Discovery and Hot Replacement of Replicated Read-Only File Systems, with Application to Mobile Computing

Erez Zadok

Computer Science Department
Columbia University

MS Thesis

Thesis Advisor: Daniel Duchamp

CUCS-036-94

October 19, 1994

Abstract

We describe a mechanism for replacing files, including open files, of a read-only file system while the file system remains mounted; the act of replacement is transparent to the user. Such a "hot replacement" mechanism can improve fault-tolerance, performance, or both. Our mechanism monitors, from the client side, the latency of operations directed at each file system. When latency degrades, the client automatically seeks a replacement file system that is equivalent to but hopefully faster than the current file system. The files in the replacement file system then take the place of those in the current file system. This work has particular relevance to mobile computers, which in some cases might move over a wide area. Wide area movement can be expected to lead to highly variable response time, and give rise to three sorts of problems: increased latency, increased failures, and decreased scalability. If a mobile client moves through regions having partial replicas of common file systems, then the mobile client can depend on our mechanism to provide increased fault tolerance and more uniform performance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The strongest trend in the computer industry today is the miniaturization of workstations into portable "notebook" or "palmtop" computers. Wireless network links [Cox91] and new internetworking technology [Ioannidis91] offer the possibility that computing sessions could run without interruption even as computers move, using information services drawn from an infrastructure of (mostly) stationary servers.

We contend that operation of mobile computers according to such a model will raise problems that require re-thinking certain issues in file system design.[1] One such issue is how to cope with a client that moves regularly yet unpredictably over a wide area.

Several problems arise when a client moves a substantial distance away from its current set of servers. One is worse latency, since files not cached at the client must be fetched over longer distances. Another problem is increased probability of loss of connectivity, since gateway failures often lead to partitions. The final problem is decreased overall system "scalability"; more clients moving more data over more gateways means greater stress on the shared network.

One obvious way to mitigate these problems is to ensure that a file service client uses "nearby" servers at all times. A simple motivating example is that if a computer moves from New York to Boston, then in many cases it is advantageous to switch to using the Boston copies of "common" files like those in */usr/ucb*. As the client moves, the file service must be able to provide service first from one server, then from another. This switching mechanism should require no action on the part of administrators (since presumably too many clients will move too often and too quickly for ad-

[1] Examples of such re-thinking can be found in [Tait91b] and [Tait91a].

ministrators to track conveniently) and should be invisible to users, so that users need not become system administrators or notice when a switch is in progress.

We have designed and implemented just such a file system — it adaptively discovers and mounts a "better" copy of a read-only file system that is fully or partially replicated. We define a better replica to be one providing better latency. Running our file service gives a mobile client some recourse to the disadvantages mentioned above. Our mechanism monitors file service latencies and, when response becomes inadequate, performs a dynamic attribute-guided search for a suitable replacement file system.

Many useful "system" file systems — and almost all file systems that one would expect to be replicated over a wide area — are typically exported read-only. Examples include common executables, manual pages, fonts, C/C++ header include files, libraries etc. Indeed, read-only areas of the file space are growing fast, as programs increase the amount of configuration information, images, sounds, and on-line help facilities.

Although our work is motivated by the perceived needs of mobile computers that might roam over a wide area and/or frequently cross between public and private networks, our work can be useful in any environment characterized by highly variable response time and/or high failure rates.

Note that for a client to continue use of a file system as it moves, there must be underlying network support that permits the movement of a computer from one network to another without interruption of its sessions. Several such schemes have been proposed [Bhagwat93, Carlberg92, Ioannidis91, Ioannidis92, Johnson93, Myles93, Myles94, Perkins93, Teraoka90, Teraoka91, Teraoka92, Teraoka93, Wada93].

The remainder of this thesis is organized as follows. In order to make a self-contained presentation, Chapter 2 provides the necessary explanations of other systems that we use in constructing ours. Chapter 3 outlines our design and by its nature includes implementation details; Chapter 4 completes our implementation notes. Chapter 5 evaluates the work. Chapter 6 provides general experiences, hints, and tips relating to kernel development accumulated while doing this work. Lastly, we mention related work in Chapter 7 and summarize in Chapter 8.

Chapter 2

Background

Our work is implemented in and on SunOS 4.1.2. We have changed the kernel's client-side NFS implementation, and outside the operating system we have made use of the Amd automounter and the RLP resource location protocol. Each is explained briefly below.

2.1 NFS

Particulars about the NFS protocol and implementation are widely known and published [Blaze92, Hitz94, Juszczak89, Juszczak94, Keith90, Keith93, Kleiman86, Macklem91, Pawlowski94, Rosen86, Rosenthal90, Sandberg85a, Sandberg85b, Schaps93, Srinivasan89, Stein87, Stern92, Sun85, Sun86, Sun89, Walsh85, Watson92].

For the purpose of our presentation, the only uncommon facts that need to be known are:

- Translation of a path name to a vnode is done mostly within a single procedure, called `au_lookuppn()`, that is responsible for detecting and expanding symbolic links and for detecting and crossing mount points.
- It is at the point during pathname translation where a mount point is crossed that we trigger much of our code.
- The name of the procedure in which an NFS client makes RPCs to a server is `rfscall()`. All of the NFS operation-specific functions (such as `nfs_close()`, `nfs_statfs()`, `nfs_getattr()`, etc.) eventually call `rfscall()` with an operation code and an opaque data structure to be interpreted by the NFS server. `Rfscall()` calls an out-of-kernel RPC

routine. It also times out that subroutine call and prints the infamous message "NFS server XXX not responding — still trying".

We have made substantial alterations to `au_lookuppn()`, and slight alterations to `rfscall()`, `nfs_mount()`, `nfs_unmount()` and `copen()`.[1]

We added two new system calls: one for controlling and querying the added structures in the kernel (`nfsmgr_ctrl()`), and the other for debugging our code (`nfsmgr_debug()`). Additional minor changes in support of debugging were made to `ufs_mount()` and `tmp_mount()`.

Finally, we added fields to three major kernel data structures: vfs and vnode structures and the open file table. Below we show these modified structures and describe their most relevant fields.

2.1.1 `struct vfs`

A vfs is the structure for a *Virtual File System* [Kleiman86]. A singly-linked list of such structures exists in the kernel, the head of which is the global `rootvfs` — a hand-crafted structure for the root filesystem. This structure was substantially modified; see Figure 2.1.[2] That is not surprising since most of our work is related to managing filesystems as a whole.

The fields of interest are:

- `vfs_next` is a pointer to the next vfs in the linked list.
- `vfs_op` is a pointer to a function-pointer table. That is, this `vfs_op` can hold pointers to UFS functions, NFS, PCFS, HSFS, etc. If the vnode interface calls the function to mount the file system, it will call whatever subfield of `struct vfsops` is designated for the mount function. That is how the transition from the vnode level to a filesystem-specific level is made; see also Section 6.1.1.3.
- `vfs_vnodecovered` is the vnode on which this filesystem is mounted.
- `vfs_flag` contains bit flags for characteristics such as whether this filesystem is mounted read-only, if the setuid/setgid bits should be turned off when exec-ing a new process, if sub-mounts are allowed, etc.

