Filesystems for Network-Attached Secure Disks

Garth A. Gibson, David F. Nagle, Khalil Amiri,
Fay W. Chang, Howard Gobioff, Erik Riedel,
David Rochberg, and Jim Zelenka

July1997
CMU-CS-97-118

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

garth+nasd@cs.cmu.edu
http://www.cs.cmu.edu/Web/Groups/NASD

Abstract

Network-attached storage enables network-striped data transfers directly between client and storage to provide clients with scalable bandwidth on large transfers. Network-attached storage also decouples policy and enforcement of access control, avoiding unnecessary reverification of protection checks, reducing file manager work and increasing scalability. It eliminates the expense of a server computer devoted to copying data between peripheral network and client network. This architecture better matches storage technology's sustained data rates, now 80 Mb/s and growing at 40% per year. Finally, it enables self-managing storage to counter the increasing cost of data management. The availability of cost-effective network-attached storage depends on it becoming a storage commodity, which in turn depends on its utility to a broad segment of the storage market. Specifically, multiple distributed and parallel filesystems must benefit from network-attached storage's requirement for secure, direct access between client and storage, for reusable, asynchronous access protection checks, and for increased license to efficiently manage underlying storage media. In this paper, we describe a prototype network-attached secure disk interface and filesystems adapted to network-attached storage implementing Sun's NFS, Transarc's AFS, a network-striped NFS variant, and an informed prefetching NFS variant. Our experimental implementations demonstrate bandwidth and workload scaling and aggressive optimization of application access patterns. Our experience with applications and filesystems adapted to run on network-attached secure disks emphasizes the much greater cost of client network messaging relative to peripheral bus messaging, which offsets some of the expected scaling results.

This research is sponsored by DARPA/ITO through ARPA Order D306, and issued by Indian Head Division, NSWC under contract N00174-96-0002. Additional support was provided by NSF and ONR graduate fellowships. The project team is indebted to generous contributions from the member companies of the Parallel Data Consortium. At the time of this writing, these companies include Hewlett-Packard Laboratories, Symbios Logic Inc., Data General, Compaq, IBM Corporation, Seagate Technology, and Storage Technology Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any supporting organization or the U.S. Government.

ACM Computing Reviews Keywords: D.4.3 File systems management, D.4.6 Access controls, C.3.0 Special-purpose and application-based systems, B.4 Input/Output and Data Communications.

1. Introduction

Users are increasingly using distributed filesystems to access data across local area networks; personal computers with hundred-plus MIPS processors are becoming increasingly affordable; perhaps as a result of faster processors, both file and total dataset sizes are increasing; and the sustained bandwidth of magnetic disk storage is expected to exceed 30 MB/s by the end of the decade. These trends place a pressing need on distributed filesystem architectures to provide clients with efficient, scalable, high-bandwidth access to reliably stored data.

A variety of approaches for improving distributed filesystem scalability and storage management have been proposed. Direct transfer of data between storage subsystem and client machine eliminates the filesystem server machine as a bottleneck [Miller88, Long94, Drapeau94] and enables filesystems to stripe data over multiple storage subsystems to increase per-client and aggregate bandwidth [Berdahl95, Hartman93]. Fast packetized storage interconnects such as Fibrechannel [Benner96] promise that storage devices, including individual disk drives [Anderson95], can be cost-effectively attached to networks, and the proposed requirement for cryptographic support in version 6 of the Internet Protocol promises cost-effective network support for security [Deering95]. Exploiting the indirection provided by interface abstractions such as SCSI, storage subsystems attack self-management by virtualizing the storage media that filesystems appear to manage [Lee96, deJonge93, Wilkes95, StorageTek94]. Decoupling filesystem policy decisions, such as access protection or client cache consistency, from the datapath of individual accesses may reduce the work on a filesystem server, enabling increased scalability for existing, small-fetch workloads[Gibson97]. Taken collectively, these approaches define the range of network-attached storage.

To deliver their promise to the majority of cost-conscious computing environments, network-attached storage devices must adhere to a standard interface and exhibit large volume manufacturing; that is, network-attached storage must evolve from and inherit the SCSI market. For this to happen, network-attached storage must provide value in a wide range of environments, notably for applications of distributed filesystems common today.

In this paper we present a prototype for a network-attached storage interface, which we call network-attached secure disks (NASD), that exhibits most of the promise of this technology. We then describe and demonstrate NASD-based adaptations of two typical and significantly different, distributed filesystems, Sun's Network File System (NFS) [Sandberg85] and Transarc's Andrew File System (AFS) [Howard88]. Having shown NASD-based adaptations for existing filesystems, we proceed to describe and demonstrate systems extended to support network striping [Hartman93] and informed prefetching [Patterson95]. Our experiments show a cryptographically sealed capability scheme [Gobioff97] for securing communications and asynchronous authorization that has little impact on performance when securing only the integrity of command, status and authorization. Securing data integrity using software cryptography reduces bandwidth by 55%, encouraging the inclusion of hardware support for message digest to support this level of protection. Experiments with aggregate bandwidth as the number of pairs of clients and drives scales from one to four shows NASD's greater scalability; with four client/drive pairs, aggregate read bandwidths are 20% to 90% higher with NASD even though our prototype's peak RPC bandwidth is about 15% lower than the comparable existing filesystems'. For less data-intensive applications, the anticipated scaling from offloading read and writes to NASD drives is offset by the increased cost of communicating with the drive using a standard RPC interface instead of UNIX raw disk and the SCSI bus. Basically, network protocols do not deliver data over short distances very efficiently. Finally, we show that extending the NASD drive with informed prefetching and caching, a technique that depends on access hints from the application, can be done with only pass-through support from the client's NFS implementation and delivers a 45% reduction in execution time for an I/O-intensive application.

This paper begins with the definition of Network-Attached Secure Disks (NASD) and a description of our prototype implementation. Section 3 discusses the general task of building a filesystem on top of NASD and describes our four NASD-adapted filesystems. Our experimental environment, NASD and NASD-adapted filesystem implementations and results are reported in Section 4. Research results related to NASD and NASD-adapted filesystems are discussed in Section 5. Finally, we present concluding remarks and future work in Section 6.

2. Network-Attached Secure Disks (NASD)

A traditional server-attached disk (SAD) system transfers data via server memory, at which time it synchronously oversees rights and privileges. Such systems typically have two independent networks, the local area network and the storage bus, and all data travels over both. While storage subsystems used by SAD do implement some self-management and extensibility, both are limited by the SAD filesystem design. This section describes the properties of NASD that overcome these limitations and promise greater performance and flexibility.

While this paper presents all network-attached storage devices as though they were single disk drives, the interfaces and results apply equally well to larger storage subsystems such as RAID controllers or to other storage media. We will use the term *NASD drive* in the rest of this paper to include all such devices. The remnants of the SAD file server functions still required in a NASD environment are handled by what we call a *file manager.*

2.1. Properties of NASD

Direct transfer: Data accessed by a filesystem client is transferred between the NASD drive and the client without indirection (store-and-forward) through a file server machine. To be most effective, all high-performance clients should be addressable and accessible without store-and-forward routing. In switched networks, network striping of large datasets and direct transfers of large requests enables scalable file bandwidth.

Asynchronous filesystem oversight: Frequently consulted but infrequently changed filesystem policy decisions, such as what operations a particular client can perform on a particular set of stored data, are made by the file manager infrequently and enforced by the NASD drive during repeated client operations without synchronous recourse to the filesystem on each request. For the highly concurrent, small-access workloads typical of most office/campus distributed filesystems, asynchronous oversight offloads predetermined access checks and common command processing to parallel storage, improving the filesystem's ability to scale up the number of clients and amount of storage managed.

Cryptographic support for request integrity: Without synchronous filesystem oversight, even a second, physically secure private network between storage and the file manager machine is insufficient to ensure storage data integrity because data is transferred over an insecure network. With a single network, NASD drives must be capable of computing keyed digests on command and status fields without bandwidth penalty. This requirement is essentially the same as the security requirement in current proposals for IPv6 [Deering95]. With hardware support for cryptography in a drive, high-performance data integrity, data privacy and command privacy are also feasible.

Storage self-management: With asynchronous filesystem oversight, NASD drives have knowledge of logical relationships between storage units. Added to the diverse transparent manipulations of physical storage that storage subsystems already practice beneath the SCSI block address space abstractions, NASD drives should customize performance improvement strategies to their customers' data and access patterns and transparently maximize usable capacity and data reliability. Through such intelligence, NASD drives target reducing the cost of storage management, sometimes estimated to be three or more times larger than the total acquisition cost of storage devices [Lee96].

Extensibility: With asynchronous filesystem oversight and direct communication between clients and storage, NASD drives are uniquely qualified to offer extensions to be used by client applications with and without file manager involvement. Notably, application hints describing client access patterns can dramatically improve storage performance by prefetching from disk to cache or from tape to disk [Cao95]. Extension interfaces can also enable groups of NASD drives to provide more powerful storage self-management such as tolerance of single drive failures [Long94, Lee96].

2.2. NASD Interface Design

Grouping of storage: The set of storage units accessible to a client as the result of an asynchronous access control decision must be named and navigated by client and NASD. Because a NASD drive must enforce access control decisions, storage grouping information must be communicated to the device. Giving this grouping information a name allows a client to simplify its view of accessible storage to a variable length set of bytes, possibly directly corresponding to a file. Such names may be temporary. For example, Derived Virtual Devices (DVD) use the communications port identifier established by the file manager's decision to grant access to a group of storage units [VanMeter96a]. In our prototype interface, a NASD drive partitions its allocated storage into containers which we call *objects*.

