients, but in general will lack access to advanced DAFS protocol capabilities.

The Collaborative did not attempt to preserve the traditional POSIX I/O API. It is difficult to precisely match the semantics of UNIX I/O APIs from user space, particularly related to signals, sharing open descriptors across *fork()*, and passing descriptors across sockets.[1] Other aspects of the POSIX API are also a poor match for high-performance computing [25]. For example, implicit file pointers are difficult to manage in a multi-threaded application, and there are no facilities for complicated scatter/gather operations. Unlike POSIX-based DAFS clients, the DAFS API allows the application to specify and control RDMA transfers.

The DAFS API is fully described in the DAFS API specification [8]; this section focuses on how the DAFS API provides inherent support for asynchronous I/O, access to advanced DAFS features like completion groups, registration of frequently-used I/O buffers, and improved locking and sharing semantics crucial for local file sharing applications. The DAFS API differs from POSIX in four signifi-

[1] One might say that 99% API compatibility doesn't mean 99% of applications will work correctly. It means applications will work 99% correctly.

cant areas:

Asynchrony The core data transfer operations in the API are all asynchronous. Applications written to use a polling completion model can entirely avoid the operating system. Asynchronous interfaces allow efficient implementations of user-space threading and allow applications to carefully schedule their I/O workload along with other tasks.

Memory registration User memory must be registered with a network interface before it can be made the destination of an RDMA operation. The DAFS API exports this registration to the application, so that commonly-used buffers can be registered once and then used many times. All of the data transfer operations in the DAFS API use *memory descriptors*, triplets of buffer pointer, length, and the memory handle that includes the specified memory region. The handle can always be set to NULL, indicating that the DAFS provider library should either temporarily register the memory or use pre-registered buffers to stage I/O to or from the network.

Completion groups The traditional UNIX API for waiting for one of many requests to complete is *select()*, which takes a list of file descriptors chosen just before the select is called. The DAFS API supports a different aggregation mechanism called *completion groups*, modeled on other event handling mechanisms such as VI's completion queues and Windows completion ports. Previous work has shown the benefits of this mechanism compared to the traditional *select()* model [2, 6]. If desired, a read or write request can be assigned to a completion group when the I/O request is issued. The implementation saves CPU cycles and NIC interrupts by waiting on predefined endpoints for groups of events instead of requiring the event handler to parse desired events from a larger stream.

Extended semantics The DAFS API provides an opportunity to standardize an interface to DAFS capabilities not present in other protocols. These include:

- Powerful batch I/O API to match the batch protocol facility. The batch I/O operation issues a set of I/O requests as a group. Each request includes a scatter/gather list for both memory and file regions. Batch responses are gathered using completion groups.
- Cancel and expedite functions. These are particularly useful for requests submitted with a large latency.
- A fencing API that allows cooperative clients to set fencing IDs and join fencing groups.
- Extended options at file open time, including cache hints and shared keys.
- An extensible authentication infrastructure, based on callbacks, that allows an application to implement any required security exchange with the file server, including GSS-API mechanisms.

5 Performance

This section presents experimental results achieved with our client and server implementation. The first results are micro-benchmarks demonstrating the throughput and client CPU benefits of DAFS. The second result is a demonstration that the DAFS API can be used to significantly improve an application's overall performance; in this case, *gzip*.

We have implemented the DAFS API in a user-space DAFS library (henceforth *uDAFS*). Our uDAFS provider is written to the VIPL API, and runs on several different VI implementations. We have also implemented a DAFS server as part of the Network Appliance ONTAP system software. The NetApp uDAFS client has been tested against both our DAFS server and the Harvard DAFS server [22, 24].

Our tests were all run on a Sun 280R client, connected to a Network Appliance F840 with 7 disks. We tested two GbE-based network configurations, one for DAFS and one for NFS. The DAFS network used an Emulex GN9000 VI/TCP NIC in both the client and file server; this card uses jumbo frames on Gigabit Ethernet as its underlying transport. The NFS network used a Sun Gigabit PCI adapter 2.0 card in the client and an Intel Gigabit Ethernet card in the server, speaking NFSv3 over UDP on an error-free network. The Sun adapter does not support jumbo frames. All connections on both networks are point-to-point.

5.1 Micro-benchmarking

Our first benchmark compares the cost of reading from a DAFS server to reads from an NFSv3 server. We show the results for both synchronous (blocking) I/O and asynchronous I/O. In each case we present two graphs. The first compares overall throughput for varying block sizes, while the second compares the CPU consumption on the client. We stress that these tests exercise the bulk data movement mechanisms of the protocols and their transports, not the higher protocol semantics.