[1] `Copen()` is the common code for `open()` and `create()`.

[2] The C preprocessor (`cpp`) symbol `NFSMGR` is used to enclose our code in the kernel sources. When defined, our changes are included in the built kernel image.

```
struct vfs {
  struct vfs    *vfs_next;              /* next vfs in vfs list */
  struct vfsops *vfs_op;                /* operations on vfs */
  struct vnode  *vfs_vnodecovered;      /* vnode we mounted on */
  int           vfs_flag;               /* flags */
  int           vfs_bsize;              /* native block size */
  fsid_t        vfs_fsid;               /* file system id */
  caddr_t       vfs_stats;              /* filesystem statistics */
  caddr_t       vfs_data;               /* private data */
#ifdef NFSMGR
  char          vfs_mnt_path[MAXPATHLEN]; /* path where FS mounted */
  struct vfs    *vfs_replaces;          /* replaces which entry */
  struct vfs    *vfs_replaced_by;       /* replaced by which entry */
  int           vfs_nfsmgr_flags;       /* special flags for nfsmgr */
  struct median_glob_t vfs_median_info; /* info relevant to medians */
  int           vfs_queue_size;         /* current size of queue */
  int           vfs_trigger_ratio;      /* in %percentage */
  dft_t         vfs_dft;                /* Duplicate File Table */
#endif /* NFSMGR */
};
```

Figure 2.1: Modified `struct vfs`

- `vfs_data` is a pointer to opaque data specific to this vfs and the type of filesystem this one is. For an NFS vfs, this would be a pointer to `struct mntinfo` (located in `<nfs/nfs_clnt.h>`) — a large NFS-specific structure containing such information as the NFS mount options, NFS read and write sizes, host name, attribute cache limits, whether the remote server is down or not, and more.

The fields specific to our work are described in Section 3.4.1.1.

2.1.2 struct vnode

This structure was only slightly modified; see Figure 2.2. A vnode exists for each open file or directory.[3] The parts of the kernel that access vnodes directly are the filesystem sections. Therefore a vnode is a representation of open files from the filesystem's point of view. Only one vnode exists for each open file, no matter how many processes have opened it, or even if the file has several names (via hard or symbolic links).

```
struct vnode {
  u_short         v_flag;                   /* vnode flags */
  u_short         v_count;                  /* reference count */
  u_short         v_shlockc;                /* # of shared locks */
  u_short         v_exlockc;                /* # of exclusive locks */
  struct vfs      *v_vfsmountedhere;        /* ptr to vfs mounted here */
  struct vnodeops *v_op;                    /* vnode operations */
  union {
    struct socket   *v_Socket;              /* unix ipc */
    struct stdata   *v_Stream;              /* stream */
    struct page     *v_Pages;               /* vnode pages list */
  } v_s;
  struct vfs      *v_vfsp;                  /* ptr to vfs we are in */
  enum vtype      v_type;                   /* vnode type */
  dev_t           v_rdev;                   /* device (VCHR, VBLK) */
  long            *v_filocks;               /* File/Record locks ... */
  caddr_t         v_data;                   /* private data for fs */
#ifdef NFSMGR
  char            v_relpath[MAXPATHLEN];    /* rel path from mnt pt */
  struct timeval  v_last_used;              /* time vnode was last used */
#endif /* NFSMGR */
};
```

Figure 2.2: Modified struct vnode

Structure fields relevant to our work are:

[3]There are other entities represented as vnodes, such as devices and network communication end-points, but these are irrelevant to our work.

- `v_flag` contains bit flags for characteristics such as whether this vnode is the root of its filesystem, if it has a shared or exclusive lock, whether pages should be cached, if it is a swap device, etc.

- `v_count` is incremented each time a new process opens the same vnode.

- `v_vfsmountedhere`, if non-null, contains a pointer to the vfs which is mounted on this vnode. This vnode thus is a directory which is a mount point for a mounted filesystem.

- `v_op` is a pointer to a function-pointer table. That is, this `v_op` can hold pointers to UFS functions, NFS, PCFS, HSFS, etc. If the vnode interface calls the function to open a file, it will call whatever subfield of `struct vnodeops` is designated for the open function. That is how the transition from the vnode level to a filesystem-specific level is made; see also Section 6.1.1.3.

- `v_vfsp` is a pointer to the vfs which this vnode belongs to. If the value of the field `v_vfsmountedhere` is non-null, it is also said that `v_vfsp` is the parent filesystem of the one mounted here.

- `v_type` is used to distinguish between a regular file, a directory, a symbolic link, a block/character device, a socket, a Unix pipe (fifo), etc.

- `v_data` is a pointer to opaque data specific to this vnode. For an NFS vfs, this might be a pointer to `struct rnode` (located in `<nfs/rnode.h>`) — a remote filesystem-specific structure containing such information as the file-handle, owner, user credentials, file size (client's view), and more.

The fields specific to our work are described in Section 3.4.1.2.

2.1.3 `struct file`

This structure was also only slightly modified; see Figure 2.3. A `file` structure exists for each file opened by a process. The kernel modules that access this structure directly are those that handle processes and user contexts. Therefore a `struct file` is a representation of open files from the <u>user's</u> and <u>process'</u> points of view. The various complex interactions between `struct file` and `struct vnode` are de-mystified after the brief explanation of various fields in this structure.

Fields of use to us are:

```
struct file {
  int     f_flag;            /* see below */
  short   f_type;            /* descriptor type */
  short   f_count;           /* reference count */
  short   f_msgcount;        /* references from message queue */
  struct  fileops {
    int     (*fo_rw)();
    int     (*fo_ioctl)();
    int     (*fo_select)();
    int     (*fo_close)();
  } *f_ops;
  caddr_t f_data;   /* ptr to file specific struct (vnode/socket) */
  off_t   f_offset;
  struct  ucred *f_cred;     /* user credentials who opened file */
#ifdef NFSMGR
  char    f_path[MAXPATHLEN]; /* path name (rel. to mnt pt.) */
#endif /* NFSMGR */
};
```

Figure 2.3: Modified `struct file`

- `f_flag` contains bit flags for characteristics such as whether this file is readable/writable/executable by the current process, if it was created new or opened for appending, [non]blocking modes, and many more.

- `f_type` determines if this file is a "real" vnode or just a network socket.[4]

- `f_count` is incremented for each process referring to the same file in the *Global Open File Table*.

- `f_data` is a pointer to an opaque and specific data — depending on whether this file is a vnode or a socket.

[4]Arguably these should not have to be distinguished at this point. After all, the vnode interface should not care if something is a vnode or a network file-descriptor. This is the unfortunate result of the "hacks" that were made to the original BSD 4.3 kernels (of which SunOS 4.x was based on) when networking code was added later.

- `f_offset` is the offset into the file.

The fields specific to our work are described in Section 3.4.1.3.