Access control enforcement: Access control decisions made by a file manager must be enforced by a NASD drive. This enforcement implies authentication of the file manager's decisions. The authenticated decision authorizes particular operations on particular groupings of storage. Because this authorization is asynchronous, a NASD device may be required to record an audit trail of operations performed, or to revoke authorization at the file manager's discretion. For a small number of a popular authentication systems, Kerberos [Neuman94] for example, NASD drives could be built to directly participate, synchronously receiving a client identity and rights in an authenticated message from a file manager during its access control processing. DVD devices use this approach, which is simplified by the availability of authenticated RPC packages, also used by filesystems such as AFS [Satya89]. An alternative that avoids dependence on the local environment's authentication system and the synchronous rights-granting message from the file manager, is to employ capabilities based on one-way functions, similar to the ICAP [Gong89] or Amoeba [Tanenbaum86] distributed environments, transported to the device via a client but only computable/mutable by file manager and NASD drive.

Operation semantics: There is wide variation in the semantics of operations as simple as read and write. For example, Amoeba's Bullet filesystem will only access files in their entirety and will not overwrite an existing file [vanRenesse89]. By forcing contiguous allocation and sequential access, Bullet can achieve high data rates, but similar benefits can be derived from other clustering schemes such as lists of lists [deJonge93], extents [McVoy91], and logs [Rosenblum91]. Written data need not consume as many storage bytes as are written; storage may compress written data [Burrows92], replace long sequences of unspecified data with holes that read as zeros, or link one storage unit into two storage groupings sharing the same contents with copy-on-write semantics. More commonly, the storage consumed is larger than the written data because of indexing information, preallocation, or failure tolerating encodings [Patterson88]. Beyond simple read and write, the semantics of filenames and directories are particularly critical [Rosenblum91]. Because information sharing is often channelled through storage, lock synchronization is sometimes provided as a storage primitive. Moreover, some implementations provide atomic operations in the faces of failures, or allow transactions composed of multiple operations to be entirely undone [Mitchell82]. While NASD should leave as many of these optimizations to be developed by device designers as possible, most must be pinned down by a particular interface specification. The operations of our prototype NASD interface are specified in the next sections.

Partitions: When a NASD drive is used to provide storage for multiple file managers (or multiple instances of the same manager managing independent collections of storage), each manager should see an independent and disjoint subset, or *partition*, of the device's storage capacity. Very similar to the partitioning of a traditional disk in a UNIX system, a NASD partition is not expected to frequently change size, but, unlike a traditional partition, it is not necessarily a specific, static set of storage bytes. The partition abstraction, like AFS volumes or Petal virtual disks, can be used to dynamically integrate changes in storage capacity and to manage data reliability through on-line or off-line redundancy.

Clocks: Efficient, secure communications defeats message replay attacks by uniquely timestamping messages with loosely synchronized clocks and enforcing uniqueness of all received timestamps within a skew threshold of each node's current time [Neuman93]. Although all that is needed in a NASD drive is a readable, high-resolution, monotonically increasing counter, we further assume that this rate is controlled by a network time protocol such as NTP [Mills88] to ensure that all NASD drives possess a loosely synchronized real-time clock.

Extensions: NASD drives must provide interfaces that enable clients and filesystems to discover available extended functions. Implementors are responsible for ensuring that extensions do not circumvent the security policies specified by the file manager. For example, an extension that delivers data to a client must require that the client hold a valid capability for reading the given object. Where possible, implementors of a particular filesystem would make available simple interfaces for applications to invoke extended NASD interfaces. For example, following the model of SCSI pass-through, a filesystem would enable locally-available capabilities to be delivered to a NASD drive along with additional arguments.

Communications: Although logically specified as a generic remote procedure call (RPC) interface, communications protocols must be pinned down in the NASD interface for commodity devices. In our work we continue to assume generic RPC to date. However, as will be discussed later, the efficiency of a communications package is important to the overall performance of NASD-based filesystems and further study is required to completely specify a NASD protocol stack.

2.3. Prototype NASD Objects and Operations

Magnetic disk are fixed-sized block devices. Traditionally, filesystems are responsible for managing the actual blocks of the disks under their control. Using notions of the disk drive's physical parameters and geometry, filesystems maintain information about which blocks are in-use, how blocks are grouped together into logical objects, and how these objects are distributed across the device [McKusick84, McVoy91]. Current SCSI disks offer virtual or logical fixed-sized blocks named in a linear address space. This is virtual because modern disks already transparently remap storage sectors to hide defective media and the variation in track densities across the disk. By locating blocks with sequential addresses at the location closest in positioning time (adjacent where possible), SCSI supports the locality-based optimizations being computed by the filesystems' obsolete disk model. More advanced SCSI devices exploit this virtual interface to transparently implement RAID, data compression, dynamic block remapping, and representation-migration [Patterson88, StorageTek94, Wilkes95, Holland94].

In our NASD interface we abandon the notion that file managers understand and directly control storage layout. Instead, NASD drives store variable-length, logical byte streams called *objects*. Filesystems wanting to allocate storage for a new file request one or more objects to hold the file's data. Read and write operations apply to a byte region (or multiple regions) within an object. The layout of an object on the physical media is determined by the NASD drive. To exploit the locality decisions made by a file manager, sequential addresses in NASD objects should be allocated on the media to achieve fast sequential access. For inter-object clustering, NASD breaks from SCSI's global address space and adopts a linked list of objects where proximity in the list encourages proximity on the media, based on the Logical Disk model [deJonge93].

Tables 1, 2, and 3 present NASD operations on objects, partitions and drives. NASD objects have associated with them several attributes visible to file managers and clients enumerated in Table 5. Most attributes have values set externally, under control of the file manager's capabilities, or communicate information used internally by the NASD drive. For example, in a particular NASD drive, the *block_size* attribute may not be adjustable, or may only be adjustable during the `create` operation. Timestamps are set by the NASD drive according to its internal clock. Because NASD drives may implement storage management strategies that dynamically change an object's representation internally, the *attribute_modify_time* may not correspond to the time of any external operation on this object.

The *fs-specific* attribute is not interpreted by NASD and is made available for filesystem book keeping. For instance, a UNIX filesystem representing files as individual NASD objects could store owner, group, and permission bits in the *fs-specific* attribute. The idea of binding arbitrary attributes to storage objects is not new, for instance, AtFS has a similar notion [Lampen91]. The *fs_data_modify_time* and *fs_attribute_modify_time* are updated by NASD when data or attributes are modi-

Operation	Arguments	Return Values	Description
create	partition, inital attribute	new identifier and attribute, status	create a new object on the specified partition, optionally setting initial attributes
remove	partition, identifier	status	remove an object from a partition
getattr	partition, identifier	attribute, status	get attributes of an object
setattr	partition, identifier, attribute	new attribute, status	change a portion of an object's attributes, retrieving current attributes when complete
read	partition, identifier, regions	data, length, status	read from a strided list of scatter-gather regions
write	partition, identifier, regions, data	length, status	write to a strided list of scatter-gather regions
fastcopy	partition, source identifier	new identifier and attributes, status	efficiently clone an object
flush	partition, list of identifiers	status	commit any cached writes to stable store
blocksrequired	partition, identifier, regions	estimated blocks required, maximum blocks required, status	determine how many additional blocks are required to store an object after completely overwriting indicated regions within the object

Table 1: NASD Object Operations (all operations require indication of which key to use, which protection options to apply, capability, nonce, and digest and return nonce and digest values)

Operation	Arguments	Return Values	Description
createpartition	partition	status	create a new partition (zero-sized)
removepartition	partition	status	remove a partition
resizepartition	partition, new size	status	set a partition's size
flushpartition	partition	status	commit any cached writes for a partition to stable store
setpartitionkey	partition, key name, key value	status	set "master" key for a partition

Table 2: NASD Partition Operations (all operations require indication of which key to use, which protection options to apply, capability, nonce, and digest and return nonce and digest values)

Operation	Arguments	Return Values	Description
initialize	new master keys, clock value and skew, master key	status	sets master keys, sets time, invalidates all other keys, clears partitions
setdrivekey	new drive key, master key	status	sets primary control key for a drive
inquiry		drive information	retrieve basic drive information (vendor, model, existing partitions)

Table 3: NASD Drive Operations (all operations require indication of which key to use, which protection options to apply, capability, nonce, and digest and return nonce and digest values)

Name	Description
Drive Control	basic control information for drive: clock, physical parameters, extensions supported, bytes allocated
Partition Control	basic control information for partition: current size, byte usage, number of object supported, number of objects allocated
Partition Contents	read-only list of identifiers of NASD objects allocated in the partition
First Object	ordinary, read/writable NASD object which is always created with length 0 when a partition is initialized

Table 4: Well-known NASD Objects

Type	Name	Set by	Semantics
Access Control	access_control_version	setattr	changing ACV logically revokes capabilities
Clustering	nearby_object	setattr	locate this object near 'nearby_object'
Cloning	copied_object	NASD	object was created as fastcopy of 'copied_object'
Size	logical_size	setattr, write	largest offset written
	blocks_allocated	NASD, setattr	number of blocks reserved for this object
	blocks_used	NASD, write	number of blocks used to store object's data and metadata
	block_size	NASD, setattr	number of bytes per block
Time	create_time	NASD	timestamp at time of object creation
	data_modify_time	NASD	timestamp at time of last data modification
	attribute_modify_time	NASD	timestamp at time of last attribute update
Filesystem	fs_data_modify_time	NASD, setattr	timestamp at time of last data modification; modifiable
	fs_attribute_modify_time	NASD, setattr	timestamp at time of last attribute update; modifiable
	fs_specific	setattr	256 bytes uninterpreted by NASD

Table 5: NASD Object Attributes

fied, but a `setattr` operation can change these values. This feature allows a filesystem to achieve synchronous modification timestamps without synchronous oversight and without surrendering the ability to adjust timestamps. There is no need for a *fs_create_time* because this can be maintained by the filesystem in the much larger *fs_specific* attribute.

