The File System of an Integrated Local Network

Paul J. Leach, Paul H. Levine,
James A. Hamilton, and Bernard L. Stumpf

Apollo Computer, Inc.
15 Elizabeth Drive, Chelmsford, MA 01824

Abstract

The distributed file system component of the DOMAIN system is described. The DOMAIN system is an architecture for networks of personal workstations and servers which creates an integrated distributed computing environment. The distinctive features of the file system include: objects addressed by unique identifiers (UIDs); transparent access to objects, regardless of their location in the network; the abstraction of a single level store for accessing all objects; and the layering of a network wide hierarchical name space on top of the UID based flat name space. The design of the facilities is described, with emphasis on techniques used to achieve performance for access to objects over the network.

1. Introduction

This paper describes the design of the distributed file system for the Apollo DOMAIN operating system. DOMAIN is an integrated local network of powerful personal workstations and server computers ([APOL 81], [NELS 81]); both of which are called *nodes*. A DOMAIN system is intended to provide a substrate on which to build and execute complex professional, engineering and scientific applications ([NELS 83]). Other systems built following the integrated model of distributed computing include EDEN [LAZO 81] and LOCUS [POPE 81].

Within the DOMAIN system, the network and the distributed file system contribute to this goal by allowing the professional to share programs, data, and expensive peripherals, and to cooperate via electronic mail, with colleagues in much the same manner as on larger shared machines, but without the attendant disadvantage of sharing processing power. Cooperation and sharing are facilitated by being able to name and access all objects in the same way regardless of their location in the network.

Thus, when we say that DOMAIN is an integrated local network, we mean that all users and applications programs have the same view of the system, so that they see it as a single integrated whole, not a collection of individual nodes. However, we do not sacrifice the autonomy of personal workstations to achieve integration: each personal workstation is able to stand alone, but the system provides mechanisms which the user can select that permit a high degree of cooperation and sharing when so desired.

Another reason we say that DOMAIN is an integrated local network is that each machine runs a complete (but highly configurable) set of standard software, which (potentially) provides it with all the facilities it normally needs – file storage, name resolution, and so forth. In contrast are server-based distributed systems, wherein network wide services are provided by designated machines ("servers") which run special purpose software tailored to providing some single service or small number of services (*e.g.* Grapevine [BIRR 82], WFS [SWIN 79], and DFS [STUR 80]). DOMAIN has server nodes; however, they are created by configuring the standard hardware and software for a special purpose – a "file server" node, say, is created using a machine with several large disks and system software configured with the appropriate device drivers.

1.1. Organization

The rest of this paper is organized as follows. The remainder of this introduction briefly descibes the hardware environment on which the system runs. Section 2 provides an overview of the file system, and breaks

it into four major component groups. Section 3 gives a block diagram of the file system structure, and a brief description of each module, locating it within one of the component groups. Sections 4, 5, 6, and 7 each describe one of these component groups. Finally, section 8 focuses on those aspects of the design which we believe have contributed most to the efficiency of the system.

1.2. Hardware Environment

A DOMAIN system consists of a collection of powerful personal workstations and server computers (generically, nodes) interconnected by a high speed local network.

1.2.1. User Interface

Users interact with their personal nodes via a display subsubsystem, which includes a high resolution raster graphics display, a keyboard and a locating device (mouse, touch pad, or tablet). A typical display has 800 by 1024 pixels, and *bit BLT* (bit block transfer) hardware to move arbitrary rectangular areas at high speed. Server nodes have no display, and are controlled over the network. More information on the user environment can be found in [NELS 84].

1.2.2. CPU

There are several models of both personal and sever nodes. Their 'tick' times [LAMP 80] range from .4 to 1.25 microseconds; their maximum main memory ranges from 3.5 megabytes to 8 megabytes. Most personal nodes have 33 to 154 megabytes of disk storage and a 1 megabyte floppy disk, but no disk storage is required for a node to operate. Server nodes configured as file servers can have 300-1000 megabytes or more of disk storage; those configured as peripheral servers can have printers, magnetic tape drives, plotters, and so forth.

All nodes have dynamic address translation (DAT) hardware which supports up to 128 processes, with each process able to to address 16 or 256 megabytes of demand paged virtual memory (depending on CPU model). The DAT hardware on some models uses a reverse mapping scheme, similar to that used in the IBM System/38 [HOUD 78]; it is a large, hardware hash table keyed by virtual address, with the physical address given by the hash table slot number in which a translation entry is stored. Other models use a forward mapping scheme, similar to the VAX [DEC 79] or System/370 [IBM 76]. The DAT also maintains used and modified statistics on a per page basis for the use of page replacement software, and access protection controlling read, write and execute access. The differences between the DATs of the different models are abstracted away by an **MMU** (memory management unit) module.

1.2.3. Network

The network is a 12 megabit per second baseband token passing ring (other ring implementations are described in [WILK 79], [GORD 79]; and reasons for preferring a ring network in [SALT 79], [SALT 81]). Each node's ring controller provides the node with a unique node ID, which is assigned at the factory and contained in the controller's microcode PROMs. The maximum packet size is 2048 bytes. The controller has a broadcast capability.

We will not discuss the network further here; for purposes of the file system, all that is required is that the it deliver messages with high probability and low CPU overhead. For more information on the ring controller and data link layer protocols see [LEAC 83].

2. File System Overview

The DOMAIN file system is actually made of four distinct components: an *object storage system* (OSS), the *single level store* (SLS), the *lock manager*, and the *naming server*. (See figure 1 for a block diagram.)

The OSS provides a flat space of objects (storage containers) addressed by unique identifiers (UIDs). Objects are typed, protected, abstract information containers: associated with each object is the UID of a type descriptor, the UID of an access control list (ACL) object, a disk storage descriptor, and some other attributes: length; date time created, used and modified; reference count; and so forth. Object types include: alphanumeric text, record structured data, IPC mailboxes, DBMS objects, executable modules, directories, access control lists, serial I/O ports, magnetic tape drives, and display bit maps. (Other objects which are *not* information containers also exist. UIDs are used to identify processes; and to identify persons, projects, organizations, and protected subsystems for authentication and protection purposes.) The distributed OSS makes the objects on each node accessible throughout the network (if the objects' owners so choose by setting the objects' ACLs appropriately). The operations provided by the OSS on storage objects include: creating, deleting, extending, and truncating an object; reading

or writing a page of an object; getting and setting attributes of an object such as the ACL UID, type UID, and length; and locating the home node of an object. The OSS automatically uses a node's main memory as a cache of recently used pages, attributes, and locations of objects, including remote ones. It does nothing to guarantee cache consistency between nodes; however, it does provide mechanisms that the lock manager can use to make and enforce such guarantees.