In order to focus the measurements on a comparison of the protocols, these experiments perform reads of data that is cached on the server. Our uDAFS client implementation does not cache data and we configure the NFS client stack to do the same by mounting with the `forcedirectio` mount option. Such an arrangement is not contrived; database engines are typically run in this fashion as host memory is better allocated to the higher level database buffer cache instead of a kernel buffer cache that would merely replicate hot blocks, and potentially delay writing them to stable storage.

All of these results are gathered using a benchmarking tool that allows us to select either the POSIX or DAFS API for I/O, and to perform the I/O using either synchronous interfaces (*read()* and *dap_read()*) or asynchronous interfaces (*aioread(), aiowait(), dap_async_read(),* and *dap_io_wait()*). When using DAFS, the program first registers its buffers with the DAFS provider. All DAFS reads are done using DAFS direct reads, that is, using RDMA instead of inline bulk data. The tool includes built-in high-resolution timing hooks that measure realtime clock and process time.

Synchronous performance Our first test compares synchronous read operations for NFSv3 and DAFS. Figures 4 and 5 show the results. As expected, DAFS requires fewer CPU cycles per operation. Direct data placement keeps the line flat as block size increases, while the NFS stack must handle more per-packet data as the size increases. The latency of synchronous operations limits throughput at smaller block sizes, but once client overhead saturates the CPU, the DAFS client can move more data over the wire.

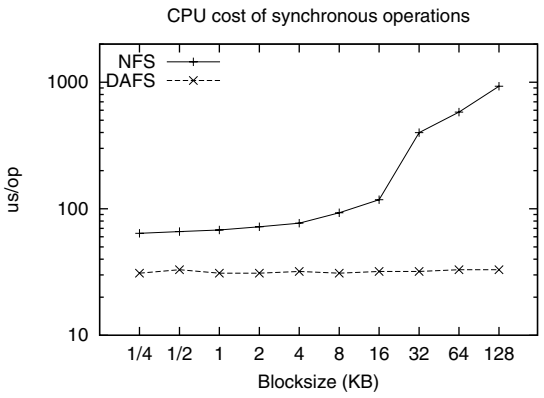

Figure 4: CPU time consumed per synchronous read request.

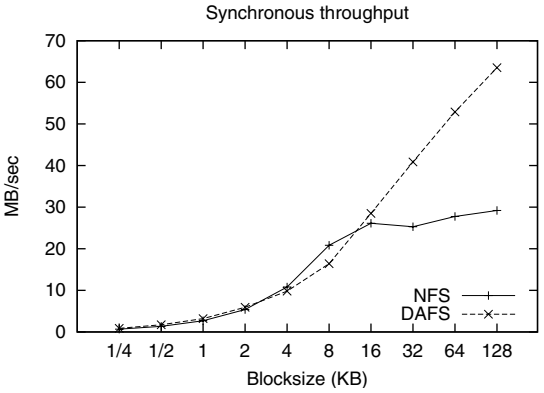

Figure 5: Synchronous throughput.

Asynchronous performance Figures 6 and 7 are a repeat of the previous tests, but use the asynchronous APIs to keep 32 reads in flight simultaneously. Here the advantages of DAFS become clearer. Asynchronous I/O improves throughput for both NFSv3 and DAFS, but the CPU time is the most significant result. While NFSv3 achieves high throughput only with an *increase* in CPU cost, DAFS requires *less* CPU time in asynchronous mode, since many results arrive before the client tries to block waiting for them. Asynchronous DAFS throughput approaches 85 MB/sec, matching our measured limit of 33 MHz PCI bus bandwidth on this platform.

5.2 GNU gzip

We converted the GNU *gzip* program to recognize DAFS filenames and use the DAFS API for access-

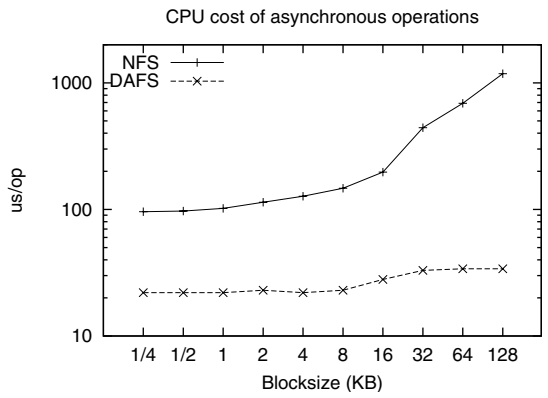

Figure 6: CPU time consumed per asynchronous read request.