There is only one *Global Open File Table* in the kernel. It has a limited size with some provisions to extend it dynamically if need be. Each `u` (user-specific) structure has an array of pointers to its open files. These `u.u_pofile_arr[idx]` are pointers into the global open file table.

When two different processes open the same file (by name or by link) they get two different `struct file` entries in the global open file table. Each file structure contains an `f_offset` field so that each process can maintain a different offset. Each file structure however, will have an `f_data` field that points to the same vnode.

The vnode structure contains the flags needed for performing advisory locking [SMCC90a, SMCC90b], and has a reference count of how many processes opened it.

Things get more complicated when a process opens a file then forks. The child inherits the same file structure pointer that the parent has. That means that if the child seeks elsewhere into the file, the parent will too, since they have the same `f_offset` field![5]

The last bit of missing information is how does the kernel tell that more than one process is sharing the same entry in the global file table. The answer is that each file structure contains an `f_count` field — a reference count similar to, but different from, the one in the vnode structure.

2.2 RLP

We use the RLP resource location protocol [Accetta83] when seeking a replacement file system. RLP is a general-purpose protocol that allows a site to send broadcast or unicast request messages asking either of two questions:

1. Do you (recipient site) provide this service?
2. Do you (recipient site) know of any site that provides this service?

A service is named by the combination of its transport service (e.g., TCP), its well-known port number as listed in `/etc/services`, and an arbitrary string that has meaning to the service. Since we search for an NFS-mountable file system, our RLP request messages contain information such

[5]Obviously code can break if people don't know of this feature. These semantics exist to support fork/dup/pipe. If a child wants to maintain a different offset into the same file, it must close and reopen it.

as the NFS transport protocol (UDP [rfc0768]), port number (2049) and service-specific information such as the name of the root of the file system.

2.3 Amd

Amd [Pendry91, Stewart93] is a widely-used automounter daemon. Its most common use is to demand-mount file systems and later unmount them after a period of disuse; however, Amd has many other capabilities.

Amd operates by mimicking an NFS server. An Amd process is identified to the kernel as the "NFS server" for a particular mount point. The only NFS calls for which Amd provides an implementation are those that perform name resolution: `lookup`, `readdir`, and `readlink`. Since a file must have its name resolved before it can be used, Amd is assured of receiving control during the first use of any file below an Amd mount point. Amd checks whether the file system mapped to that mount point is currently mounted; if not, Amd mounts it, makes a symbolic link to the mount point, and returns to the kernel. If the file system is already mounted, Amd returns immediately.

An example, taken from our environment, of Amd's operation is the following. Suppose `/u` is designated as the directory in which all user file systems live; Amd services this directory. At startup time, Amd is instructed that the private mount point (for NFS filesystem which it will mount) is `/n`. If any of the three name binding operations mentioned above occurs for any file below `/u`, then Amd is invoked.[6] Amd consults its maps, which indicate that `/u/foo` is available on server `bar`. This file system is then mounted locally at `/n/bar/u/foo` and `/u/foo` is made a symbolic link to `/n/bar/u/foo`. (Placing the server name in the name of the mount point is purely a configuration decision, and is not essential.)

Our work is not dependent on Amd; we use it for convenience. Amd typically controls the (un)mounting of all file systems on the client machines on which it runs, and there is no advantage to our work in circumventing it and performing our own (un)mounts.

[6] Amd is invoked for every file operation which traverses its automount filesystem or acts on its node. Most of these operations are empty stubs and simply return without performing any action (for example, NFS_WRITE). Once the name resolution passed the path component of the automounter, by crossing the symbolic link which Amd had presented it with, the invoking process is not at the "mercy" of the automounter any more, but whatever filesystem server it crossed over to.

2.3.1 How Our Work Goes Beyond Amd

Amd does not already possess the capabilities we need, nor is our work a simple extension to Amd. Our work adds at least three major capabilities:

1. Amd keeps a description of where to find to-be-mounted file systems in "mount-maps." These maps are written and maintained by administrators and are static in the sense that Amd has no ability for automated, adaptive, unplanned discovery and selection of a replacement file system.

2. Because it is only a user-level automount daemon, Amd has limited means to monitor the response of `rfscall()` or any other kernel routine.

   Many systems provide a tool, like `nfsstat`, that returns timing information gathered by the kernel. However, `nfsstat` is inadequate because it is not as accurate as our measurements, and provides weighted average response time rather than measured response time. Our method additionally is less sensitive to outliers, and measures both short-term and long-term performance.

3. Our mechanism provides for transparently switching *open* files from one file system to its replacement.

Amd might be considered the more "natural" place for our user-level code, since Amd makes similar mount decisions based on some criteria. Some coding could have been saved and speedups made if we placed our user-level management code inside Amd. However, we saw two main problems with this approach:

1. Amd is maintained by different people, and we would have to continually keep Amd and our programs in sync.

2. Not everyone uses Amd as their automounter, if any at all. By placing our code inside Amd we would have forced other administrators to run and maintain Amd as well.

Chapter 3

Design

The key issues we see in this work are:

1. Is a switching mechanism really needed? Why not use the same file systems no matter where you are?
2. When and how to switch from one replica to another.
3. How to ensure that the new file system is an acceptable replacement for the old one.
4. How to ensure consistency if updates are applied across different replicas.
5. Fault tolerance: how to protect a client from server unavailability.
6. Security: NFS is designed for a local "workgroup" environment in which the space of user IDs is centrally controlled.

These issues are addressed below. Not all implementation details are mentioned in this section. The rest of the implementation which did not have direct impact on the design of our system is detailed in Chapter 4.

3.1 Demonstrating the Need

We contend that adaptive client-server matchups are desirable because running file system operations over many network hops is bad for mobile and/or wide area computing in three ways: increased latency, increased failures, and decreased scalability. It is hard to ascertain exact failure rates and load on shared resources without undertaking a full-scale network study; however, we were able to gather some key data to support our claim. We performed a simple study to measure how latency increases with distance.

First, we used the *traceroute* program[1] to gather <*hop-count, latency*> data points measured between a host at Columbia and several other hosts around the campus, city, region, and continent. The results are listed in Table 3.1. Latencies were measured by a Columbia host, which is a Sun-4/75 equipped with a microsecond resolution clock. The cost of entering the kernel and reading the clock is negligible, and so the measurements are accurate to a small fraction of a millisecond.

Hostname	**NFS Time** (seconds)	**Ping Time** (mSec)	**No. Hops**
`ground.cs.columbia.edu`	4.55	4	1
`tune.cs.columbia.edu`	8.60	6	2
`sol.ctr.columbia.edu`	10.68	9	3
`omnigate.clarkson.edu`	339.26	482	10
`gatekeeper.dec.com`	97.90	198	12
`mvb.saic.com`	157.58	287	13
`pit-manager.mit.edu`	138.01	158	13
`wuarchive.wustl.edu`	401.60	979	18
`zaphod.ncsa.uiuc.edu`	183.06	227	19

Table 3.1: NFS Read Latency, Network Hop Count, and Ping Time of Various Hosts (1KB block size)

We used the *ping* program with 1024-byte packets because the default size of *ping* packets (56 bytes) is too small to mimic NFS traffic.