A unique aspect of the DOMAIN system is its network wide single level store (SLS). (Multics [ORGA 72] and the IBM System/38 [FREN 78] are examples of a single level store for centralized systems.) Programs access all objects by presenting their UIDs and asking for them to be "mapped" into the program's address space (see [REDE 80] on the desirability of mapping in distributed systems); subsequently, they are accessed with ordinary machine instructions, utilizing virtual memory demand paging.

The purpose of the single level store is not to create network wide shared memory semantics akin to those of a closely coupled multiprocessor; instead, it is a form of lazy evaluation: only required portions of objects are actually retrieved from disk or over the network. Another purpose is to provide a uniform, *network transparent* way to access objects: the mapping operation is independent of whether the UID is for a remote or local object. As long as programs make the worst case assumption that their objects are not local, and hence that operations on them are subject to communication failures, they need not be aware of their location. (See [POPE 81] on the desirability of network transparency.)

The lock manager serializes multiple simultaneous access to objects by many processes, including ones on different nodes. A process must lock an object prior to its use; the lock manager arbitrates lock requests, and uses the sequence of requests to keep main memory caches consistent.

The naming server allows objects to be referred to by text string names. It manages a collection of directory objects which implements a hierarchical name space much like that of Multics or UNIX[1] [RITC 74]. The result is a uniform, network wide name space, in which objects have a unique canonical text string name as well as a UID. The name space supports convenient sharing, which would be severely hampered without the ability to uniformly name the objects to be shared among the sharing parties.

[1]UNIX is a trademark of Bell Laboratories.

3. File System Structure

Figure 1 shows a block diagram of the file system. Each of the major component groups is indicated by a different form of shading. The arrows between blocks indicate call dependencies; in addition, all modules above the "pageable" boundary have an implicit dependency on the SLS.

The system is stuctured using a data abstraction approach, sometimes called a "type manager" approach when applied to operating systems ([JANS 76]). Each module has a set of operations and a private database in which to record its state. Thus, in describing the components of the system, we will identify the managers which comprise that component, and then, for for each manager, the essential operations provided by that manager, and an indication of the form of the database and algorithms used to implement the operations. (Note: in the descriptions of calls in this paper, irrelevant details have often been suppressed for ease of exposition; the intent is to capture the semantic flavor of the interfaces, not their precise syntax.)

4. Object Storage System

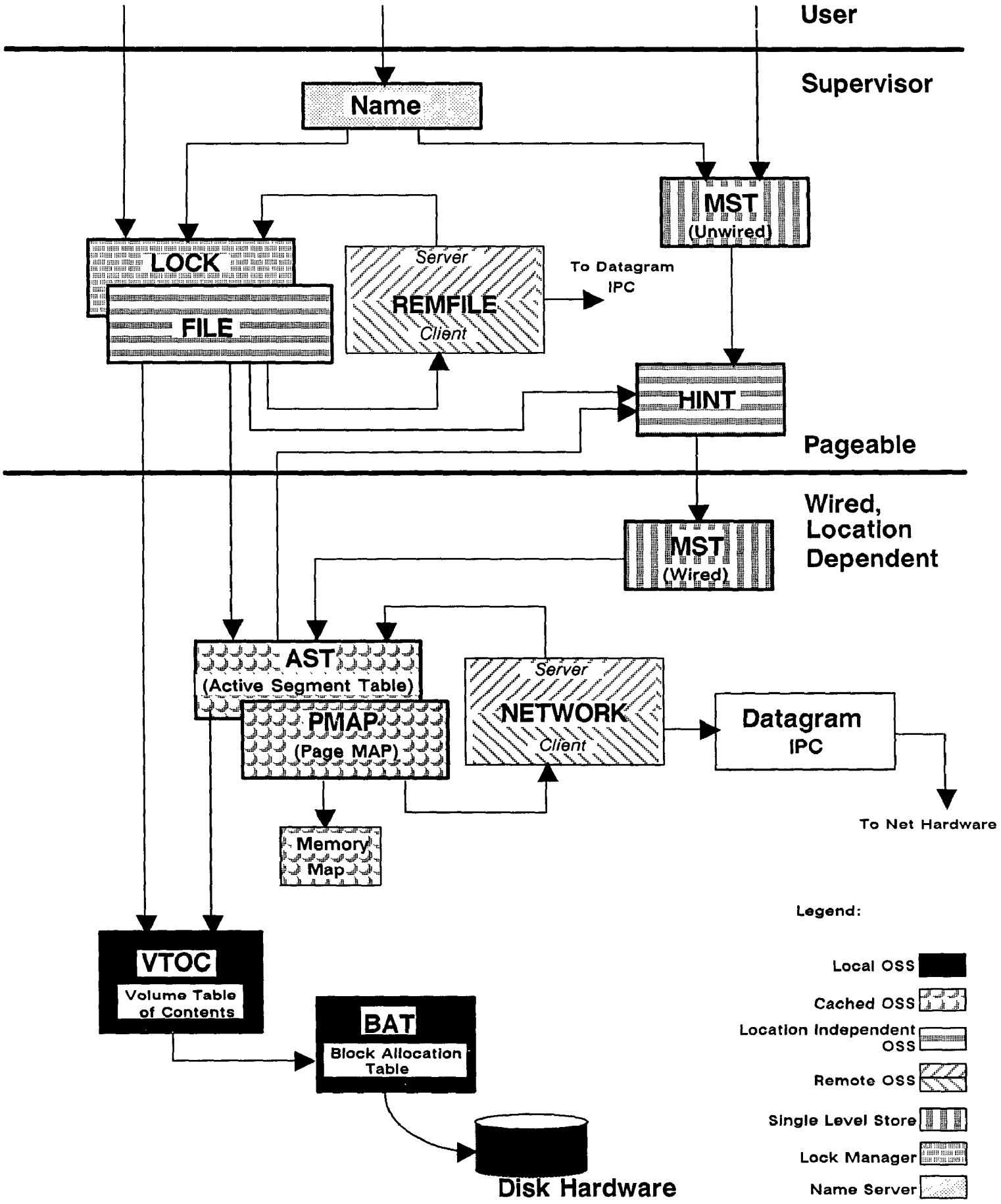
The OSS is the DOMAIN counterpart of distributed file systems such as WFS [SWIN 79] and DFS [STUR 80]. The purpose of the OSS is to provide permanent storage for objects, and to allow objects to be identified by and operated on using UIDs, independent of their location in the network.

At the level we will discuss here, an object is just a data container: an array of uninterpreted data bytes, or more precisely, an array of pages (1024 byte units into which objects are divided). Other object attributes, such as it's type descriptor and access control list are not used by the OSS, but are simply stored for the use of higher levels. (Not all objects are represented by storage containers: for example, processes are identified by UIDS, but are not associated with any permanent storage.)