While most NASD operations apply to a single object, or independently to a set of objects, the `fastcopy` operation creates and returns a new object whose data is an identical copy of a existing object and whose *copied_object* attribute names the original object. Other attributes of the new object default to the equivalent in the copied object, but NASD-maintained fields are set in accordance with `create` operation semantics. When implemented with copy-on-write representation of the duplicated data, this operation supports efficient snapshots for checkpoint and backup. NASD-embedded compression similarly introduces a possibly temporary reduction in space needed to store an object. In both of these cases *blocks_used* is "honest"; for copy-on-write, the sum of the *blocks_used* attributes of the two objects is the total space needed to represent both. NASD drives allow filesystems to ensure that an object is able to grow to its full size through the `blocksrequired` operation and the *blocks_allocated* attribute. The `blocksrequired` operation reports the most likely and maximal number of blocks that would be added to the representation of an object if a region of its existing contents were completely overwritten with arbitrary data. For example, a NASD drive might implement transparent object compression on a copy-on-write represented object: the estimated increase in *blocks_used* may assume that the compression ratio of the new data approximates that of the old data, but the new data may compress substantially less well (e.g. if a text file is overwritten with binary data).

NASD partitions have four objects with well known names, *Drive Control*, *Partition Control*, *Partition Contents*, and *First Object*, that are created with each partition and described in Table 4. The control objects are modelled after SCSI's mode sense and select operations; they maintain drive global and partition global information in a published format. *Drive Control* describes the drive's physical parameters (magnetic, NVRAM, and RAM capacity, etc.), number and size of partitions, clock parameters and value, and interface extensions supported. There is only one drive control object per drive. *Partition Control* is unique in each partition and describes the partition's resources (bytes in use, bytes available, object descriptors in use, object descriptors available, etc.). The *Partition Contents* object is distinct in each partition and contains an array of object names, one for each object in use in that partition, for use by filesystem recovery mechanisms. The *First Object* allows filesystems to locate an object with a well-known name that they can use as a starting point for filesystem-specific state (e.g. for filesystem consistency checks on startup).

2.4. Prototype NASD Capabilities and Secure Communication

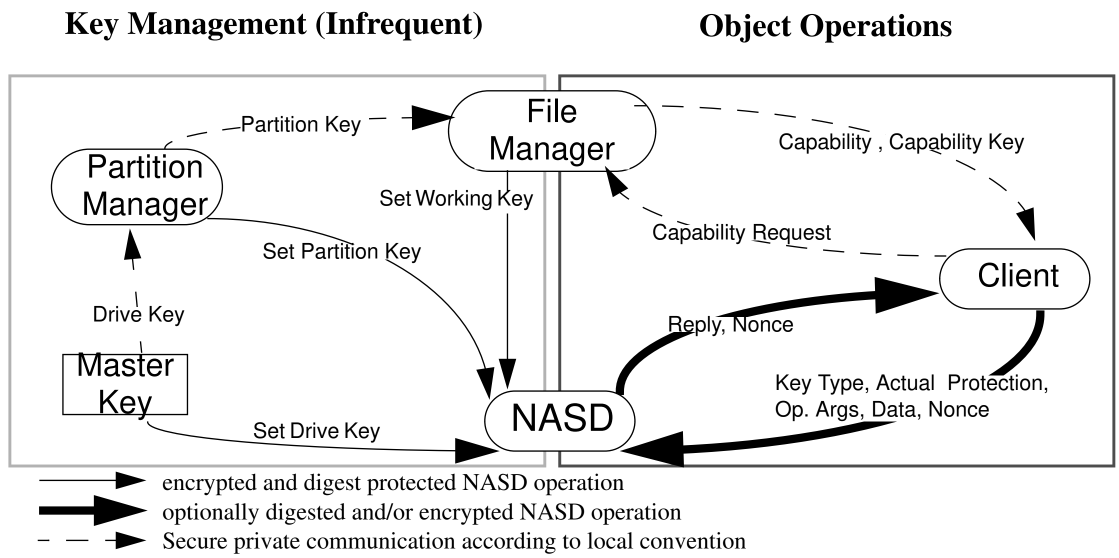
NASD efficiently supports asynchronous oversight (access control) and secure communications using capabilities that have been cryptographically sealed by the appropriate authority (filesystem manager or partition manager). This approach depends on a long-term relationship, based on a shared secret, between a NASD drive and the appropriate authority. NASD protects this relationship with a four-level key hierarchy as shown in Table 6. The master key is the longest-lived secret; in highly secure models it will be immutable once set and not recorded online outside of the NASD drive. It is used very infrequently to establish a new drive key in the event of failures or desynchronization.

The drive key is a long-term, online key used to manipulate NASD partitions and changed when compromise is feared or when the partition manager and drive have become desynchronized. The drive key, in turn, establishes new partition keys for each partition. Assuming a filesystem manages a partition, the partition key is the strongest and longest-term key available to the filesystem. It is used to establish a pair of working keys for each partition that are employed for the frequent act of generating capabilities. A working key will be changed regularly, perhaps daily, to limit the use and exposure of the key. In addition to the current working key, the previous period's working key is maintained to prevent the regular change of a working key from rendering all outstanding capabilities invalid[1]; that is, the second key allows capabilities to gracefully expire. The key hierarchy, clock, and a per-object *access_control_version* are the only long-term state necessary for security and only the key hierarchy must be kept private. All the other information relating to capabilities can be generated on the fly.

Figure 1 shows the use of the key hierarchy to protect the relationship between NASD drive and file manager and the much more common capability-authorized communication between clients and drives. On request, a file manager creates a capability (capability_args and a secret capability_key) for a client and delivers it privately using the local environment's secure communications system. The capability key is computed by the file manager as a keyed digest of fields in the capability and state on the drive. Because the current working key, a secret shared only by the file manager and drive, is used to construct capability keys, a client holding the capability cannot modify it without invalidating the capability key.

Requests to a particular NASD drive are authorized by demonstrating that the appropriate capability is held by the requester; specifically, by computing and transmitting the capability and appropriate keyed digest. Figure 1 also shows the computations performed by a NASD drive on which a request's authorization rests. All of the data used in the drive's authorization check is transmitted with every request, except for the on-drive copy of the drive's minimum protection options (see Table 7), the object's *access_control_version* attribute, and the secret working keys. Because the NASD's on-drive copy of the

1. If desired, changing the working key twice immediately can achieve this effect.



An operation(key_type,actual_protection,capability_args,args,data,nonce,digest) passes the digest test if and only if

digest == $\text{MessageDigest}_{key}$(args, nonce) when `INTEGRITY_ARGS` is specified OR

digest == $\text{MessageDigest}_{key}$(args, data,nonce) when `INTEGRITY_ARGS` & `INTEGRITY_DATA` are specified

where

key =	master key, drive key, partition key, working key, or capability_key (depending on key_type)
capability_args =	drive_name, rights, expiration, audit_identifier, minimum_protection_options, partition, object, region
capability_key =	$\text{MessageDigest}_{working\_key}$(drive_name, partition, object, region, access_control_version, rights, expiration, audit_identifier, minimum_protection_options)

Figure 1: Protection of operations with the key hierarchy. The left part of the figure describes the operations that manage the key hiearchy while the right describes the more frequently used object operations. The lower portion describes the mechanism to verify that a request is correctly bound to a capability or a member of the key hiearchy.

Name	Storage; Use	Frequency of Use
master key	offline; by drive owner or site administrator	used only to set drive key
drive key	online; by drive administrator (partition or backup manager); manipulate partitions, set partition keys, and optionally for any object operation	minimal
partition key	online; by filesystem manager (or equivalent); set working keys and optionally for any object operation	minimal
current working key	online; by filesystem manager; basis for capability-generation and optionally for object operations	high
previous working key	online; by a filesystem manager; basis for capabilities generated under the previous working key	high

Table 6: NASD Key Hierarchy

Protection Option	Description
NO_PROTECTION	digest values are not used or checked
INTEGRITY_ARGS	digest includes only command, status control field, and nonce, not transferred data
INTEGRITY_DATA	digests include transferred data
PRIVACY_ARGS	command, status control field, and nonce are encrypted
PRIVACY_DATA	transferred data is encrypted
PRIVACY_CAPABILITY	capability is encrypted under the working key

Table 7: NASD Protection Options

object's *access_control_version* is used in the authorization digest computation, a file manager can synchronously revoke all of the currently distributed capabilities for a particular object by changing this attribute. Similarly, if the file manager raises the drive's minimum protection options, then low protection capabilities will be synchronously invalidated. Because there are two working keys at any time, the drive's authorization check may be done twice if the associated capability key is older than the current working key.