Next, we mounted NFS file systems that are exported Internet-wide by certain hosts. We measured the time needed to copy 1MB from these hosts using a 1KB block size. We took measurements with two different datagram sizes: 40 bytes and 8KB, hypothesizing that distance degrades performance worse when packets are large. Forty bytes is the minimum packet sent by *traceroute*; eight kilobytes was chosen since it is the size of NFS block transfers in our facility. A typical result is plotted in Figure 3.1. Latency jumps by almost two orders of magnitude at the tenth hop, which

[1] Written by Van Jacobson and widely available by anonymous ftp from `ftp.uu.net` in `/networking/ip/trace/traceroute_pkg.tar.Z`.

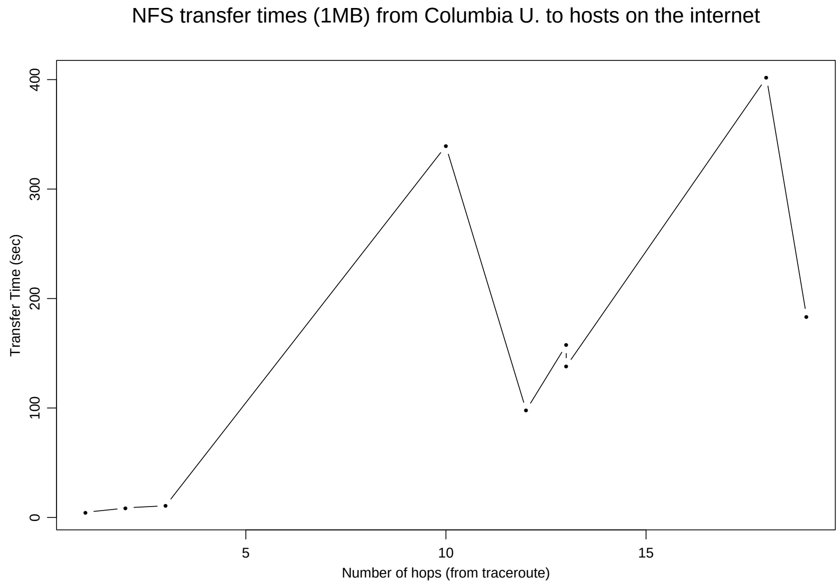

Figure 3.1: NFS Read Latency vs. Network Hop Count (1KB block size)

represents the first host outside Columbia. Each plotted point is the median of 10 trials. We performed similar studies, measuring latency to 25 hosts that are each a number of hops away from Columbia; all results were similar to those plotted in Figure 3.1.

We conclude that (current) NFS performance over long distances is poor; too many UDP packets are lost resulting in retransmission requests of whole blocks. However, if we could constrain ourselves to using NFS servers in our "neighborhood," we could keep performance reasonable.

3.2 When to Switch

We have modified the kernel so that `rfscall()` measures the latency of every NFS *lookup* and maintains a per-filesystem data structure storing a number of recently measured latencies.

We chose to time the *lookup* operation rather than any other operation or mixture of operations for two reasons. The first is that *lookup* is the most frequently invoked NFS operation. We experimented and found that

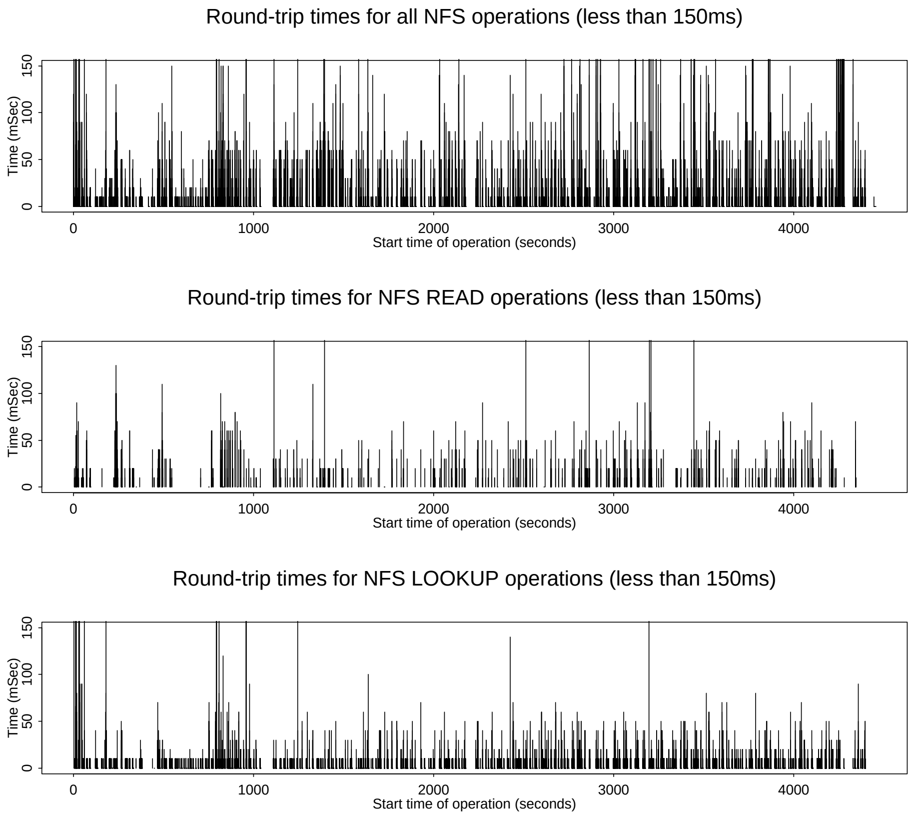

Figure 3.2: Variability and Latency of NFS Operations

other calls did not generate enough data points to accurately characterize latency. The second reason is that *lookup* exhibits the least performance variability of the common NFS operations.[2] Limiting variability of measured server latencies is important in our work, since we want to distinguish transient changes in server performance from long-term changes. It took lots of experimentation to come to a meaningful and stable measurement mech-

[2]These are the NFS_GETATTR, NFS_READ, and sometimes NFS_NULL operations. The latter might have been better suited for our needs as it exhibits the least variability, but it does not occur often enough. Note also that the *null* operation does not account for the whole performance of the server (for example, including disk performance), but mostly characterizes the network.

anism. At the outset of our work, we measured variances in the latency of the most common NFS operations and discovered huge swings, shown in Figure 3.2, even in an extended LAN environment that has been engineered to be uniform and not to have obvious bottlenecks. The measured standard deviations were 1027 msec for all NFS operations, 2547 msec for *read*, and only 596 msec for *lookup*.

After addition of each newly measured *lookup* operation, the median latency is computed over the last 30 and 300 calls. We compute medians because medians are relatively insensitive to outliers. We take a data point no more than once per second, so during busy times these sampling intervals correspond to 30 seconds and 5 minutes, respectively.