The OSS consists of several component subgroups: a *local OSS*, *remote OSS*, *cached OSS*, and an *object locating service*. The top-level *location independent OSS* abstraction is created utilizing these services.

4.1. Identifying Objects

UIDs of objects are bit strings (64 bits long); they are made unique by concatenating the unique ID of the



File System Structure

node generating the UID and a time stamp from the node's timer. (The system does not use a global clock.) UIDs are also *location independent*: the node ID in an object's UID can not be considered as anything more than a hint about the current location of the object. (More detail on the use and implementation of UIDs is presented in [LEAC 82].)

At any point in time, the permanent storage for an object resides entirely at only one node; also, the system never attempts to transparently move it to a different node. So, for every object there is always one distinguished node which is its "home", and which serves as the locus of operations on the object. Above the OSS level, only UIDs are used to address objects; an operation whose UID addresses a remote object is sent to the object's home node to be performed.

4.2. Local OSS

This subgroup provides access to local objects: *i.e.*, those objects stored on disk volumes which are attached to the node accessing them. It provides operations to create and delete local objects, and to access the attributes and contents (pages) of existing objects (see figure 2). There are two managers in this group: the VTOC (volume table of contents) and the BAT (block allocation table).

The VTOC for a volume contains an entry for each object on the volume; an object's VTOC entry contains the object's attributes and the root of its *file map*, which translates page numbers within an object to disk block addresses. (VTOC entries are very similar to UNIX *inodes* [THOM 78].) The VTOC is organized as an associative lookup table keyed by object UID, which permits rapid location of an object's VTOC entry given its UID. (Using a large direct mapped hash table with chained overflow buckets and avoiding high utilization, the average lookup time is just over one disk access.)

To access the contents of an object requires two steps: translate the object reference to disk block address, then read (or write) the disk block. (An object reference is a pair consisting of the object's UID and a page number within the object.) The VTOC only provides operations to do the translation, not the reads or writes, because the translations are then cached and used by the cached OSS (see below). The translation is done by reading or writing the file map for 32 page units of the file called *segments*.

The BAT for a volume keeps track of which disk blocks are available for allocation on that volume. The principle operations on the BAT are ones to allocate

---

allocate — allocate a VTOC entry for an empty object and set its attributes

The object is created on the local disk volume specified by 'vol-index'. The object descriptor contains the object's UID and initial attributes.

FUNCTION allocate(vol-index, obj-decriptor): vtoc-index

---

lookup — get the VTOC index of an object

FUNCTION lookup(vol-index, obj-uid): vtoc-index

---

read — get the VTOC entry of an object given its VTOC index

Attributes in the 'vtoc-entry' include: object UID; type UID; ACL UID; length; time created, used, and modified; reference count, etc.

FUNCTION read(vol-index, vtoc-index): vtoc-entry

---

write — write the VTOC entry of an object given its VTOC index

Note: overwriting a VTOC entry for an object with an empty VTOC entry has the effect of deleting the object.

FUNCTION write(vol-index, vtoc-index, vtoc-entry)

---

read-fm — get the file map for a segment of an object

Object are divided into 32 page segments; the 'seg-no' indentifies the segment; the 'file-map' is an array of 32 disk block addresses, one for each page in the segment.

FUNCTION read-fm(vol-index, vtoc-index, seg-no): file-map

---

write-fm — write the file map for a segment of an object

FUNCTION write-fm(vol-index, vtoc-index, seg-no, file-map)

Figure 2: Sample VTOC Operations

and free disk blocks. One interesting feature is that the allocation operation aids in creating locality of the pages within an object on the disk. One of the input parameters of the allocation operation is a disk block address; an attempt is made to make the newly allocated block as close as possible to it. When a new page is being added to an object, this parameter is usually set to the disk address of the previous logical page of that object. We observe that this causes much better clustering of objects on the disk than not doing anything at all, except when the disk is nearly full. (We have not analyzed the benefit quantitatively. Also, to get really good locality, it is probably necessary to use the more comprehensive methods of [MCKU 84].)

4.3. Cached OSS

Disk operations and remote operations are both expensive, so it is desirable to avoid them when possible. One means of doing so is to cache recently obtained results of such operations, and reuse them when it can be ascertained that they are still valid.

The cached OSS consists of the AST, PMAP, and MMAP managers. The AST (active segment table) caches locations, pages, and attributes of *active* (recently used) objects, whether local or remote. Each entry in the AST contains the UID, location and attributes of an object, plus the PMAP for one segment of the object. The PMAP (page map) for a segment contains the file map for that segment, plus references to all resident main memory pages. Part of the maintenance of PMAPs is done by the *purifier* process, which periodically writes back modified pages to secondary storage (local or remote, as need be). The MMAP (memory map) is the allocator of main memory pages, and keeps track of their contents.

The AST provides operations to access pages and attributes (including locations) of objects (see figure 3). If the requested information is not in its cache (or PMAP's), then it uses the local or remote OSS to get the necessary information and encache it. The *touch* operation fetches object contents (pages). (There is no write operation; pages are modified via the single level store while in the cache, then written back later by the PMAP purifier process.) The *get-attr* operation fetches object attributes, and *set-attr* allows objects' attributes to be individually changed.

The AST also provides operations to manage its cache's consistency with that of other nodes, and which are designed to be used by the lock manager: it only allows access to objects if they are properly locked; it

---

touch — cause several consecutive pages of an object to be cached in main memory

Cause 'n' pages pages starting with 'page-num' of object with UID 'object-uid' to be cached. The object 'location' is the ID of the remote node or local volume where the object resides.

FUNCTION touch(location, object-uid, page-num, n): phys-page-list

---

get-attr — get an object's attributes

Attributes in the 'attr-rec' include: type UID; ACL UID; length; time created, used, and modified; reference count, etc.

FUNCTION get-attr(object-uid): attr-rec

---

set-attr-X — set attribute X of an object

This is a set of operations, where X can be replaced by any of the attributes above.

PROCEDURE set-attr-X(object-uid, X-value)

---

cond-flush — remove stale pages of an object from the cache

The boolean 'flushed' is true if any stale data was flushed.

FUNCTION cond-flush(object-uid, dtm): flushed

---

purify — send all modified pages of an object back to its 'home' node

if 'force' is true, write the pages to disk immediately at the home node, else just leave them in the home node's cache.

PROCEDURE purify(object-uid, force)

Figure 3: Sample AST Operations

maintains a version number for each object; and it provide operations to control the contents of the cache.