In addition to checking that the proper message digest was received from the client, the NASD drive performs the following checks, based on capability and request fields:

- actual protection options specified by client meets the capability's minimum protection options;
- the capability's minimum protection level meets the drive's minimum protection options;
- the nonce (timestamp) value falls within a small window around the current time;
- the nonce has not been seen previously;
- current time is less than the capability's expiration time;
- the capability's drive name agrees with the NASD drive's internal name[1];
- capability's rights authorize requested operation; and
- the object region specified is a proper subregion of the region in the capability.

The drive's reply to an authorized request will be protected by the same protection as the request, use the same capability key, and include a freshly-generated nonce. As an optimization, a drive may maintain a cache of capabilities and their associated keys. This avoids recalculation of the message digests required to produce a capability key when multiple requests that use the same capability occur in quick succession.

A NASD drive can employ a wide spectrum of measures to protect the integrity of its critical state. For instance, a drive could utilize a tamper resistant memory to store the secrets or a secure coprocessor to protect computation [Tygar95]. Ultimately, the security of the system will rest on how well both the drive and the file manager protect the key hierarchy.

3. NASD-Adapted Filesystems

The primary purpose of a distributed filesystem is to provide distributed access to shared storage organized under the abstraction of a shared name space. The fundamental operations supported by all distributed filesystems are operations for storing and retrieving data, and namespace operations. Traditionally, this is accomplished by having a file server handle all client operations, shielding clients from low-level storage details and synchronously enforcing filesystem-specific policies (e.g. access control, cache consistency, and quotas). However, the file server is a single, shared resource and can become a bottleneck on client performance. As discussed in Section 2, a NASD environment is a natural substrate for building distributed filesystems with improved performance and scalability because it supports the separation of storage operations from namespace manipulations and filesystem policy decisions. To demonstrate this, we have ported two popular distributed filesystem, NFS and AFS, to our NASD environment. We have also implemented two NASD-enabled extensions to NFS, striped NFS and informed-prefetching NFS. All but the last of these are transparent to applications running on client machines.

In a NASD-adapted filesystem, files and directories are stored in NASD objects. To support direct client to drive data requests, clients must obtain information mapping files to NASD objects on NASD drives, and offsets in files to offsets in objects. For our NFS and AFS ports, each file and each directory occupies exactly one object, and offsets in files are the same as offsets in objects. This allows some typical file attributes (e.g. length and last modification time) to correspond directly with NASD-maintained object attributes (i.e. *logical_size* and *fs_data_modify_time*, respectively). The remainder (e.g. owner and mode bits) are stored in the *fs-specific* attribute. Traces of NFS and AFS filesystems indicate that the majority of client requests are reads of file attribute information, so it is important that these requests be satisfied by direct client to drive communication. Clients can determine the attributes for a file by obtaining the corresponding objects' attributes from a NASD drive.

As discussed in Section 2.4, a file manager in a NASD environment uses the capability mechanism provided by NASD drives to restrict client access to filesystem storage. Since file managers can perform authentication and authorization processing before issuing a capability, they can issue capabilities which enable direct client to drive requests for all operations that, in traditional filesystems, use the file manager mainly to provide access control (store-and-forward). Thus, clients can be permitted to directly read object data and attributes, and write data in objects which correspond to files, and clients should cache these capabilities to avoid involving the file manager in every request. However, file managers generally withhold capabilities which would allow clients to perform operations that, in traditional filesystems, involve the file server for other reasons (e.g. for preserving the integrity of the filesystem structure, or to enforce filesystem-specific policies other than access control). Thus, clients must continue to direct requests for directory and attribute modifications to the file manager, and, as a side-effect, the file manager must be involved in all operations which require modifications of file attributes which are stored in *fs-specific* object attribute.

---

1. ANSI X3T10 has proposed 64-bit, worldwide unique, device names to be administered by IEEE NAA.

In a SAD environment, all client data requests are handled by the file server, which maintains a single, large, shared data cache. In a NASD environment, data requests go directly to NASD drives, so data caching should be performed at the drives. The aggregate caching resources of the filesystem need to be partitioned amongst multiple entities, which amplifies the problem of disk hotspots [Kim86] if client data requests are not load-balanced across drives. However, striping with a fine granularity distributes localized accesses [Livny87]. Using AFS workload traces, we performed a simple cache simulation to estimate the impact of splitting the data cache. In our simulation of a NASD environment, files are striped (with a 64KB stripe unit) across NASD drives, and the aggregate data cache is split between a small (8 MB) file manager cache used only for directory data and some number of NASD drives. Varying the total cache sizes over a range from 64 MB to 96 MB, the simulation reveals that a split cache (with 4 or 8 drives) experiences a less than 10% decrease in cache hit rate compared to a SAD file server with the same aggregate cache size. In these simulations, the SAD file server achieved a hit rate ranging from 46% to 54%, which is consistent with past file server caching studies [Baker91]. However, this comparison is unfair to NASD since on-disk caches in current SAD servers are not used effectively (all data except read-ahead blocks are redundant because they are also cached in the file server), but would be effectively used in NASD drives. If we include the 1 MB to 2 MB on-disk caches available in current high-end disk drives [Seagate96a] when measuring the data cache of a SAD file server, the hit rate difference between SAD and NASD configurations would decrease.

3.1. NFS in a NASD environment

NFS's simple distributed filesystem model of a stateless server, weak cache consistency, and few mechanisms for filesystem management makes it easy to port to a NASD environment. In our implementation, the file manager distributes NASD mapping information by encoding the drive identifier and object identifier in the NFS file handle. Data-moving operations (`read`, `write`) and attribute reads (`getattr`) are redirected to the NASD drive while all other requests are handled by the file manager. Capabilities are piggybacked on the file manager's response to `lookup` operations.

3.2. AFS in a NASD environment

To demonstrate that the NASD interface can support more complex, distributed filesystem personalities, we also ported AFS to a NASD environment. We expanded AFS file identifiers to include a NASD drive and object identifier. As with NFS, data-moving requests (`FetchData`, `StoreData`) and attribute reads (`FetchStatus`, `BulkStatus`) are redirected to NASD drives, while all other requests are sent to the file manager. Because AFS clients perform lookup operations locally by parsing directory files, we added a new RPC to the AFS file manager interface allowing clients to obtain capabilities explicitly.

Implementing AFS on the NASD interface provides some challenges, specifically in maintaining the sequential consistency guarantees of AFS, and in implementing volume quotas. AFS's sequential consistency guarantee is that a client's store operation does not complete until all other clients holding a callback on the object being changed are notified of the change.

In a NASD environment, a client `StoreData` is encapsulated in a NASD/AFS request which does not succeed until the client has completed all direct-to-drive writes and has notified the file manager that writes are complete and callbacks should be broken. Should a client incorrectly exit before this notification, the file manager can limit the time until callbacks are broken by issuing write capabilities with short expiration times. Before sending callback breaks to other AFS clients, the file manager insures that the write capability is not valid, possibly explicitly revoking it by advancing the *access_control_version*.

Quota restrictions are handled in a similar manner. If a client requests that the file manager allow it to write past the current end of the object, the file manager escrows the appropriate number of bytes against the AFS volume quota for the duration of the capability's lifetime and constructs a capability which enables writing to the object with maximum offset equal to the escrowed size of the object. When the capability is returned to the file manager (by either write-complete notification or capability expiration), the file manager examines the object to determine its new size. It then updates its actual volume quota consumption total, as well as the maximum amount of unconsumed quota escrowed to valid write capabilities. If all unconsumed volume quota is escrowed against future object-extending writes when a client requests another escrow against the volume, the file manager may block the capability-request operation until a write capability issued earlier is returned or expires (it could also revoke escrow-holding capabilities, update actual space usage and escrow less space when revoked capabilities are reacquired). Once a volume is full, the file manager will reject requests to extend files within the volume.

3.3. Striped NFS in a NASD environment

Current distributed filesystems exhibit poor scalability due to the inherent limitation of the file server's total transfer bandwidth. In a NASD environment, striping files across NASD drives enables scalable high bandwidth to a single client, as well as substantially higher aggregate bandwidth to multiple clients. To demonstrate the potential of striping in a NASD environment, we implemented a striped NFS prototype in which the striping is transparent to the NASD drives. Striping can be implemented with or without the knowledge of the file manager. In a file manager-aware design, the file manager maintains the stripe map for a striped file and returns the map and the necessary capabilities to the client during the response to a `lookup` request. We chose not to implement this design because it integrates the management of striping and fault tolerance in the file manager, which limits transparency and flexibility. Instead, our design encapsulates striping in a separate striping manager, which exports a NASD interface to the file manager.

Our implementation uses the (unmodified) NASD/NFS file manager and two new entities: *striping managers* and *client clerks*. Striping managers look like "virtual" NASD drives to both the file manager and the client filesystems. The file manager creates a striped file by sending a `create` to the virtual drive exported by the striping manager and authorizes clients to access objects on these virtual drives. The striping manager handles allocation of striped files, construction of capabilities to authorize access to the objects which correspond to portions of striped files, and reconstruction of failed drives if necessary. The client filesystem accesses striped files through the client clerk which contacts the striping manager to obtain mapping information and capabilities and issues requests to NASD drives on the client's behalf. It also handles error-recovery and maintains a cache of mapping information. In our prototype, file data is striped but directories are stored in single objects, as in NASD/NFS. The portions of a striped file which reside on a single NASD drive are contained in a single object, and the attributes for a striped file, including the aggregate length and last modification time, are maintained by the striping manager.

3.4. Extensions to NASD

Extensions in our NASD prototype are described by their numeric RPC operation code. The interface provides a well-known mapping of an operation code to an area of the *Drive Control* object that describes the extension. Drives also enumerate the extensions they support at a well-known location in the *Drive Control* object.