The signal to switch is when, at any moment, the short-term median latency exceeds the long-term median latency by a factor of 2. This policy provides insurance against anomalies like ping-pong switching between a pair of file systems: a file system can be replaced no more frequently than every 5 minutes. Looking for a factor of two difference between short-term and long-term medians is our attempt to detect a change in performance which is substantial and "sudden," yet not transient. The length of the short-term and long-term medians as well as the ratio used to signal a switch are heuristics chosen after experimentation in our environment.[3] All these parameters can be changed from user level through a debugging system call that we have added; see Section 4.2.

3.3 Locating a Replacement

When a switch is triggered, `rfscall()` starts a non-blocking RPC[4] out to our user-level process named *nfsmgrd*. The call names the guilty file server, the root of the file system being sought, the kernel architecture, the current median (round-trip time) values, and any mount options affecting the file system. *Nfsmgrd* uses this information to compose and broadcast an RLP request. The file system name keys the search, while the server name is a filter: the search must not return the same file server that is already in use.

The RLP message is received by the *nfsmgrd* at other sites on the same broadcast subnet. To formulate a proper response, an *nfsmgrd* must have a view of mountable file systems stored at its site and also mounted file systems that it is using — either type could be what is being searched

[3]A better trigger function would take into account the absolute latency; see Section 8.1.

[4]Non-blocking operation is provided by a special kernel implementation of Sun RPC.

for. Both pieces of information are trivially accessible through `/etc/fstab`, `/etc/exports`, and `/etc/mtab`.

The *nfsmgrd* at the site that originated the search uses the first response it gets; we suppose that the speed with which a server responds to the RLP request gives a hint about its future performance. (The Sun Automounter [Callaghan89] makes the same assumption about replicated file servers.) If a read-only replacement file system is available, *nfsmgrd* instructs Amd to mount it[5] and terminates the out-of-kernel RPC, telling the kernel the names of the replacement server and file system. The flow of control is depicted in Figure 3.3.

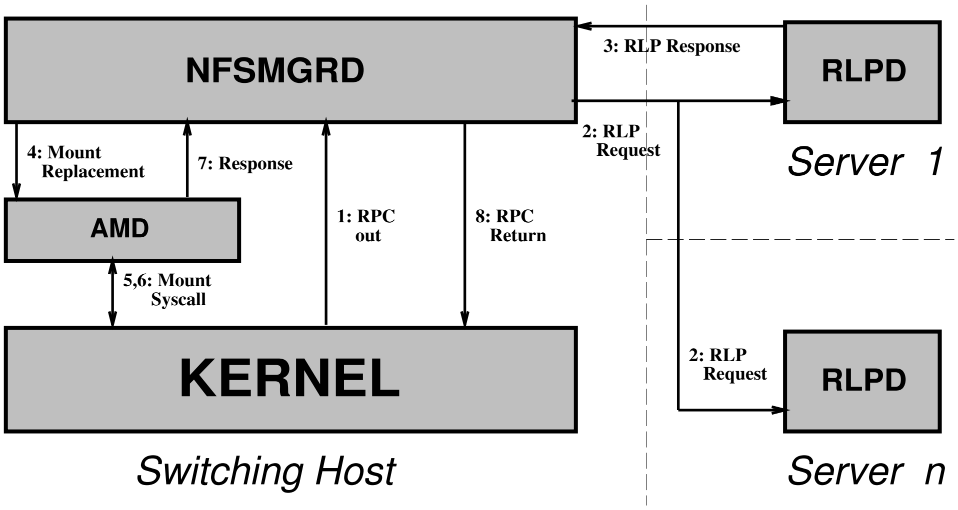

Figure 3.3: Flow of Control During a Switch

3.4 Using the Replacement

Once a replacement file system has been located and mounted, all future attempts to open files on the replaced file system will be routed to the replacement whenever the two files are identical. Also, in all cases for which it is possible, *open files* on the replaced file system will be switched to their

[5] Amd provides an RPC interface used by its query client `amq` that we use to query and control Amd.

equivalents on the replacement. We describe these two cases in Sections 3.4.2 and 3.4.3, respectively.

3.4.1 Relevant Changes to Kernel Data Structures

In order to accommodate file system replacement, we have added some fields to three important kernel data structures: `struct vfs`, which describes mounted file systems; `struct vnode`, which describes open files; and `struct file`, which describes file descriptors.

3.4.1.1 `struct vfs`

See Figure 2.1 for a complete listing of this structure, and Section 2.1.1 for explanation of important fields in the structure as it existed before our changes. The fields we added to `struct vfs` are:

- `vfs_mnt_path` is valid in every vfs; it stores the absolute pathname of the file system's mount point. It is used to locate a vfs based on its mount-point name, and in the construction of full pathnames from the root to the actual file being replaced (concatenating this field with the relative pathname of the file from mount-point).

- `vfs_replaces` is valid in the vfs structure of the replacement file system; it points to the vfs structure of the file system being replaced.

- `vfs_replaced_by` is valid in the replaced file system's vfs struct; it points to the vfs structure of the replacement file system. When a replacement file system is mounted, our altered version of `nfs_mount()` sets the replaced and replacement file systems pointing to each other.

- `vfs_nfsmgr_flags` is valid for any NFS file system. One flag indicates whether the file system is managed by *nfsmgrd* and is turned on by our `nfsmgr_ctrl()` system call. The idea is to provide the administrators of the client host the ability to control which filesystems are managed by our code, and suspend any use of it when needed.

  Another flag indicates whether a file system switch is in progress. Part of the filesystem switching process involve making asynchronous out-of-kernel RPCs. The calls invoke local daemons, which may call other local daemons and remote servers. Obviously the initiating process is blocked until all answers are returned to the kernel, or if timeouts are reached. That opens the possibility that another process might hit a threshold, and cause another search for a replacement filesystem to the same one in progress. To avoid this problem, only one process is allowed to be the (inadvertent) initiator of a filesystem search; all other processes see that the flag is on, and block until the flag is off.

- `vfs_median_info` contains almost all of the pertinent information about the performance of the file system, including the 300 most recent `nfs_lookup()` response times, the 30 most recent measures, the trigger ratio, many pointers in support of the double-threading nature of this data structure, and more.

- The field `vfs_dft` is the *Duplicate File Table* (DFT). This per-filesystem table lists which files in the replacement file system have been compared to the corresponding file on the original file system mounted by Amd. Only equivalent files can be accessed on the replacement file system. The mechanism for making comparisons is described in Section 3.4.2.

  The size of the DFT is fixed (but changeable) so that new entries inserted will automatically purge old ones. This is a simple method to maintain "freshness" of entries.

  The DFT is a hash table whose entries contain a file pathname relative to the mount point, a pointer to the vfs structure of the replacement file system, and an extra pointer for threading the entries in insertion order. This doubly-threaded data structure permits fast lookups keyed by pathname and quick purging of older entries.

- `vfs_queue_size` tells how many entries were allocated in the DFT. It is used especially when a DFT needs to be purged; we improve speed by not freeing and then re-allocating kernel memory for DFT entries.