4.3.1. Lock Enforcement

As one of its attributes, each file system object has a *lock key*. The lock key is set to either a network node ID or one of (for now) two special values: **readbyall** and **writebyall**. When an object's lock key is set to N, only OSS requests from node N are processed. All other requests are denied with an error indication of concurrency violation. When the lock key is set to **readbyall**, read requests (for pages and attributes) from every node are allowed while all write requests are denied regardless of their source. Finally, a lock key value of **writebyall** completely disables the OSS level concurrency control checking and so all requests are always fulfilled.

4.3.2. Object Versions

A time stamp based version number scheme is used to support the cache validation mechanism. An object's version number is its *date-time modified* (DTM) attribute. (See [KOHL 81] for a survey of distributed concurrency techniques.) Every object has a DTM with 8 millisecond resolution associated with it, which records the time the object was last modified.

The DTM of an object is maintained at its home node. When an object is modified by locally originating memory writes, the page modified bits in the DAT hardware record that fact; periodically, the modified bits are scanned and cause the object's DTM to be updated. If an object is modified by a remote node, eventually the object's modified pages are sent back to the home node; the paging server updates an object's DTM in response to remotely originating OSS requests to write its pages.

In addition, every node also remembers the DTM for all remote objects whose pages it has encached in its main memory. Every time a page of an object is read from or written back to its home node, the latest DTM is sent with the network reply message. Recall that the requests for page level operations are filtered through the lock key based low-level concurrency control.

4.3.3. Content Control

There are several operations explicitly provided by the AST to allow for cache management by higher level synchronization mechanisms.

1. A *conditional flush* operation expunges from the cache all pages of an object that are not from the current version of the object. (This is used by the lock manager when it discovers that the DTM associated with the cached pages of an object is different from the object's real DTM.)
2. A *get-attr* operation returns (among other attributes) the DTM of the current version of an object.
3. A *purification* operation sends copies of all modified pages of an object back to the home node of the object (but leaves the pages encached for possible later use). (This is used by the lock manager at unlock time.)
4. A *force write* variant of the purification operation causes a page to be written to permanent store on its home node; its purpose is to be a minimally sufficient toe hold with which to implement more complex atomic operations.

We shall see that using by using the AST's lock enforcement, object version, and cache content control facilites, the lock manager can effectively guarantee cache consistency for all clients who obey the system locking rules (see section 6).

create — create an object

the new object is created on the same node as 'loc-object-uid'

FUNCTION create(loc-object-uid): new-object-uid

delete — delete an object

PROCEDURE delete(object-uid)

locate — return the node address of the home node of an object

FUNCTION locate(object-uid): node-id

Figure 4: Sample FILE Operations

4.4. Location Independent OSS

Location independent access to objects is provided by the SLS and the location independent OSS. The SLS provides access to the contents of already existing objects, while the location independent OSS provides access to object attributes, and supports object creation and deletion.

The location independent OSS consists of the FILE manager, and the HINT manager. The FILE manager exports the attribute access and cache control operations of the AST to user programs in a location independent way. In addition, it implements a *create* operation to create new objects, a *delete* operation to destroy them, and a *locate* operation to return the node ID of the home node of an object (see figure 4). To create location independence, the FILE manager uses the HINT manager to determine the location of an object, then either does the operation locally (using the local or cached OSS), or uses the services of REMFILE (see below) if it must go remote.

The HINT manager is the backbone of the locating service: given an object's UID, it finds the ID of the node on which an object resides. This is the fundamental distributed algorithm in the system: no global state information is kept about object locations. Instead, a heuristic search is used to locate an object. Complete details are in [LEAC 82], including design considerations and the evolutionary history of the algorithm. To summarize briefly, the current algorithm relies heavily on hints about object location. One source is the node ID in the object's UID, another is the *hint file*. Any time a software component can make a good guess about the location of an object, it can store that guess in the hint file for later use; one particularly good source of hints is the naming server, which guesses that objects are co-located with the directory in which they are catalogued. If all hints fail to locate the object, then the requesting node's local disk is searched for the object. The algorithm works because, although it is possible for objects to do so, they rarely move from the node where they were created; and if they do, then the naming servers hint will nearly always be correct. A last resort, which would be completely sufficient, would be to accept user input into the hint file; this has not yet been implemented, as it hasn't really been needed.

4.5. Remote OSS

The remote OSS is separated into two parts which are at two very different layers of the system: the NETWORK manager, which provides remote access to the attributes and contents of already existing objects; and the REMFILE manager, which provides facilities to remotely create and delete objects. This is in contrast to the local OSS, where one set of managers provides both capabilities; the purpose is to separate the pieces of the remote OSS which are needed to resolve page faults from those which are not. This both minimizes the amount of code and data which must be permanently resident in main memory in order to implement virtual memory, and allows the REMFILE manager to use the virtual memory provided by the SLS. Both NETWORK and REMFILE are location dependent abstractions: in order to access a remote object, its location must already be known. Both of these managers can be thought of as hand-coded stubs for a simple form of remote procedure call (RPC) [BIRR 84].

The NETWORK manager is divided into a *client side* and a *server side*. The client side is used by the cached OSS to access the attributes and contents (pages) of already existing remote objects that are not in the main memory cache. When the client side is called to make a remote access, it is given the request parameters and the node ID of the home node of the object being accessed. (The request parameters always include the UID of an object, and, for a read page request, would include the page number of the object to read, for example). It packages the request parameters into a message, sends it to the given node using the low-level *socket* datagram IPC and waits for a response. Since the requests are all idempotent, it can use a very simple request-response protocol ([SPEC 82]); for more details on sockets and protocols see [LEAC 83].

The server side uses a *remote paging server* process to handle client requests, which services all remotely originating requests to read or write pages and attributes of objects on that node. The paging server has a socket assigned to it, with a well known ID, upon which it receive requests; it uses the local access mechanism to fulfill those requests. Remote paging operations are requested via **(UID, page number)** pairs only, never by disk address, and other remote operations only via UIDs; thus, a node never depends on any other node for the integrity of its object store. (This is one of the reasons the system is truly a collection of autonomous nodes – to which are added mechanisms permitting a high degree of cooperation – as distinguished from, say, a locally dispersed loosely coupled multiprocessing system.)

The REMFILE manager is also divided into client and server sides, and except that the operations are to create and delete objects, its structure is nearly identical to the NETWORK manager. The server side uses a *remote file server* process; it services client requests by calling the FILE manager to service requests. REMFILE also handles remote lock requests for the LOCK manager; see section 6.