For applications to directly take advantage of extensions, they must be able to resolve file names and offsets in the client namespace to NASD drives, partitions, objects, and offsets. They must also be able to acquire capabilities for those objects and send RPCs. To this end, we propose a new client-manager interface for filesystems built on NASD. Similar to SCSI pass-through, the application specifies an extension type, a list of files and offsets, and extension-specific data. The filesystem code on the client resolves the names and offsets to NASD drives, partitions, objects, and offsets, and then forwards the specified RPC to the drives with extension-specific data as arguments.

4. Experiments

Our experimental testbed contains four NASD drives, each one a DEC Alpha 3000/400 (133 MHz, 64 MB, Digital UNIX 3.2g-3) with a single 1.0 GB HP C2247 disk. We use four DEC Alpha 3000/400s as clients. For the NFS tests, both the SAD server and NASD file manager are run on a DEC Alpha 3000/500 (150 MHz, 128 MB, Digital UNIX 3.2g-3). The SAD configuration uses four HP C2247s attached to four individual busses. The AFS tests utilize a DEC AlphaStation 255 4/233 (233 MHz, 128 MB, Digital UNIX 3.2g-3) with four HP C2247s on a single bus. All these machines are connected by a 155 Mb/s OC-3 ATM network (DEC Gigaswitch/ATM).

Our experiments compare the performance of the original SAD file servers with direct-attached SCSI drives against the same machine acting as a NASD file manager and a comparable number of prototype NASD drives. Except where noted, workload is proportional to client-disk pairs and each client accesses files on a separate disk. To ensure consistent and meaningful results, before each iteration of each experiment, all drive, file manager, and client caches were flushed. Total data moved and run time are our primary metrics. Additionally, we measure CPU utilization by recording time spent in the operating system idle loop. All experiments were executed five times, and means and standard deviations are reported.

We use several benchmarks to evaluate our prototype filesystems: the Andrew benchmark [Howard88], a raw read bandwidth test, 'agrep' full-text search over a large number of small files or a small number of large files, and gnuld linking several hundred object modules into an 8 MB Digital UNIX kernel. The raw bandwidth test measures the sequential read of a single 15 MB file. The Andrew benchmark measures several different access patterns intended to represent a software development workload. The agrep benchmark sequentially searches each file for a pattern that will never be found. The gnuld benchmark requires the linker to read medium-size object files nonsequentially, and outputs the resulting binary into a single large file.

In all tests, file managers, clients, and drives have their caches enabled. To ensure fair comparisons between SAD filesystems and NASD file managers and drives, we fix the total memory available to cache the tests' data. In our largest configuration, the total cache available to the SAD file server was 64 MB across four SCSI drives, so smaller configurations were run with 16 MB of cache per client-drive pair. For NASD, the total cache space was the same as for the comparable SAD tests, except that cache space is partitioned between file manager and drives. In our runs, file manager cache space was set to 2 MB per client-drive pair and drives were each equipped with 14 MB caches. In all cases, client caching was identical; Digital NIX dynamically partitions the 64 MB client memory between operating systems, application virtual memory and file cache.

4.1. Prototype NASD Drive

We have implemented a working prototype of a NASD drive running as a kernel module in Digital UNIX. The prototype uses DCE RPC 1.0.3 over UDP/IP for its communications layer, and a modified UFS to store objects on disk. Non-filesystem-specific attributes that are not already tracked by UFS (such as the *access_control_version*, *object_creation_time*, etc.) are added to UFS on-disk inode (dinode). To record *fs-specific* attributes, a single UFS file contains a concatenation of all objects' *fs-specific* attribute. Incoming RPCs are dispatched by kernel threads which use low-level interfaces to this object storage system to accomplish their designated tasks. Our prototype NASD drive does not implement all of the interfaces specified in Section 2.3. Specifically, it does not yet implement support for partitions, fast-copy, well-known objects, privacy protection levels, key management, or inter-object clustering. None of these features are used by our benchmarks or NASD-adapted filesystems, except possibly implicit inter-object clustering, whose absence should only be pessimistic to NASD. While our prototype implements `INTEGRITY_ARGS` and `INTEGRITY_DATA`, unless explicitly stated, all numbers cited were collected with `NO_PROTECTION` in order to provide functionality comparable to the existing filesystems.

4.2. NASD/AFS

We have implemented a client and server for NASD/AFS on Digital UNIX and AFS 3.4a. NASD/AFS stores AFS files in single NASD objects with the AFS FID (file identifier) constructed from the NASD identifier and the identity of the NASD drive on which the corresponding object is stored. Some files are stored locally on a file manager's disk. Specifically, the root object for each AFS volume and the volume index files are maintained on the file manager's local disk. NASD/AFS objects use their *fs-specific* NASD attribute to hold the AFS `VnodeDiskData` structure, which includes such information as the file's owner, group, unix mode bits, and the AFS notion of the modification time.

This implementation also entailed a few simple modifications to the Transarc AFS code. File and directory creation operations were modified to return a capability for accessing the newly-created object, as well as the object FID. Three new RPCs were added: `GetCapability`, `GetWCapability`, and `ReturnCapability`. `GetCapability` is called by a client to obtain a read and getattr capability whose lifetime is equal to the unexpired time of the client's AFS token. `GetCapability` also registers a callback. `GetWCapability` returns a limited-duration capability which allows the client a reasonably small window of opportunity to write an object. `ReturnCapability` indicates that a client is done writing an object, the write capability returned may be revoked, and outstanding callbacks on the object may be broken.

As Table 8 shows, times for the various components of the Andrew benchmark are roughly 10% for SAD over NASD/AFS. Because these tests have no parallelism, no contention and we deliberately make no attempt to tune our NASD/AFS more than our SAD implementation, we do not expect to run any faster. Moreover, because of our experimental design consists of regular workstations with SCSI disks representing NASD drives, the RPC communication costs of all operations mediated by the file manager suffer extra overhead. Our raw bandwidth benchmark reads a large file sequentially in 512k chunks. Figure 2a shows that, as the number of client/disk pairs increase, the NASD configuration is able to linearly scale up the aggregate transfer bandwidth, while the SAD configuration is limited by the throughput of the AFS server. In this figure, NASD/AFS bandwidth is about 14% lower with 1 and 2 client-drive pairs. This results from a different RPC package being used in SAD (Rx) than NASD/AFS (DCE RPC).

While the raw read benchmark demonstrates the improved scalability of NASD over SAD for data transfer, it does not reflect performance in a real workload. For this we use the agrep and gnuld benchmarks. Our agrep test reads many small files sequentially and must perform directory operations to traverse a multilevel tree to locate each file. Figure 2b shows that, for this workload, NASD/AFS also offers 20% lower runtime relative to SAD at four client-drive pairs.

The gnuld benchmark offers a richer set of activity than either of these benchmarks. To complete its task, it must read a large number of object files of varying size in a nonsequential manner, and write a single large output file. The amount of computation required is considerable, so the workload is not a continuous stream of I/Os, but bursts of filesystem activity interleaved with periods of computation. Figure 2c demonstrates that NASD scales more effectively than SAD on this workload running in 30% less time in the four client-drive configuration.

4.3. NASD/NFS

In the NASD/NFS implementation, translation of NFSv3 to NASD operations is done at the RPC level with minimal changes to the existing NFSv3 client code. Requests that would normally go to a single NFS server are simply redirected to the

Operation	Description	NFSv3 seconds	NASD/NFS seconds	AFS 3.4a seconds	NASD/AFS seconds
Phase 1	creating directories	2 (0.0)	2 (0.5)	1 (2.5)	1 (0.0)
Phase 2	copying files	9 (1.0)	9 (0.4)	6 (35.6)	8 (29.6)
Phase 3	recursive directory stats	6 (0.6)	9 (0.5)	3 (5.1)	3 (2.0)
Phase 4	scanning each file	9 (0.4)	10 (0.5)	5 (2.0)	5 (3.7)
Phase 5	compilation	33 (0.4)	35 (0.8)	27 (3.7)	29 (3.7)
Total		59	65	42	46

Table 8: Comparison of NASD/NFS and NASD/AFS performance for the Andrew benchmark against NFSv3 and AFS 3.4a on Digital UNIX. All results are averages of five runs with standard deviations in parentheses.

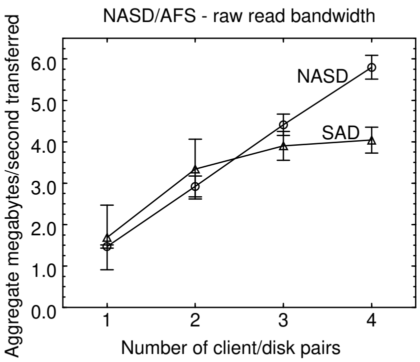


Figure 2a: The raw read benchmark shows the aggregate bandwidth each configuration is able to achieve. As the number of client/disk pairs increases, the SAD server is unable to linearly scale up the data transfer rate, because the file server itself becomes a bottleneck, while separate client/NASD pairs are able to transfer data blocks without file manager intervention

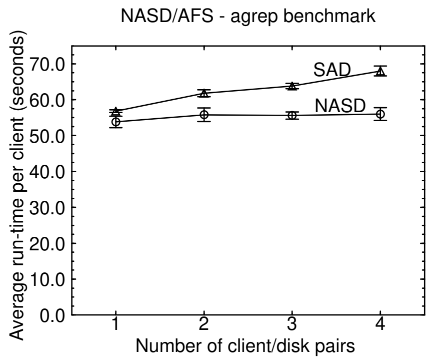


Figure 2b: The agrep benchmark requires clients to locate and process files in a multilevel tree. NASD clients must obtain capabilities to access each of these files before contacting the drive. As the number of client/disk pairs increases, the average time for the NASD configuration remains constant while it increases for SAD, suggesting that the fileserver represents a scalability bottleneck.