3.4.1.2 `struct vnode`

See Figure 2.2 for a complete listing of this structure, and Section 2.1.2 for explanation of important fields in the structure as it existed before out changes. The two fields we added to `struct vnode` are:

- `v_relpath` is valid in every vnode which may be replaced. It stores the relative pathname of the file from the root of the mounted filesystem. It is used to construct full pathnames of two files to be compared, and in determining the filename of an opened file during the process of hot replacement. See Sections 3.4.2 and 3.4.3.

- `v_last_used` is valid for every NFS vnode. This is the last time that `rfscall()` made a remote call on behalf of this vnode. This information is used in our "hot replacement".

3.4.1.3 `struct file`

See Figure 2.3 for a complete listing of this structure, and Section 2.1.3 for explanation of important fields in the structure as it existed before our changes. The only field we added to `struct file` is:

- `f_path` is valid in every open file on an NFS filesystem. It is very similar to the `v_relpath` field added to the vnode structure. This field is necessary to distinguish among several opened file-descriptors to the same hard-linked file, when the `open()` calls used different path names. See Section 2.1.3 for details of the interaction between the file and vnode structures in the SunOS 4.x kernel.

3.4.2 After Replacement: Handling New Opens

When Amd mounts a file system it makes a symlink from the desired location of the file system to the mount point. For example, `/u/foo` would be a symlink pointing to the real mount point of `/n/bar/u/foo`; by our local convention, this would indicate that server `bar` exports `/u/foo`. Users and application programs know only the name `/u/foo`.

The information that `bar` exports a proper version of `/u/foo` is placed in Amd's mount-maps by system administrators who presumably ensure that the file system `bar:/u/foo` is a good version of whatever `/u/foo` should be. Therefore, we regard the information in the client's Amd mount-maps as authoritative, and consider any file system that the client might mount and place at `/u/foo` as a correct and complete copy of the file system. We call this file system *the master copy*, and use it for comparison against the replacement file systems that our mechanism locates and mounts.

The new open algorithm is shown in Figure 3.4. After a replacement file system has been mounted, whenever name resolution must be performed for any file on the replaced file system, the file system's DFT is first searched for the relative pathname.

If the DFT contains an entry for the pathname, then the file on the replacement file system has already been compared to its counterpart on the master copy. A field in the DFT tells if the comparison was successful or not. If not, then the rest of the pathname has to be resolved on the master copy. If the comparison was successful, then the file on the replacement file

```
open() {
    examine vfs_replaced_by field to see if there is
        a replacement file system;
    if (no replacement file system) {
        continue name resolution;
        return;
    }
    if (DFT entry doesn't exist) {
        create and begin DFT entry;
        call out to perform file comparison;
        finish DFT entry;
    }
    if (files equivalent) {
        get vfs of replacement from vfs_replaces field;
        continue name resolution on replacement file system;
    } else
        continue name resolution on master copy;
}
```

Figure 3.4: New Open Algorithm

system is used; in that case, name resolution continues at the root of the replacement file system.

If the DFT contains no entry for the pathname, then it is unknown whether the file on the replacement file system is equivalent to the corresponding file on the master copy. The two files must be compared.

To test equivalence, `au_lookuppn()` calls out of the kernel to *nfsmgrd*, passing it the two host names, the name of the file system, and the relative pathname to be compared. `Au_lookuppn()` gathers all this information as follows:

1. The filesystem switch trigger happens while resolving a pathname, finding that the `v_vfsmountedhere` field of that current vnode is not NULL, indicating that this is a mount point. The `vfs_mnt_path` of this vfs is recorded as the mount point of the master filesystem.

2. If the `vfs_replaced_by` field of the vfs of that vnode is not NULL,

there is a replacement filesystem for this vfs. The `vfs_mnt_path` of it is recorded as the mount point of the replacement filesystem.

3. Since `au_lookuppn()` did not complete resolving the full pathname, the unresolved part is the relative (to the mount point) pathname to the file being resolved.

Each of the two mount point names (one per host) is prepended to the name of the file to be compared, yielding two full pathnames to (hopefully) the same file, but on two different filesystems.

At this point a partial DFT entry is constructed, and a flag in it is turned on to indicate that there is a comparison in progress and that no other process should initiate the same comparison.[6]

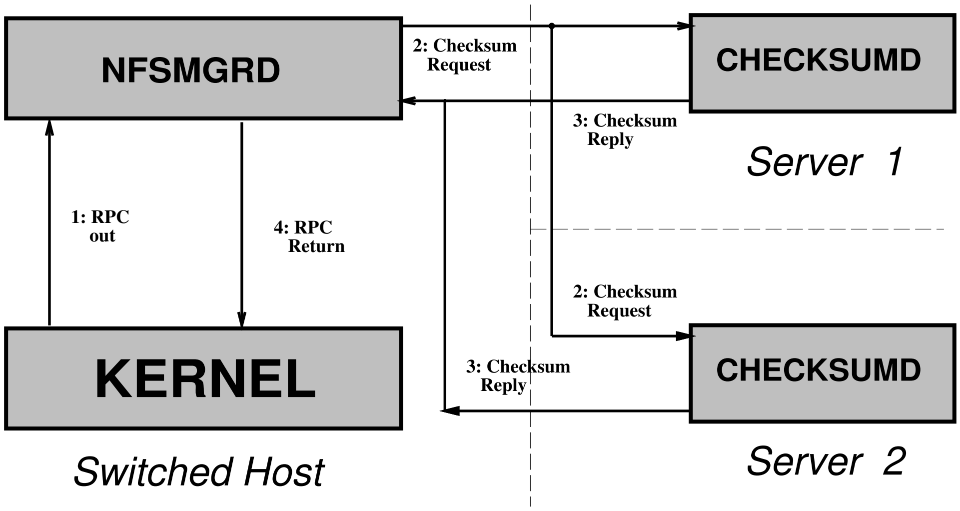

Figure 3.5: Flow of Control During File Comparison

Nfsmgrd then applies, at user level, whatever tests might be appropriate to determine whether the two files are equivalent. This flow of control is depicted in Figure 3.5. A more complete example of our new pathname resolution scheme is shown in Figure 3.6. Presently, we are performing file checksum comparison: *nfsmgrd* calls a *checksumd* daemon on each of the

[6]This avoids the need to lock the call out to *nfsmgrd*.

file servers, requesting the checksum of the file being compared. *Checksumd*, which we have written for this work, computes MD4 file checksums [Rivest91] on demand and then stores them for later use; checksums can also be pre-computed and stored.