5. Single Level Store

The single level store concept means that all memory references are logically references directly to objects. This is in contrast to a multi-level store, which typically has a "primary" store and one (or more) "secondary" store(s); only the primary store is directly accessible by programs, so they have to do explicit "I/O" operations to copy an object's from secondary to primary store before the data can be accessed. To make the distinction between primary and secondary store transparent, a single level store has to manage main memory as a cache over the object store: fetching objects (or portions of objects) from permanent store into main memory as needed, and eventually writing back modified objects (or portions thereof) to the permanent store. SLS is thus a form of virtual memory, since all referenced information need not (indeed could not) be in main memory at any one time.

Our implementation of SLS has many aspects in common with implementations of SLS for a centralized system: main memory is divided into page frames; each page frame holds one object page; main memory is managed as a write-back cache; DAT hardware allows references to encached pages at main memory speeds. If an instruction references a page of an object which is not in main memory, the DAT hardware causes a page fault, and supplies the faulting virtual address and the ID of the faulting process to software. The page fault handler finds a frame for the page; reads the page into the frame; updates the DAT related information to show that the page is main memory resident; and restarts or continues the instruction.

The SLS is implemented by the MST manager, which comes in two modules: one which is permanently resident, called MST-wired; and one which is pageable, called MST-unwired. Both manipulate a per process table, the *Mapped Segment Table* (MST), which translates a virtual address to a **(UID, page number)** pair.

MST-unwired implements a *map* operation, which adds an object to the address space of a process given the object's UID; an *unmap* operation, which removes an object; a *get-uid* operation to inquire about the objects in an address space; and a *set-touch-ahead-cnt* operation to cause read-ahead on page faults. To map an object into the address space, an entry defining the **(virtual address, UID)** association is made in the MST; unmapping just removes the appropriate entry. None of these operations are required while servicing a page fault; thus, the module can be pageable.

MST-wired implements a *touch* operation, which

---

map — make an object accessible through a virtual address space range
FUNCTION map(object-uid, protection, grow-ok, out obj-length): virt-addr

---

unmap — remove an object from the address space
PROCEDURE unmap(virt-addr)

---

getuid — get the UID of a mapped object
FUNCTION getuid(virt-addr): object-uid

---

set-touch-ahead-cnt — set demand paging cluster factor for a mapped object

Causes pages of the object to be read/written in 'cluster-size' units.

PROCEDURE set-touch-ahead-cnt(virt-addr, cluster-size)

---

touch — cause a page to be cached in main memory

The page refered to by virtual address 'virt-addr' is brought into memory, and the MMU is loaded with the 'virt-addr' <-> 'phys-page-addr' association.

PROCEDURE touch(virt-addr): phys-page-addr

---

wire — cause a page to be cached in main memory and made non-pageable
PROCEDURE wire(virt-addr): phys-page-addr

---

find — find the phyical page address for a virtual address

Optionally wire the page if 'wire-flag' is true.

PROCEDURE find(virt-addr,wire-flag): phys-page-addr

Figure 5: Sample MST Operations

for a given virtual address, causes the object page associated with it to be cached in main memory. The *touch* operation is given the virtual address of the faulting page, which it looks up in the MST to get the UID of the object mapped at that address; fetching the page is then just a request to the OSS, even if the page belongs to a remote object (see figure 5). If the touch ahead count is more than one, it will also pre-fetch succeeding pages of the object. Other operations include a *wire* operation, which is similar to *touch*, except that the page is made permanently resident as well; and a *find* operation, which returns the main memory address of a page if it is resident.

What distinguishes our implementation from a centralized one is the necessity of dealing with multiple main memory caches: in fact, one for each node in the network. This leads to the problem of synchronizing the caches in some way: of finding and fetching the most up-to-date copy of an object's page on a page fault, and of avoiding the use of "stale" pages (ones that are still in a node's cache, but have been more recently modified by another node). The objective of synchronization is to give programs a consistent view of the current version of an object in the face of (potentially) many updaters. A second objective is that the synchronization algorithm should be quite simple and need only a small data base, as it would be part of the SLS implementation and hence be permanently resident in main memory.

These objectives appeared, for practical purposes, to be mutually exclusive, so our SLS implementation does *not* guarantee consistency or the use of the current version. Instead, the implementation *does* provide operations and information from which a higher level can build a mechanism that makes the stronger guarantees. In addition, the higher level can use the virtual memory provided by SLS, and thereby be in large measure freed of the constraints mentioned earlier on the size of it and its data base. The system provides a readers/writers locking mechanism at the higher level; however, other clients are free to construct their own synchronization mechanism at this level if they do not wish to use ours.

---

lock — lock an object

See text for explanation of 'obj-mode'; 'acc-mode' is one of read, write, or read-intend-write. The boolean 'locked' is returned true if the object was locked; the caller never waits.

FUNCTION lock(object-uid, obj-mode, acc-mode): locked

---

relock — change the access mode of an lock

The boolean 'changed' is returned true if the access mode was changed.

FUNCTION relock(object-uid, acc-mode): changed

---

unlock — unlock an object

FUNCTION unlock(object-uid, acc-mode)

---

read-entry — find the lock entry record for an object

the 'lock-rec' contains the object uid, process uid of the locking process, the object and access modes of the lock, and a transaction ID (see text).

FUNCTION read-entry(object-uid): lock-rec

---

iter-entry — iterate through all locked objects

if 'volume-uid' is non-nil, restrict the iteration to just objects on that volume; 'N' starts at 0, and after each call is the index of the next entry to be returned.

FUNCTION iter-entry(volume-uid, N, object-uid): lock-rec

Figure 6: Sample LOCK Operations

6. Lock Manager

The LOCK manager provides clients of the file system the means to obtain control over an object and to block processes that wish to use the object in an incompatible way. The tools that the lock manager has

at its disposal are its own lock data base and the lock key attribute associated with each object.

The *lock* operation supports two locking modes for objects. The more familar is the many readers or single writer lock mode [HOAR 74]. A *co-writers* (co-located writers) lock mode is also provided, which makes no restrictions on the number of readers and writers, but demands that they be co-located at a single network node. This mode allows the use of shared memory semantics, but only among processes located at the same node. (Guardians [LISK 79] employ this same notion, but at the level of linguistic support for distributed computation.) For either mode, several types of access mode are supported: read, write, read with intent to write later [GIFF 79].

Other operations include: *unlock*, to unlock an object; *relock*, to change one type of lock to another without unlocking; *read-entry*, to inquire whether an object is locked, and if so, how; and *iter-entry*, to list all locked objects on a node.

An instance of the lock manager exists on every network node, and each lock manager keeps its own lock data base. This data structure records all of the objects, local or remote, that are locked by processes running on the local node. The same structure also records locks that remotely running processes are holding over local objects. Lock and unlock requests for remote objects are always sent to the home node of the object involved, and both the requesting node and the home node update their data bases. The LOCK manager uses the REMFILE manager to handle the remote requests.