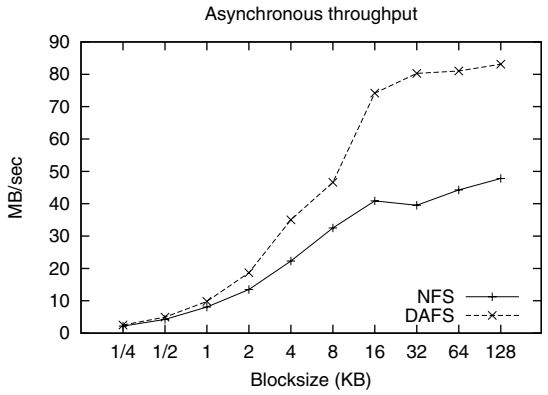

Figure 7: Asynchronous throughput.

ing those files. The conversion to the DAFS API adds memory registration, read-ahead, and write-behind capabilities. Figure 8 shows two comparisons. The test on the left measures the wall clock time of a single instance of *gzip* compressing a 550 MB file. Two factors account for the speedup. The *gzip* program uses 16 KB block sizes; Figures 4 and 6 show DAFS requiring 20 microseconds per operation at that block size, whereas NFSv3 consumes 100 microseconds. A 550 MB file corresponds to roughly 35,000 read operations, yielding nearly 3 seconds of CPU time saved just on the basis of read I/O cost. Moreover, during this test, the client CPU reported 0.4% idle time when running the DAFS version of *gzip* and 6.4% idle time when running the stock NFS version, accounting for a further 10 seconds of CPU time. By allowing asynchronous I/O without excessive CPU cost, the DAFS version hardly spends any time blocked waiting for data, so the CPU can spend more cycles generating the compressed blocks.

The second set of numbers compares the runtime of two *gzip* processes running in parallel, each operating on half of the full dataset. Here the uDAFS client demonstrates the advantages of operating system avoidance. The DAFS client achieves a nearly perfect 2X speedup, whereas the NFS versions are limited by kernel contention. In this case, both processors remained 100% busy while executing the DAFS version of *gzip*, but reported 34% idle time in the NFS case.

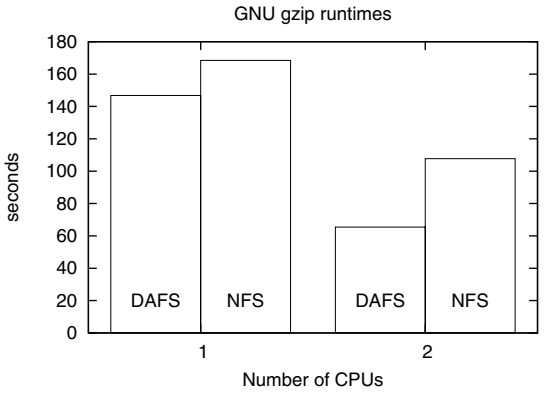

Figure 8: GNU gzip elapsed time.

6 Conclusions

The DAFS protocol enables high-performance local file sharing, and is targeted at serving files to clusters of clients. In this environment, the clients themselves are often application servers. DAFS takes advantage of Direct Access Transports to achieve high bandwidth and low latency while using very little client CPU. To support clustered environments, DAFS provides enhanced sharing semantics, with features such as fencing and shared key reservations, as well as enhanced locking. As an open client-server protocol, DAFS enables multiple interoperable implementations, while allowing for multi-protocol access to files on the server.

DAFS leverages the benefit of user-space I/O by providing asynchronous operations both in the protocol and in its application programming interface. DAFS shows significant measured performance gains over NFS on synchronous and asynchronous reads, and can yield a substantial performance improvement on I/O intensive applications, as demonstrated by our *gzip* experiment.

The unique combination of traits enabled by DAFS is extremely well-suited to the needs of local file sharing environments, such as data centers. The DAFS protocol is the basis for high-performance, scalable, and sharable network file systems that exploit current technology trends.

7 Acknowledgements

We'd like to acknowledge the contributions of many people from Network Appliance and all of the individuals representing the companies within the DAFS Collaborative in creating the DAFS protocol and API. We also wish to recognize the contributions of Somenath Bandyopadhyay, Tim Bosserman, Jeff Carter, Sam Fineberg, Craig Harmer, Norm Hutchinson, Jeff Kimmel, Phil Larson, Paul Massiglia, Dave Mitchell, Joe Pittman, Joe Richart, Guillermo Roa, Heidi Scott, and Srinivasan Viswanathan for their work on the file access protocol and API; Caitlin Bestler, Tracy Edmonds, Arkady Kanevsky, Rich Prohaska, and Dave Wells for their help in defining DAFS transport requirements; and Salimah Addetia, Richard Kisley, Kostas Magoutis, Itsuro Oda, and Tomoaki Sato for their early implementation experience and feedback.