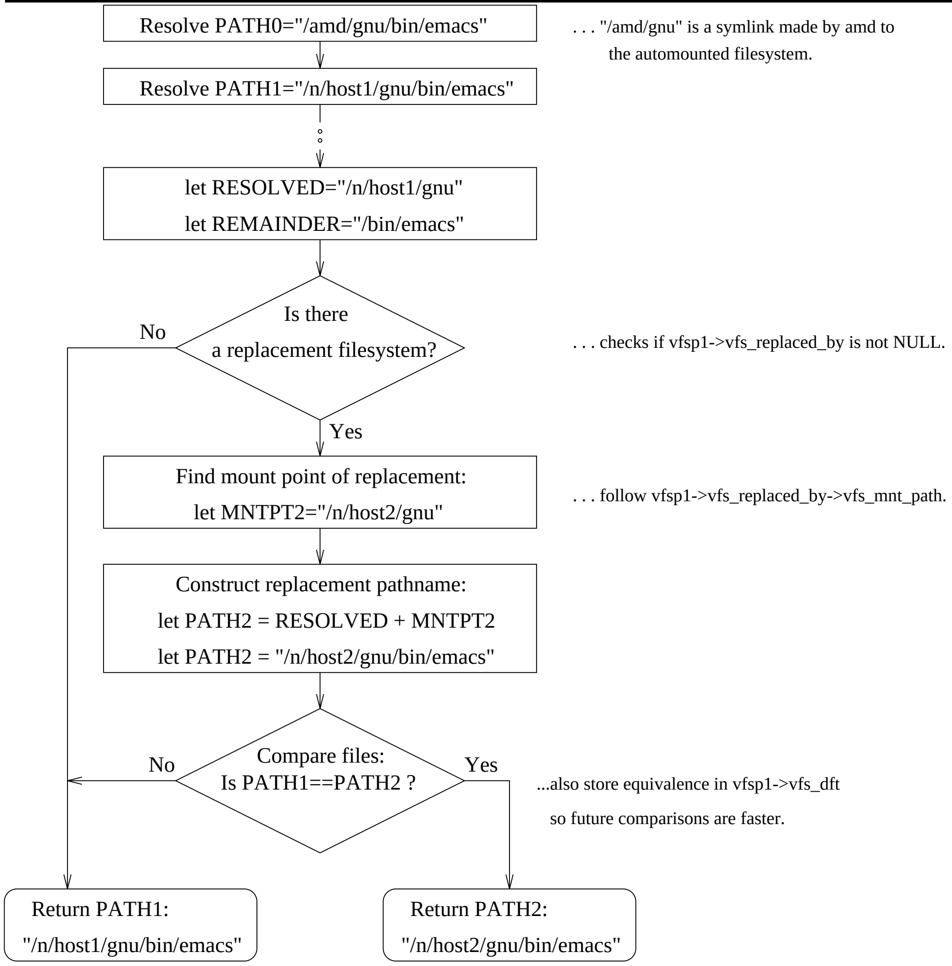

Figure 3.6: An Example of the New Pathname Resolution Algorithm

Nfsmgrd collects the two checksums, compares them, and responds to the kernel, telling `au_lookuppn()` which pathname to use, indicating the file on the replacement file system if possible. `Au_lookuppn()` completes the construction of the DFT entry, unlocks it, and marks which vfs is the proper one to use whenever the same pathname is resolved again.

In this fashion, all new pathname resolutions are re-directed to the replacement file system whenever possible.

Note that the master copy could be unmounted (e.g., Amd by default unmounts a file system after a few minutes of inactivity[7]), and this would not affect our mechanism. The next (new) use of a file in that file system would cause some master copy to be automounted, before any of our code is encountered.

3.4.3 After Replacement: Handling Files Already Open

When a file system is replaced, it is possible that some files will be open on the replaced file system at the moment when the replacement is mounted. Were these processes to continue to use the replaced file system, several negative consequences might ensue. First, since the replacement is presumed to provide faster response, the processes using files open on the replaced file systems experience worse service. Second, since the total number of mounted file systems grows as replacements happen, the probability rises that some file system eventually becomes unavailable and causes processes to block. Further, the incremental effect of each successive file system replacement operation is reduced somewhat, since files that are open long-term do not benefit from replacement. Finally, kernel data structures grow larger as the number of mounted file systems climbs. Motivated by these reasons, we decided to switch *open files* from the replaced file system to the replacement file system whenever the file on the replacement file system is equivalent to that on the master copy.

Although this idea might at first seem preposterous, it is not, since we restrict ourselves to read-only file systems. We assume that files on read-only file systems[8] change very infrequently and/or are updated with care to guard against inconsistent reads.[9] Whether operating conditions uphold

[7] Note also that a filesystem cannot be unmounted, even if `nfs_umount` is called, as long as there are open file-descriptors in use on that filesystem. That means that even if we can avoid using a filesystem because we have a replacement for it, we may not be able to release the kernel resources it occupies.

[8] That is, they are *exported* as read-only to some hosts (including our client hosts), although they might be exported as read-write to others.

[9] An example of "careful update" is provided by the SUP utility [Shafer92]. SUP transfers the new file to a temporary name, renames the target file to another temporary name, renames the newly transferred file to the final name, and then unlinks the old file which was also renamed. This is meant to make sure that any open descriptors on the old file can still access it and will not encounter possible paging problems.

this assumption or not, the problem of a file being updated[10] while being read exists independently of our work, and our work does not increase that danger.

We allow for a replacement file system to be itself replaced. This raises the possibility of creating a "chain" of replacement file systems. Switching vnodes from the old file system to its replacement limits this chain to length two (the master copy and the current replacement) in steady state. For example, if host A is replaced by host B, and host B is replaced by C, we "short circuit" the replacement information stored in the vfs structure and allow for C to directly replace A; see Sections 3.4.1.1 and 4.2. This saves us unnecessary comparisons and delays while we have to contact more hosts. After all, host B has already been determined as "bad", so there is no need to use it any longer.

Hot replacement requires knowing pathnames. Thanks to our changes, the vfs structure records the pathname it is mounted on and identifies the replacement file system; also, the relative pathname of the file is stored in the file table entry. This information is extracted, combined with the host names, and passed out to *nfsmgrd* to perform comparison, as described above. If the comparison is successful, the pathname on the replacement file system is looked up, yielding a vnode on the replacement file system. This vnode simply replaces the previous vnode in all entries in the open file table. This results in a switch the next time a process uses an open file descriptor.

The "hot replacement" code scans through the global open file table, keying on entries by (a pointer to the) vfs. Once an entry is found that uses the file system being replaced, a secondary scan locates all other entries using the same vnode. In a single entry into the kernel (i.e., "atomically"), all file descriptors pointing to that vnode are switched, thereby avoiding complex questions of locking and reference counting. A table scan is necessary because while vnodes point to their vfs, the reverse is not true.

Hot replacement is made possible by the statelessness of NFS and by the vfs/vnode interfaces within the kernel. Since the replaced server keeps no state about the client, and since the open file table knows only a pointer to a vnode, switching this pointer in every file table entry suffices to do hot replacement.

An interesting issue is at which time to perform the hot replacement of vnodes. Since each file requires a comparison to determine equivalence, switching vnodes of all the open files of a given file system could be a lengthy process. The four options we considered are:

[10] That is, updated by a host to which the file system is exported read-write.

1. Switch as soon as a replacement file system is mounted (the early approach).
2. Switch only if/when an RPC for that vnode hangs (the late approach).
3. Switch if/when the vnode is next used (the "on-demand" approach).
4. Switch whenever a daemon instructs it to (the "flexible" approach).