The lock manager enforces compatible use of an object by not granting conflicting lock requests. However, it guards against accidental or malicious subversion of the locking mechanism by communicating its current intent to the OSS on a per object basis through the lock key. When an object is locked in a way that excludes any writers, the lock manager sets its lock key to the **readbyall** value. When an object is locked for use by a single writer, the lock manager sets its key to the node ID of the writing process. This causes both reads and writes from any other node in the network to be refused as *concurrency violations*. Today's implementation of the lock manager does not use the **writebyall** value for the lock key, however newly created objects have their lock key initialized to this value.

Locks are either granted immediately or refused; processes never wait for locks to become available, so there is no possibility of deadlock (but indefinite postponement is of course possible). This kind of locking is not meant for distributed database types of transactions, or for providing atomicity in the face of node failures, but for human time span locking uses such as file editing. For this same reason, locks are not timed out, since realistic time outs would be unreasonably long.

6.1. Cache Consistency

In a centralized virtual memory system, the main memory is the single cache over the permanent storage of a file system object. Since all of the users (both simultaneous and serial) of an object run on the same system, the memory cache is common to each of them and so no cache validation need ever be done. When the object is "unlocked" by one process, its pages may stay in the main memory cache for awhile, and if another process comes along to use the same file, that second process will always see the latest version of the object.

In the DOMAIN distributed SLS the simultaneous users of a particular file are either all readers (in which case the data they see is identical), or all processes running on the same node (in which case the main memory cache they see is the same as in the case of a single centralized system). All other simultaneous uses of a file system object are unsupported by the DOMAIN file system. However, we would like *serial* users of an object in the DOMAIN file system to each correctly see all changes made to the file by earlier users.

The simplest demonstration of the problem we faced requires two nodes **A** and **B**. Suppose a one page long file system object **O** resides on a disk that is physically connected to node **A**. A process on **B** locks the object **O** and reads its single page. That page moves through the network from **A** to **B** and ends up in the main memory of system **B**. After studying the page for some time, the process on **B** unlocks the file and goes about its business. A short time later, another process on **B** wants to read the same file **O**. It locks **O** for reading and accesses that page. We wanted the second user of **O** to be able to dependably use (or knowingly discard) the copy of the page cached in **B**'s main memory. It should be able to use that page (without refetching it from the network) if the file **O** has not been modified since the page was fetched, and it must refetch the page if the file has been modified. In this case, we needed to be able to answer the question: Did a process on **A** modify **O** between the time the page was delivered to **B** and the time the second **B** process wanted to use it? The mechanism described below allows us to efficiently answer that question, and to invalidate the cached copy if it was modified by **A**.

The version number (DTM) kept by the AST for each object can be used to synchronize main memory caches, as follows. The remote user of an object can prove the validity of his cached copy by verifying that the current DTM (as kept by the home node of the object) is identical to the DTM his node has remembered for the cached pages. Should they be different, the locally cached pages need to be invalidated. The lock manager performs this validation at lock time for all remote objects: a request to lock a remote object that is granted returns the current version number (DTM) of the object, which is used in a conditional flush operation, thereby removing stale pages of the object from the requesting nodes main memory.

A second version of the caching problem is to insure that if (extending the example above) the first **B** process to use **O** had modified the object, that the change be available to a process on **A** that wants to use the object immediately after the **B** process releases it. To guarantee correctness in this case, copies of all changed pages of remote objects are delivered back to their home node before the object is unlocked. This function is performed by the lock manager as part of the unlock function: a request to unlock a remote object first purifies the object (forces modified pages back to the home node), then frees the lock to make the object available.

Note that concurrency violations can only occur in multi-node situations: if an object is never locked, and is used by only one node, that node is the only source of version number changes, and will hence always see a consistent view of the current version. This is why the LOCK and HINT managers' state can be stored in virtual memory: the objects that store their code and data do not need to be locked because they are only used on one node.

6.2. Discussion

This two-layer approach to concurrency management has several desirable attributes. First is that it allows the (presumably) more complicated and larger higher level protocol to use the services of OSS to maintain its data base. Second is its flexibility. Changes to the higher-level lock manager can be accomplished without affecting the OSS-level implementation at all. Also, because the operations to manage the cache are exported, clients can implement their own schemes, any number of which can coexist as long as they manage disjoint sets of objects. Lastly, the burden of lock key checking assigned to the per-page operations at the OSS level is very slight compared to the lock manager's data base maintenance.

One restriction that it would be desirable to relax is that the concurrency granularity of the current implementation is at the level of entire objects. The lock key as described is insufficient for some forms of concurrency control. However, if the higher-level protocols wanted to take on the entire control task, the lock key could be set to its **writebyall** value to disable concurrency checking by the OSS-level. Note that the per-object techniques described above, but with a version number (DTM) per page, would allow page level concurrency control. We already store the DTM with each page on backing store; thus keeping one DTM per main memory page frame would suffice for this extension.

7. Naming Objects

For users, UIDs are not a very convenient means to refer to objects; for them, text string names are preferable. However, like UIDs, they should be uniform throughout the network, so that the name of an object does not change from node to node. In DOMAIN, text string names for objects are provided by a directory subsystem layered on top of the single level store. The name space is a hierarchical tree, like Multics [ORGA 72] or UNIX [RITC 74], with directories at the nodes and other objects at the leaves. A directory is just an object, with its own UID, containing primarily a simple set of associations between *component names* (strings) and UIDs. (A symbolic link facility, like that of Multics, is the other major feature of directories.) A single component name is *resolved* in the context of a particular directory by finding its associated UID (if any). The *absolute path name* of an object is an ordered list of component names. All but (possibly) the last are names of directories, which, when resolved starting from a network-wide distinguished "root" directory, lead to the UID of the object. Thus, an absolute path name, like a UID, is valid throughout the entire network, and denotes just one object. (There are other forms of path name besides the absolute form; these *relative* path names are mainly for convenience, since absolute path names are potentially very long in a large network with large numbers of objects. They are all expressible as the concatenation of some absolute path name prefix to the relative path name itself.)

8. Lessons

The first implementation of the DOMAIN system was completed in March of 1981. Since then, the system has been tested, used, and measured extensively. At this writing, the largest operational DOMAIN network system is a single token-ring network consisting of over 600 nodes, and DOMAIN installations of over 70 nodes are not uncommon. As a result of this almost four years of experience, we believe we have learned some important practical lessons – some of which validate (and in some cases vindicate) our choices and others that suggest alternative implementations.

8.1. Choosing SLS

The DOMAIN-chosen technique mapping file system objects into process address space and then turning MMU faults into object read requests of the form (UID, pageno) has been very successful. It enjoys the benefits of simplicity of implementation, stateless remote servers and the efficency of demand-paging lazy evaluation. Further, a single main memory cache management mechanism equally manages object pages for local and remote objects. Our original goal for the remote paging system was to have remote sequential file system I/O take no more than two times longer than the file I/O from a local disk. Over the years, this ratio has averaged around 1.8 to 1.