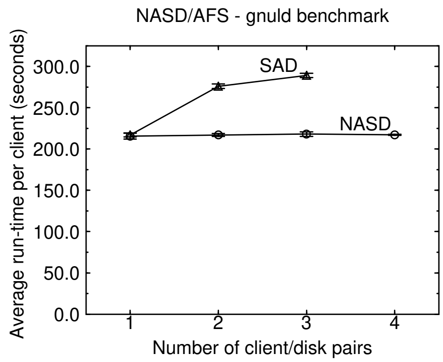


Figure 2c: In the gnuld benchmark, clients randomly access many medium-sized files. The clients interleave computation and I/O, and alternately read input files and write a single output file. A 4 client/4 drive point is not available for the SAD configuration because four clients saturate a server with a burst of requests sufficient to cause clients to time out and declare the server unavailable.

NASD/NFS file manager or the NASD drive as appropriate (e.g. `read` is directed to the drive, while `mkdir` is handled by the file manager which generates the necessary `create` and `write` operations to the drive). NFS files and directories map directly to NASD objects, and a combination of drive identifier and object identifier are used as the unique NFS file handle. File objects are read and written directly by the client, while directory objects are controlled by the file manager. NFSv3 attributes are either computed from NASD object attributes (e.g. modify times and object size) or kept in the *fs-specific* attribute (e.g. mode and uid/gid). The NASD/NFS file manager maintains an in-memory cache of directories and attributes similar to the way the buffer cache on a standard NFS server would be used.

The core portions of the new filesystem are the same as the code used by NFSv3 with minor changes to support mounting of the new filesystem type and redirection of the on-the-wire RPCs, a total of 325 lines of code. The redirection itself totals a further 4,750 lines of straightforward code to match NFSv3 and NASD/NFS data structures and send DCE RPC requests instead of the original ONC RPC requests. Drive identifiers, object identifiers, and capabilities are stored in the existing NFSv3 filehandle structure which provides 64 bytes of storage per client file and directory. The NASD/NFS file manager was written from the ground up and comprises a total of 6,300 lines of new code. Redundant communication with the drive is reduced by a file manager cache that stores recently used capabilities, attributes, and directory information.

Similar to AFS, basic NASD/NFS performance was measured using the Andrew benchmark. From Table 8, we see that NASD/NFS and NFSv3 have similar performance, with NASD/NFS again running 10% slower than NFSv3 for similar reasons. Specifically, DCE RPC's performance is not as optimized as ONC RPC; NFSv3 is capable of handling twice as many null RPCs per second as NASD/NFS. Also, the NASD/NFS file manager is implemented in a user-level process, requiring extra processing overhead for kernel crossings. Finally, some NASD/NFS operations simply spend more time on the wire, requiring messages to both the file manager and drive.

We also measured NASD/NFS scalability using the raw read and gnuld benchmarks as shown in Figure 3. The raw bandwidth results are similar to the NASD/AFS results, with NASD scaling to 12.5 MB/s at four clients with only 1.5% file manager CPU utilization. SAD, in contrast, reaches full utilization at 10.5 MB/s. Like the Andrew benchmark, gnuld run time on NASD/NFS is slightly longer than SAD, but file manager CPU utilization is significantly less.

The NASD architecture supports many performance optimizations that could significantly improve the performance of NASD/NFS. For example, clients could be allowed to parse the filesystem metadata, allowing operations such as `readdir` to occur directly between the client and drive. Implementing NASD-specific performance optimizations, however, would have made it a less fair comparison between NASD/NFS and SAD because NFSv3 clients should have the same opportunity.

4.4. NASD/NFS with Protected Communication

Our NASD prototype drive implements `NO_PROTECTION`, `INTEGRITY_ARGS`, and `INTEGRITY_DATA` protection options and uses timestamp-nonces. For the keyed hash function, we use HMAC-MD5 because of the wide acceptance of MD5 [Rivest92] and the theoretical strengths of the HMAC construction [Bellare96]. We use the reference implementation of HMAC-MD5 with MD5 code from the cryptolib library compiled by AT&T Bell Labs. We also utilize a capability cache on the drive to avoid unnecessary recalculations of capability keys. So far, the NASD/NFS client and file manager code also sup-

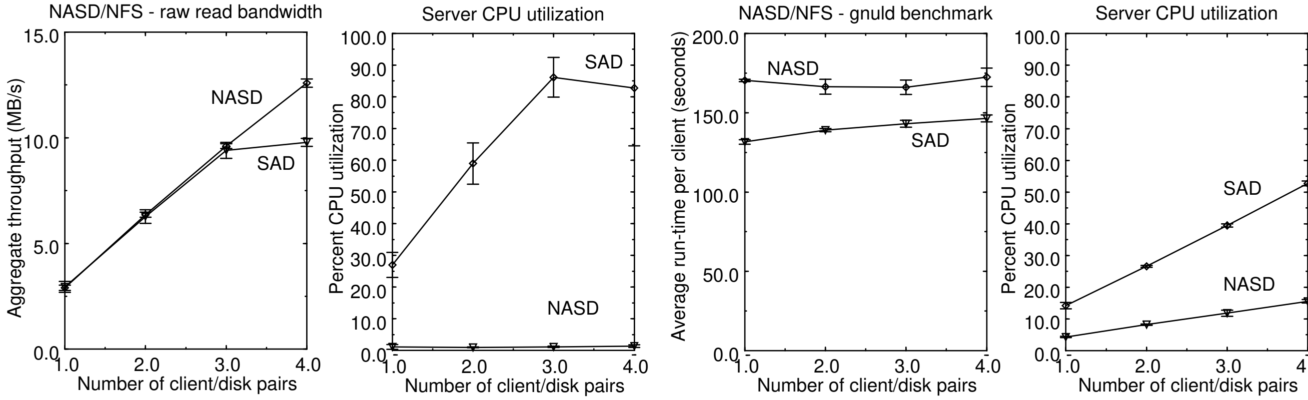


Figure 3. Scaling of the NASD/NFS file manager in comparison with an NFSv3 server. The chart shows bandwidth, run time and file manager CPU utilization as the number of client/drive pairs is increased. The server is the in-kernel NFSv3 implementation provided with Digital UNIX and the user-level NASD/NFS file manager described in the text.

port the `INTEGRITY_ARGS` and `INTEGRITY_DATA` protection levels. Experimental results show that using `INTEGRITY_ARGS` reduces peak bandwidth by 15% and by 55% when using both `INTEGRITY_ARGS` and `INTEGRITY_DATA`. While computing digests of all data is particularly expensive, this cost can be reduced with hardware support or selection of a cryptographic hash function optimized for a particular platform.

4.5. Striped NASD/NFS

We measured a striping NASD/NFS filesystem implemented as a user-level library (because our NFS kernel modifcations for high bandwidth are not ready yet). This allowed us to make large requests to NASD drives and demonstrate increased bandwidth without modifying the structure of the in-kernel client. Currently, the striping manager supports RAID level 0 and RAID level 1 in 2,935 lines of new code. The client clerk totals 6,700 lines of code and the file manager is the same as in NASD/NFS. The basic performance of most operations is identical to NASD/NFS; however, a `lookup` in striped NASD/NFS requires an extra message to the striping manager to obtain the stripe map and generate more capabilities.

To demonstrate the scalability of the striped NASD/NFS prototype, we set up two experiments. The first experiment is a synthetic benchmark consisting of *n* clients simultaneously reading a set of 5 different 8 MB files striped over all drives. Figure 4 contrasts the aggregate bandwidth for striped NASD/NFS and SAD; striped NASD/NFS scales linearly while SAD's throughput saturates quickly.

The second experiment consists of a large file text search (agrep) benchmark, consisting of *n* clients searching (agrep) through a 32 MB text file each. Figure 5 plots the number of MB/s searched for both NASD and SAD. For this benchmark, the `stdio` library used by agrep was replaced by an aggressive read-ahead library similar to Digital UNIX disk read-ahead. Agrep invokes the `stdio` library with 4 K requests. The read-ahead library initially reads a buffer size of 256 KB. The size of the read-ahead by the library doubles with each exhausted buffer for files exhibiting sequential access up to a maximum of 8 MB. In this experiment, file data is striped over the disks at a striping unit of 256 KB. As the graph shows, the aggregate search data rate in the SAD case flattens at 8.5 MB/s, while it keeps increasing for NASD at a rate of about 3.1 MB/s per client-drive pair.