We also thank Jeff Chase, our shepherd, and Mike Eisler for their valuable feedback.

References

[1] InfiniBand Trade Association. Infiniband architecture specification, release 1.0a, June 2001.

[2] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. A scalable and explicit event delivery mechanism for UNIX. In *USENIX Annual Technical Conference*, pages 253–265, June 1999.

[3] Matthias Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward Felten, and Jonathan Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 142–153, April 1994.

[4] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995.

[5] Greg Buzzard, David Jacobson, Milon Mackey, Scott Marovich, and John Wilkes. An implementation of the Hamlyn sender-managed interface architecture. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*. USENIX Association, October 1996.

[6] Abhishek Chandra and David Mosberger. Scalability of Linux event-dispatch mechanisms. In *Proceedings of the USENIX Annual Technical Conference*, pages 231–244, June 2001.

[7] Jeff Chase, Andrew Gallatin, and Ken Yocum. End system optimizations for high-speed TCP. *IEEE Communications Magazine*, 39(4):68–74, 2001.

[8] DAFS Collaborative. Direct Access File System: Application programming interface, October 2001.

[9] DAFS Collaborative. Direct Access File System protocol, version 1.0, September 2001.

[10] DAT Collaborative. uDAPL: User Direct Access Programming Library, version 1.0, July 2002.

[11] Compaq, Intel, and Microsoft. Virtual Interface Architecture specification, version 1.0, December 1997.

[12] Brian Pawlowski et al. The NFS version 4 protocol. *Proceedings of the 2nd international system administration and networking conference (SANE2000)*, page 94, 2000.

[13] Internet Engineering Task Force. Remote direct data placement charter.

[14] MPI Forum. MPI-2: Extensions to the message-passing interface, July 1997.

[15] Robert W. Horst. Tnet: A reliable system area network. *IEEE Micro*, 15(1):37–45, 1995.

[16] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, 1988.

[17] Chet Juszczak. Improving the performance and correctness of an NFS server. In *USENIX Winter Technical Conference*, pages 53–63, January 1989.

[18] Mitsuhiro Kishimoto, Naoshi Ogawa, Takahiro Kurosawa, Keisuke Fukui, Nobuhiro Tachino, Andreas Savva, and Norio Shiratori. High performance communication system for UNIX cluster system. *IPSJ Journal Abstract*, 41(12 - 020).

[19] John Kohl and B. Clifford Neuman. Internet Engineering Task Force, RFC 1510: The Kerberos network authentication service (v5). September 1993.

[20] Ed Lee and Chandu Thekkath. Petal: Distributed virtual disks. In *Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 84–93, October 1996.

[21] John Linn. Internet Engineering Task Force, RFC 2743: Generic security service application program interface, version 2, update 1, January 2000.

[22] Kostas Magoutis. Design and implementation of a direct access file system. In *Proceedings of BSDCon 2002 Conference*, 2002.

[23] Kostas Magoutis, Salimah Addetia, Alexandra Fedorova, and Margo Seltzer. Making the most of direct-access network-attached storage. In *Proceedings of the Second Conference on File and Storage Technologies (FAST)*. USENIX Association, March 2003.

[24] Kostas Magoutis, Salimah Addetia, Alexandra Fedorova, Margo Seltzer, Jeff Chase, Richard Kisley, Andrew Gallatin, Rajiv Wickremisinghe, and Eran Gabber. Structure and performance of the Direct Access File System. In *USENIX Technical Conference*, pages 1–14, June 2002.

[25] Paul R. McJones and Garret F. Swart. Evolving the Unix system interface to support multithreaded programs. Technical Report 21, DEC Systems Research Center, September 1987.

[26] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *Computer Systems*, 2(3):181–197, 1984.

[27] Sun Microsystems. Internet Engineering Task Force, RFC 1094: NFS: Network File System protocol specification, March 1989.

[28] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[29] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the First Conference on File and Storage Technologies (FAST)*. USENIX Association, January 2002.

[30] SNIA. Common Internet File System (CIFS) technical reference, revision 1.0. *SNIA Technical Proposal*.

[31] Robert Snively. FCP draft, revision 8.0. *T11 Document X3T11/94-042v0*, 1994.

[32] Steve Soltis, Grant Erickson, Ken Preslan, Matthew O'Keefe, and Thomas Ruwart. The Global File System: A file system for shared disk storage. In *IEEE Transactions on Parallel and Distributed Systems*, October 1997.

[33] Raj Srinivasan. Internet Engineering Task Force, RFC 1831: RPC: Remote Procedure Call protocol specification version 2, August 1995.

[34] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles (SOSP)*, pages 224–237, October 1997.