8.2. Seduction by SLS

The characteristics of network location transparency and a low penalty for remote transparent access combine to make the "map-it, use-it, unmap-it" approach to object manipulation terrifically attractive. However, we have learned that there are sometimes compelling pratical reasons for avoiding the allure of network transparency at the SLS level for some object managers that want to provide a higher level of abstraction.

Our naming server, which implements the directory hierarchy and the name-to-UID translation, was originally implemented completely on top of the location transparent SLS level. As a result, it mapped and operated on directories without regard to their location in the network. The naming server, then, did not, in fact could not, distinguish between directories on local disks and those on remote disks. As a result, the server was straightforward to implement, and as soon as it worked on local directories, it worked on remote directories.

The problem with this implementation strategy for the naming server was that the storage system (naturally) provided no layer of abstraction for the notion of directory. The SLS provided access to the raw bits of a directory to each naming server that wanted to manipulate that directory. This was fine as long as each naming server in the network could operate on directories of the same format. In practice, however, the naming servers are *not* the same on every node in the network (generally due to software updates occuring at different times) and the older naming servers are unable to handle constructs added to directories by newer naming servers running on other nodes.

Directories are an important example for a system like DOMAIN. They are permanent (stored on disk), heavily shared by multiple nodes, and most transactions on them take very little time. Also, they are likely candidates for extensions and improvements over time. Because we can never demand simultaneous update of software on every node in a network, and because we want very much to offer cross-release compatibility, we have found ourselves constrained by our original implementation.

As if that were not enough, we have found that the performance of the naming server tree-walk was significantly increased by asking the node that owned the target directory do the lookup work itself, rather than sending pages of the directory over to the requesting node. This change demanded that the naming server learn the difference between local and remote directories, and was an example of when "moving the work to the data" was a win over "moving the data to the computation."

8.3. Use Simple Protocols

The key to the attainment of our remote performance goals has been the use of light-weight problem-oriented protocols. We have taken full advantage of the relatively clean environment provided by our high-speed ring network to avoid often costly protocol supported reliability.

Operations that are idempotent (i.e. for which repeated applications have the same effect as a single application) use a *connectionless* protocol [SWIN 79] and retry often enough to achieve the desired level of reliability. Network operations to read and write attributes and pages are all of this form.

Operations which are not idempotent (i.e. which have side effects), but which naturally have some state associated with them, can often be made idempotent

using a transaction ID. Each time a client sends a new request (not a retry) to perform an operation, it chooses a new transaction ID. If an operation was performed once with a particular transaction ID, the receipt of a second request with the same ID should be rejected. File locking, for example, saves the the transaction ID of the operation which set the lock along with the lock state.

The SLS protocols we use are inexpensive because they are end-to-end protocols [SALT 80] and do not rely on the communications substrate to provide any service guarantees. Instead, each remote operation individually implements the least mechanism required by its reliability semantics.

8.4. Obtaining High Performance

Much has been written on this subject lately for distributed systems. (In particular, see [CHER 83] and [LAZO 84].) The DOMAIN file system has evolved over the years to provide as much as six times the performance of its original implementation. Certainly in the case of completely diskless nodes, but also very frequently in the case of disked nodes, the performance-critical information needed is elsewhere in the network. Our performance goals coupled with our aggressive remote-to-local ratio goal has influenced the implementation in several ways.

The disk subsystem implements fairly familiar techniques for performance enhancement including: physical locality optimizing, control structure caching, batched reads, and clustered writes. Physical locality is encouraged by the increasingly clever allocation of successive file blocks and their file maps and VTOC entries. The basic disk control structures (free-block allocation tables and VTOC control blocks) are cached in their own set of control block buffers. File page reads are "batched" at the SLS-level. Recall that in DOMAIN, all file read activity is caused by touching the bytes of the file with normal CPU instructions and thereby page-faulting on the needed page. When the SLS catches the page-fault and determines the need for some **(UID, pageno)**, it may ask the lower levels for up to 31 additional successive object pages. Most disk write operations are instigated by the page purifier process, and it tries to hand the low-levels a large collection of pages to write so that seek-ordering and rotational-ordering can be performed. In addition, for remote file system I/O, DOMAIN implements trans-network batched reads; a single read page request message may result in as many as eight reply pages in anticipation of their need. In this way, the ultimate client receives more of the benefit of disk page touch-ahead.

We have ended up caching more kinds of information than we originally expected and probably in slightly different ways. In cases where the cost of a disk access would have been barely acceptable, the cost of a network message pair in addition encouraged the use of more aggressive caching strategies.

8.5. Indefinite Postponement

In theory, the remote file server running on one node can service requests from any number of clients. In practice, however, a single server can be flooded with requests from ten, twenty, even one hundred hungry clients. Because the communications protocol layer provides no delivery guarantees to the higher layers, it blithely discards messages it receives after its assorted queues and buffers fill up. In theory, the issuer of the discarded message will send a time-out based retry and all will be well. In practice, indefinite postponement is a definite possibility. As networks get larger, and in particular as server nodes get busier, a solution that formally addresses this problem completely is needed (rather than an ad hoc approach that, for example, increases the depth of the queues periodically).

8.6. Conclusion

The essential ingredients to good performance of a distributed file system include all those things required for a good centralized file system: caching, bulk data transfer from the disk, and good object locality on the disk. In addition, the distributed file system needs more: it needs caching of remote data to avoid as many remote operations as possible; cheap, fast protocols; and bulk data transfer over the network, even when the protocols are very cheap.

REFERENCES

[APOL 81] Apollo Computer, Inc. **Apollo DOMAIN Architecture**, Apollo Computer Inc., Chelmsford, Mass., 1981.

[BIRR 82] Birrel, A. D., Levin, R., Needham, R. M., Schroeder, M. D. "Grapevine: An Exercise in Distributed Computing," *Communications of the ACM*, **25**, 4 (April 1982), pp. 260-274.

[BIRR 84] Birrel, A. D., Nelson, B. J.

"Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems*, **2**, 1 (February 1984), pp. 39-59.

[CHER 83] Cheriton, D. R., Zwaenepoel, W.
"The Distributed V Kernel and its Performance for Diskless Workstations," *Proceedings of the Ninth Symposium on Operating Systems Principles*, October 1983, pp. 128-139.

[DEC 79] Digital Equipment Corporation.
VAX 11/780 Hardware Handbook, Digital Equipment Corporation, Maynard, MA, 1979.