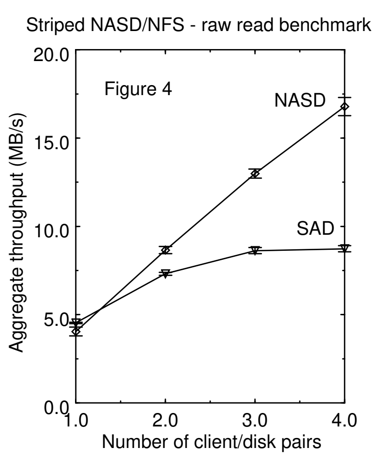
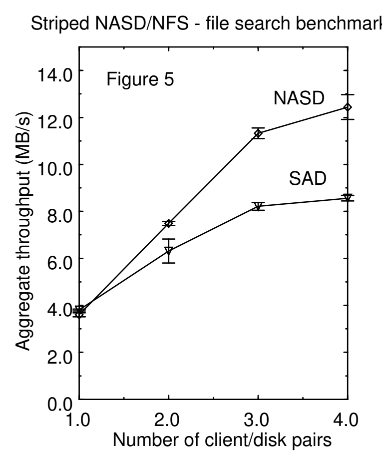


Figure 4: Per-client raw read bandwidth for striped NASD/NFS in comparison with NFSv3. Each NASD drive consists of two 1.0 GB HP C2247 drives striped at 64K stripe unit. For SAD, the same total number of disks were attached to the file server via 4 independent SCSI busses and striped with a 256K stripe unit.

Figure 5: Scaling for striped NASD/NFS in comparison with NFSv3. The NFSv3 server machine used is the same as the one used for NASD/NFS benchmark.



4.6. Informed Prefetching NASD/NFS

To demonstrate the value of NASD drive extensions, we have added an informed prefetching system [Cao95, Patterson95] to a NASD drive and the NASD/NFS client. Applications issue hints to the drives to describe their future read accesses. Drives use this knowledge to prefetch data so that they will be able to satisfy client requests from their caches.

In our informed prefetching NASD/NFS prototype, the application issues hints to client code that resolves the names to NASD drive and object identifiers with the help of the NASD/NFS name resolution code. This code batches the hints, and delivers them to the drives along with a read capability provided by NASD/NFS. Since the client also has an LRU cache of file buffers, and our hints do not encapsulate a notion of expected consumption time, the drive cannot differentiate between an application that is rapidly progressing through its hint stream, but hitting in its LRU cache, and an application that is compute-bound and slowly issuing I/Os from its hint stream. To resolve this confusion, the client periodically notifies the drive of the reads it has issued. The drives use a slightly-modified version of the [Patterson95] system to translate the hints into prefetching requests and caching decisions.

To evaluate the performance benefit of our NASD prefetching extension, we used a hinting version of the XDataSlice 3D scientific visualization application [NCSA89] to render 25 random planar slices through a 256 x 256 x 256 cube of 32-bit data values. We used a DEC 3000/400 (133 MHz, 64 MB, Digital UNIX 3.2g-3) client with a DEC 3000/600 (175 MHz, 64 MB, Digital UNIX 3.2g-3) for the drive, and an AlphaStation 600 5/266 (266 MHz, 64 MB, Digital UNIX 3.2g-3) as the file manager. The drive machine striped its data over three local 1.0 GB HP C2247 disks. Without prefetching, slice rendering took an average of 120.2 seconds. With prefetching and read notifications, slice rendering took an average of 68.1 seconds, a speed up of 1.76. Benefits like this will motivate specific applications to purchase appropriate disks provided the specialized disk's advantages can be achieved without changing the community's distributed filesystem.

5. Related Work

The idea of a simple, disk-like network-attached storage server whose functions are employed by high-level distributed filesystems has been around for a long time. Cambridge's Universal File Server used a simple memory abstraction, like SCSI today, but advocated that the server do all allocation and reclamation [Birrell80]. Accordingly, a directory-like index structure had to be understood by the server so it could detect no reference to a range of storage and free it. Cambridge UFS used capabilities primarily to distinguish one high-level filesystem from another. The follow-on filesystem at Cambridge, CFS, used unrevokable capabilities to authorize clients' access to files, continued UFS's automatic reclamation through storage-understood indices, and added undoable (for a period of time after initiation) transactions into the filesystem interface [Mitchell82]. With the well-understood journalling technology available in higher-level file systems today [Chutani92, Birrell93], we chose to avoid the cost of automatic reclamation of objects and arbitrarily larger storage-level transactions in NASD.

Recently, there has been renewed interest in the logical interface to storage, driven primarily by the huge cost of managing rapidly growing storage repositories. We have borrowed representation of storage clustering hints as a list of lists from deJonge's logical disk model to escape the SCSI block address space and the heroics that filesystems do to achieve contiguity in these linear address spaces [deJonge93, McKusick84, McVoy91, Rosenblum91]. Others have simply resorted to transparent manipulation of clustering by either writing to the closest free block [English92] or reorganizing according to observed access activity [StorageTek94, Wilkes95]. Focusing on the artificial "partition exhausted" problem of filesystems whose logical space management is implemented as immutable commitment of disk space, the concept of virtual volumes or virtual disks has been advocated [Lee96, IEEE94]. NASD achieves much of this by regarding partitions as a grouping of objects whose total size is constrained but mutable rather than a region of storage; however, striped NASD/NFS also constructs virtual objects and partitions. Another significant trend in storage management is the exploitation of access pattern and workload attributes [Golding95, IEEE94]. While we have not extensively explored this area, we believe that NASD implementations can offer extended interfaces to interpret filesystem-specific attribute fields in this way.

Direct transfer from storage to client is a prominent feature in many storage designs. Most rely on synchronous file manager oversight and offer network striping [IEEE94, Drapeau94, Cabrera91]. However, Berkeley's Zebra network-striped LFS post-processes authorization for client actions [Hartman93] and ISI's Derived Virtual Devices [VanMeter96a] binds file definition and access authorization into the initialization of communication state and exploits authenticated RPC to verify a client's binding to the authorized communication state.

Several projects have also investigated the security issues raised in network-attached storage environments. Derived Virtual Devices [VanMeter96a] present a mechanism, based on Kerberos [Neuman94], to secure shared access to network-attached peripherals. Capabilities are a well established concept [Dennis66] for regulating access to resources. In the past, many systems have used capability systems that rely on hardware support or trusted operating system kernels to protect system integrity [Karger88, Wilkes79, Wulf74]. Within NASD, we do not make assumptions about the integrity of the client which maintains capabilities. Therefore, we utilize cryptographic techniques similar to ICAP [Gong89] and Amoeba [Tanenbaum86]. In these systems, the act of issuing a capability and validating a capability must have access to a large amount of private information about all the issued capabilities. Both ICAP and Amoeba are described in terms of a single entity performing issuing and validating functions while in NASD these functions are done in distinct locations and no per-capability state is exchanged between issuer and validator.

6. Conclusions

With dramatic performance improvements and cost reductions in microprocessors anticipated for the foreseeable future, high-performance personal computers will continue to proliferate, computational data sets will continue to grow, and distributed filesystem performance will increasingly impact human productivity and overall system acquisition and operating cost.

Network-attached secure disks (NASD) is a storage interface and architecture for meeting the needs of next-generation distributed filesystems and clients. NASD drives enable scalable bandwidth by directly transferring data between storage and client; they improve load scaling by offloading common request processing and authentication to storage devices; they ensure high levels of security by employing cryptographically secured capabilities independent of the local security environment; they reduce operating costs by increasing awareness of relationships among storage units, which enables more effective self-management; and they enable storage designers to provide application-specific interfaces for optimizations such as informed prefetching transparent to the managing filesystem.

In this paper we have presented a detailed description of a prototype NASD drive interface based on storage objects and shown how its capabilities enable both asynchronous enforcement of access control at the drive and secure communications. We have also described four filesystem adaptations to this NASD interface. While Sun's NFSv3 is simply layered on NASD objects, Transarc's AFS requires more exploitation of time-limited capabilities and revocation to support cache-consistency and quota semantics. Our NFS extension for striping files over NASD drives appears as a pseudo-NASD that serves striped files to the NASD/NFS file manager and manipulates multiple NASD drives internally. Finally, our NASD extension for informed prefetching and caching of objects customizes NASD performance to specific applications that can exploit the extension while requiring only a simple "pass-through" interface to direct extension information to the correct NASD drive and pass the necessary capabilities.

Our experiments with these prototype implementations, although limited to only four pairs of client and drive machines, demonstrate scalable bandwidth. Notably, aggregate transfer bandwidth is 20%-90% higher for NASD than for traditional systems when both have four clients and drives. Scaling the number of client-drive pairs with more typical workloads shows less benefit than expected (the Andrew benchmark runs 10% slower in NASD) primarily because file manager operations, which should have similar costs in SAD and NASD architectures, in fact cost significantly more in NASD. This is because moving small amounts of data through the DCE RPC, UDP/IP, and ATM layers of our network is much more expensive than moving the same data over a SCSI bus. Finally, informed prefetching and caching, an optimization that may serve too small an application market to persuade high-volume filesystem designers to change the core of their filesystem cache implementations, can be implemented as a NASD extension and provide specific I/O-intensive applications with runtime reductions of 45%.

NASD architectures and distributed parallel filesystems built on NASD present many open problems. Storage self-management and powerful extension optimizations are probably the most significant, although determining how to design and construct cost-effective fast storage messaging for local area networks is also critical.

7. Acknowledgments

This research is sponsored by DARPA/ITO through ARPA Order D306, and issued by Indian Head Division, NSWC under contract N00174-96-0002. Additional support was provided by NSF and ONR graduate fellowships. The project team is indebted to generous contributions from the member companies of the Parallel Data Consortium, including: Hewlett-Packard Laboratories, Symbios Logic Inc., Data General, Compaq, IBM Corporation, Seagate Technology, and Storage Technology Corporation.

We would like to thank Eugene Feinberg, Paul Mazaitis, Berend Ozceri, Hugo Patterson, and Doug Tygar for their help with the arguments presented here and their invaluable assistance in running our experiments. We would also like to thank all the members of the Parallel Data Lab who provided support and endured the inconvenience of our experiments. Finally, we would like to thank the Computing, Media, and Communications Laboratory at Carnegie Mellon for graciously allowing us to borrow their ATM switch at the eleventh hour.