[FREN 78] French, R. E., Collins, R. W., Loen, L. W.
"System/38 Machine Storage Management," *IBM System/38 Technical Developments*, IBM General Systems Division, pp. 63-66, 1978.

[GIFF 79] Gifford, D. K.
"Weighted Voting for Replicated Data," *Proceedings of the Seventh Symposium on Operating Systems Principles*, December 1979, pp. 150-159.

[GORD 79] Gordon, R. L., Farr, W., Levine, P. H.
"Ringnet: A Packet Switched Local Network with Decentralized Control," *Computer Networks*, **3**, North Holland, 1980, pp. 373-379.

[HOAR 74] Hoare, C. A. R.
"Monitors: an Operating System Structuring Concept," *Communications of the ACM*, **17**, 10 (October 1974), pp. 549-557.

[HOUD 78] Houdek, M. E., Mitchell, G. R.
"Translating a Large Virtual Address," *IBM System/38 Technical Developments*, IBM General Systems Division, pp. 22-24, 1978.

[IBM 76] International Business Machines Corporation
IBM System/370 Principles of Operation, GA22-7000-5, IBM, 1976

[JANS 76] Janson, P. A.
"Using Type Extension to Organize Virtual Memory Mechanisms," *Technical Report LCS/TR-167*, Laboratory for Computer Science, M.I.T., Cambridge, Mass., September, 1976.

[KOHL 81] Kohler, W. H.
"A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems," *Computing Surveys*, **13**, 2 (June 1981), pp. 149-184.

[LAMP 80] Lampson, B. W., and Redell, D. D.
"Experience with Processes and Monitors in Mesa," *Communications of the ACM*, **23**, 2 (February 1980), pp. 105-113.

[LAZO 81] Lazowska, E., Levy, H., Almes, G., Fischer, M., Fowler, R., Vestal, S.
"The Architecture of the Eden System," *Proceedings of the Eighth Symposium on Operating Systems Principles*, December 1981, pp. 148-159.

[LAZO 84] Lazowska, E. D., Zahorjan, J., Cheriton, D. R., Zwaenepoel, W.
"File Access Performance of Diskless Workstations", *Technical Report 84-06-01*, Department of Computer Science, University of Washington, Seattle, WA, June 1984.

[LEAC 82] Leach, P. J., Stumpf, B. L., Hamilton, J. A., Levine, P. H.
"UIDs as Internal names in a Distributed File System," *Proceedings of the 1st Symposium on Principles of Distributed Computing*, Ottawa, Canada, Aug. 1982.

[LEAC 83] Leach, P. J., Levine, P. H., Douros, B. P., Hamilton, J. A., Nelson, D. L., Stumpf, B. L.
"The Architecture of an Integrated Local Network," *IEEE Journal on Selected Areas in Communication*, **SAC-1**, 5 (November 1983), pp. 842-857.

[LISK 79] Liskov, B. H.
"Primitives for Distributed Computing," *Proceedings of the Seventh Symposium on Operating Systems Principles*, December 1979, pp. 33-42.

[MCKU 84] McKusick, M. K., Joy, W. N., Leffler, S. J., Fabry, R. S.
"A Fast File System for UNIX," *ACM Transactions on Computer Systems*, **2**, 3 (August 1984), pp. 181-197.

[NEED 79] Needham, R. M.
"Systems Aspects of the Cambridge Ring," *Proceedings of the Seventh Symposium on Operating Systems Principles*, December 1979, pp. 82-85.

[NELS 81] Nelson, D. L.
"Role of Local Network in the Apollo Computer System," *Newsletter of IEEE Tech. Comm. on Distributed Processing*, **1**, 2 (December 1981), pp. 10-13.

[NELS 83] Nelson, D. L.
"Distributed Processing in the Apollo DOMAIN,"

The CAD Revolution, Second Chautauqua on Productivity in Engineering and Design, (sponsored by Schaeffer Analysis, Inc., Mont Vernon, New Hampshire). Kiawah Island, South Carolina, November 1983, pp 45-51.

[NELS 84] Nelson, D. L., Leach, P. J.
"The Architecture and Applications of the Apollo DOMAIN," *IEEE Computer Graphics and Applications*, **4**, 2 (April 1984), pp. 58-66.

[ORGA 72] Organick, E. I.
The Multics System: An Examination of Its Structure M.I.T. Press, 1972.

[POPE 81] Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G., Thiel, G.
"LOCUS: A Network Transparent, High Reliability Distributed System," *Proceedings of the Eighth Symposium on Operating Systems Principles*, December 1981, pp. 169-177.

[REDE 80] Redell, D. D., Dalal, Y. K., Horsley, T. R., Lauer, H. C., Lynch, W. C., McJones, P. R., Murray, H. G., Purcell, S. C.
"Pilot: an Operating System for a Personal Computer," *Communications of the ACM*, **23**, 2 (February 1980), pp. 81-91.

[RITC 74] Ritchie, D. M., Thompson, K.
"The UNIX time-sharing system," *Communications of the ACM*, **17**, 7 (July 1974), pp. 365-375.

[SALT 79] Saltzer, J.H., Pogran, K.T.
"A Star-Shaped Ring Network with High Maintainability," *Proceedings of the Local Area Communications Network Symposium*, Mitre Corp, May 1979, pp. 179-190.

[SALT 80] Saltzer, J. H., Reed, D. P., Clark, D. D.
"End-to-End Arguments in System Design," *Notes from IEEE Workshop on Fundamental Issues in Distributed Systems*, Pala Mesa, Ca., December 15-17, 1980.

[SALT 81] Saltzer, J. H., Clark, D. D., Pogran, K. T.
"Why a Ring," *Proceeding Seventh Data Communications Symposium*, October 27-29, 1981, pp. 211-217.

[SPEC 82] Spector, A. Z.
"Performing Remote Operations Efficiently On a Local Network," *Communications of the ACM*, **25**, 4 (April 1982), pp. 246-260.

[STUR 80] Sturgis, H., Mitchell, J., Israel, J.
"Issues in the Design and Use of a Distributed File Server," *Operating Systems Review*, **14**, 3 (July 1980), pp. 55-69.

[SWIN 79] Swinehart, D., McDaniel, G., Boggs, D.
"WFS: A Simple Shared File System for a Distributed Environment," *Proceedings of the Seventh Symposium on Operating Systems Principles*, December 1979, pp. 9-17.

[THOM 78] Thompson, K.
"UNIX Implementation," *Bell System Technical Journal*, **57**, 6 (July-August 1978), pp. 1931-1946.

[WILK 79] Wilkes, M. V., and Wheeler, D. J.
"The Cambridge Digital Communication Ring," *Proceedings of the Local Area Communications Network Symposium*, May, 1979, pp. 47-61.