8. Bibliography

[Anderson95] Anderson, D., Seagate Technology Inc., Personal communication, 1995.

[Baker91] Baker, M.G. et al., "Measurements of a Distributed File System", 13th SOSP, Oct. 1991, pp. 198-212.

[Bellare96] Bellare, M., Canetti, R., and Krawczyk, H., "Keying Hash Functions for Message Authentication", Advances in Cryptology: Crypto '96 Proceedings, 1996.

[Benner96] Benner, A.F., “Fibre Channel: Gigabit Communications and I/O for Computer Networks”, McGraw Hill, New York, 1996.
[Berdahl95] Berdahl, L., Draft of “Parallel Transport Protocol Proposal”, Lawrence Livermore National Labs, Jan. 1995.
[Birrell80] Birell, A.D. and Needham, R.M., “A Universal File Server”, IEEE Transactions on Software Engineering 6,5, Sept. 1980.
[Birrell93] Birrell, A. et al., “The Echo Distributed File System”, Research Report 111, DEC SRC, Palo Alto, CA, Sept. 1993.
[Burrows92] Burrows, M. et al., “On-line Data Compression in a Log-structured File System”, 5th ASPLOS, Oct. 1992.
[Cabrera91] Cabrera, L. and Long, D., “Swift: Using Distributed Disk Striping to Provide High I/O Data Rates”, Computing Systems 4:4, Fall 1991.
[Cao95] Cao, P. et al., “A Study of Integrated Prefetching and Caching Strategies,” SIGMETRICS 95, May 1995.
[Carey94] Carey, M.J. et al., “Shoring Up Persistent Applications”, SIGMOD 94, 1994, pp. 383-394.
[Chutani92] Chutani, S. et al., “The Episode File System”, Winter 1992 USENIX, 1992, pp. 43-60.
[Deering95] Deering, S. and Hinden, R., “Internet Protocol Version 6 Specification”, RFC 1883, Dec. 1995.
[deJonge93] de Jonge, W., Kaashoek, M.F. and Hsieh, W.C., “The Logical Disk: A New Approach to Improving File Systems,” 14th SOSP, Dec. 1993.
[Dennis66] Dennis, J.B. and Van Horn, E.C., “Programming Semantics for Multiprogrammed Computations”, CACM 9,3, 1966, pp. 143-155.
[Drapeau94] Drapeau, A.L. et al., “RAID-II: A High-Bandwidth Network File Server”, 21st ISCA, 1994, pp.234-244.
[English92] English, R.M. and Stepanov, A.A., “Loge: a Self-Organizing Disk Controller”, Winter 1992 USENIX, Jan. 1992, pp. 237-251.
[Gibson97] Gibson, G. et al., “File Server Scaling with Network-Attached Secure Disks”, 1997 SIGMETRICS, June 1997.
[Gobioff97] Gobioff, H. Gibson, G., Tygar, J.D., “Security for Network Attached Storage Devices”, Work in Progress.
[Golding95] Golding, R., Shriver, E., Sullivan, T., and Wilkes, J., “Attribute-managed storage,” Workshop on Modeling and Specification of I/O, San Antonio, TX, October 1995.
[Gong89] Gong, L., “A Secure Identity-Based Capability System” IEEE Symposium on Security and Privacy, May 1989, pp. 56-64.
[Hartman93] Hartman, J.H. and Ousterhout, J.K., “The Zebra Striped Network File System”, 14th SOSP, Dec. 1993, pp. 29-43.
[Holland94] Holland, M. et al., “Architectures and Algorithms for On-Line Failure Recovery in Redundant Disk Arrays”, Parallel and Distributed Databases 2,3, July 1994.
[Howard88] Howard, J.H. et al., “Scale and Performance in a Distributed File System”, ACM TOCS 6, 1, Feb. 1988, pp. 51-81.
[IEEE94] IEEE P1244. “Reference Model for Open Storage Systems Interconnection-Mass Storage System Reference Model Version 5”, Sept. 1995.
[Karger88] Karger, P.A., “Improving Security and Performance for Capability Systems”, University of Cambridge Computer Laboratory Technical Report No. 149, Oct. 1988.
[Katz92] Katz, R.H., “High-Performance Network- and Channel-Attached Storage”, Proceedings of the IEEE 80, 8, Aug. 1992.
[Kim86] Kim, M.Y., “Synchronized disk interleaving”, IEEE Transactions on Computers C-35, 11, Nov. 1986.
[Lampen91] Lampen, A., “Advancing Files to Attributed Software Objects”, 1991 Winter USENIX, 1991, pp. 219-229.
[Lee95] Lee, E.K., “Highly-Available, Scalable Network Storage”, 1995 Spring COMPCON, Mar. 1995.
[Lee96] Lee, E.K. and Thekkath, C.A., “Petal: Distributed Virtual Disks”, 7th ASPLOS, Oct. 1996.
[Livny87] Livny, M., “Multi-disk management algorithms”, 1987 SIGMETRICS, May 1987.
[Long94] Long, D.D.E., Montague, B.R., and Cabrera, L., “Swift/RAID: A Distributed RAID System,” Computing Systems 7,3, Summer 1994.
[McKusick84] McKusick, M.K. et al., “A Fast File System for UNIX”, ACM TOCS 2, Aug. 1984, pp. 181-197.
[McVoy91] McVoy, L.W. and Kleiman, S.R., “Extent-Like Performance from a UNIX File System”, Winter 1991 USENIX, 1991.
[Miller88] Miller, S.W., “A Reference Model for Mass Storage Systems”, Advances in Computers 27, 1988, pp. 157-210.
[Mills88] Mills, D., “Network Time Protocol Version 1 Specification and Implementation”, RFC 1059, July 1988.
[Mitchell82] Mitchell, J. and Dion, J., “A Comparison of Two Network-Based File Servers”, 8th SOSP, Dec. 1981.
[NCSA89] National Center for Supercomputing Applications "XDataSlice for the X Window System" UIUC, 1989.
[Neuman93] Neuman, B.C. and Stubblebine, S.G., “A Note on the Use of Timestamps and Nonces”, OS Review 27,2, Apr. 1993.
[Neuman94] Neuman, B.C. and Ts’o, T., “Kerberos: An Authentication Service for Computer Networks”, IEEE Communications 32,9, Sept. 1994, pp. 33-38.
[Patterson88] Patterson, D.A., Gibson, G. and Katz, R.H., “A Case for Redundant Arrays of Inexpensive Disks (RAID)”, 1988 SIGMOD, June 1988, pp. 109-116.
[Patterson95] Patterson, R.H. et al., “Informed Prefetching and Caching”, 15th SOSP, 1995.
[vanRenesse89] van Renesse, R. et al. “The Design of a High-Performance File Server”, 9th International Conference on Distributed Computer Systems, 1989, pp. 22-27.
[Rivest92] Rivest, R., “The MD5 Message-Digest Algorithm”, RFC 1321, Apr. 1992.
[Rosenblum91] Rosenblum, M. and Ousterhout, J.K., “The Design and Implementation of a Log-Structured File System”, 13th SOSP, 1991.
[Sandberg85] Sandberg, R. et al., “Design and Implementation of the Sun Network Filesystem”, Summer 1985 USENIX, June 1985, pp. 119-130.
[Satya89] Satyanarayanan, M., “Integrating Security in a Large Distributed Environment”, ACM TOCS 7,3, Aug. 1989, pp. 247-280.
[Seagate96] Seagate Technology Inc., “Barracuda Family Product Brief (ST19171)”, 1996.
[Seagate96a] Seagate Technology Inc., “Elite Family Product Brief (ST423451FC)”, 1996.
[Shekita90] Shekita, E. and Zwilling, M., “Cricket: A Mapped Persistent Object Store”, Persistent Object Ssytems Workshop,

Martha's Vineyard, MA, Sept. 1990.
[Shirriff92] Shirriff, K. and Ousterhout, J., "Sawmill: A High Bandwidth Logging File System", Summer 1994 USENIX, June 1994.
[StorageTek94] Storage Technology Corporation, "Iceberg 9200 Storage System: Introduction", STK Part Number 307406101, 1994.
[Tanenbaum86] Tanenbaum, A.S. et al., "Using Sparse Capabilities in a Distributed System", 6th International Conference on Distributed Computing, 1986, pp. 558-563.
[Tygar95] Tygar, J.D. and Yee, B.S., "Secure Coprocessors in Electronic Commerce Applications," Proceedings 1995 USENIX Electronic Commerce Workshop, 1995, New York.
[VanMeter96] Van Meter, R., "A Brief Survey Of Current Work on Network Attached Peripherals (Extended Abstract)", OS Review 30,1, Jan. 1996.
[VanMeter96a] Van Meter, R., Holtz, S., and Finn G., "Derived Virtual Devices: A Secure Distributed File System Mechanism", Fifth NASA Goddard Conference on Mass Storage Systems and Technologies, College Park, MD. Sept. 1996.
[Wilkes79] Wilkes, M.V. and Needham, R.M., The Cambridge CAP Computer and Its Operating System, 1979.
[Wilkes95] Wilkes, J. et al., "The HP AutoRAID Hierarchical Storage System", 15th SOSP, Dec. 1995.
[Wulf74] Wulf, W.A. et al., "HYDRA: The Kernel of a Multiprocessor Operating System", CACM 17,6, June 1974, pp. 337-345.