The decision to switch earlier or later is affected by the tradeoff that early switching more quickly switches files to the faster file system and improves fault tolerance by reducing the number of file systems in use, but possibly wastes effort. Vnode switching is a waste if the switched vnode will not be used again. Early switching also has the disadvantage of placing the entire delay of switching onto the single file reference that is unlucky enough to be the next one.

We chose the "flexible" approach of having a daemon make a system call into the kernel which then sweeps through the open file table and replaces some of the vnodes which can be replaced. We made this choice for three reasons. First, we lacked data indicating how long a vnode lingers after its final use. Second, we suspected that such data, if obtained, would not conclusively decide the question in favor of an early or late approach. Third, the daemon solution affords much more flexibility, including the possibility of more "intelligent" decisions such as making the switch during an idle period.

We emphasize that the system call into the kernel switches "some" of the vnodes, since it may be preferable to bound the delay imposed on the system by one of these calls. Two such bounding policies that we have investigated are, first, switching only N vnodes per call, and, second, switching only vnodes that have been accessed in the past M time units. Assuming that file access is bursty (a contention supported by statistics [Ousterhout85, Ruemmler93]), the latter policy reduces the amount of time wasted switching vnodes that will never be used again. We are currently using this policy of switching only recently used vnodes; this policy makes use of the `v_last_used` field that we added to the vnode structure.

Note that if the time since last use is chosen such that vnodes used since the last run of the daemon are switched, then processes will not remain hung indefinitely waiting for a remote file system (assuming that the file causing the hang can be replaced). Since `rfscall()` will have timestamped the vnode at the beginning of the hung call, the next run of the daemon will cause an attempt to switch the vnode of the hung call. If we want to make sure processes are "unhung" sooner we could increase the frequency of the daemon's run.

However, vnodes that cannot be replaced still pose a rather difficult problem. They can easily bring a machine down, especially if the processes that are hung on the RPC call are vital to the stability of the system.

3.5 Security

The NFS security model is the simple uid/gid borrowed from UNIX, and is appropriate only in a "workgroup" situation where there is a central administrative authority. Transporting a portable computer from one NFS user ID domain to another presents a security threat, since processes assigned user ID X in one domain can access exported files owned by user ID X in the second domain.

Accordingly, we have altered `rfscall()` so that every call to a replacement file system has its user ID and group ID both mapped to "nobody" (i.e., value -2). Therefore, only world-readable files on replacement file systems can be accessed. Of course any files left owned by "nobody" will be at risk.

3.6 Code Size

Counting blank lines, comments, and debugging support, we have written close to 11,000 lines of C. More than half is for user-level utilities: 1200 lines for the RLP library and daemon, 3200 for *nfsmgrd*, 700 lines for *checksumd*, and 1200 lines for a control utility (called *nfsmgr_ctl*). New kernel code totals 4000 lines, of which 800 are changes to SunOS, mostly in the NFS module. The remaining 3200 lines comprise the four modules we have added: 880 lines to deal with storing and computing medians; 780 lines are the "core NFS management" code, which performs file system switching, pathname storage and replacement, and out-of-kernel RPC; 540 lines to manage the DFT; and 1000 lines to support the `nfsmgr_ctl` system call.

The `nfsmgr_ctl` system call allows query and control over almost all data structures and parameters of the added facility. We chose a system call over a kmem program for security. This facility was used heavily during debugging; however, it is meant also for system administrators and other interested users who would like to change these "magic" variables to values more suitable for their circumstances. See Section 4.2 for more details.

Chapter 4

Implementation

4.1 RLP

We implemented only the parts of the Resource Location Protocol which we needed:

- RLP_MSG_WHO_ANYWHERE_PROVIDES
- RLP_MSG_DOES_ANYONE_PROVIDE
- RLP_MSG_THEY_PROVIDE

See an introduction of RLP in section 2.2.

We use RLP in its "miscellaneous" message format, one allowing the client to send an arbitrary byte-stream, which is meaningful only to the server which can decode it. The actual fields in the arbitrary field are as follows:

1. `msg_size`: the total size of this message.

2. `hostname`: the hostname of the filesystem which is now considered bad and which we are trying to replace. If the same server which is bad receives this RLP request, it is forbidden from answering for itself, to avoid picking the same bad host as a replacement for itself.

3. `filesystem`: the name of the filesystem for which a replacement is sought. The name of the filesystem (for example `/usr/gnu`) is insufficient to describe everything that is in that filesystem. That is why we also relied on comparing individual files to determine their existence and equivalence. See Section 3.4.2.

4. `flags`: arbitrary flags describing the replaced filesystem. This field was not in use in our work, and is there for future expansion. One possible use for it was to tell the remote resource server that the replaced filesystem needs to be mounted with special privileges (say the `anon=0` option). A remote server might then decide not to reply to such a client for security reasons.

5. `architecture`: the host architecture of the replaced filesystem is used to advise the client whether the server is likely to supply it with the same type files it needs. The server reports back if its architecture is identical to that of the client or not. The client then may choose to select or decline using that server.

4.2 Management and Control Facilities

We thought of several ways to query and control various features of our system. One mechanism which might have not required any kernel modifications was to use the kernel memory access mechanism, `/dev/kmem`, and write a program that "walks" the different structures in the kernel, retrieving information as needed, and very carefully modifying others. There were several problems with this approach:

1. **Security:** it is better for the kernel to protect itself against malicious processes rather than rely on user-level programs not corrupting vital kernel data. A program that can read kernel memory might be abused or misused into reading or modifying parts of the kernel for which it was not designed.

2. **Atomicity:** some of the management and control operations we allow require the operation go to completion, and that no other process could access that data being modified (i.e., exclusive lock). Otherwise kernel data will be left in a corrupt state. Kernel facilities are provided to system calls to make them more atomic, and it is a lot easier to lock out certain parts of our code while they are being modified with the appropriate *spl* level.[1]

3. **Portability:** some operating systems such as CMU's "UX" server for Mach-3 don't have a kmem interface, making any future port of our system more difficult. Eventually a system-call mechanism would have had to be used.

[1] The *spl* are kernel routines that *Set Process Lock* levels.

Accordingly we decided to add a new system call to the kernel. A system call is a considered a mechanism far "cleaner" than kmem. (However, it does require access to kernel source.)

Our system call is designed with "object oriented" like programming style in mind. The calling convention of `nfsmgr_ctrl()` take 3 arguments:

1. **obj:** is the code for the object we want to manipulate or query.
2. **cmd:** is the command code we want to apply to the object.
3. **args:** is a pointer to a control structure which contains the necessary information which needs to be passed to the kernel. It also provides allocated space in the user-space for operations which need to return data back to the user process.

Following are the various "objects" which could be manipulated using the system call, and the commands which could be applied to them:

1. `NMO_NONE`: no (N)FS (M)anager (O)bject needs to be manipulated. This one exists mostly for trapping errors, and serves as the "null" call.
2. `NMO_DFT`: manage the *Duplicate File Table*. The allowed operations are:
   - `NMC_READ`: this (N)FS (M)anager (C)ommand will read the full contents of the DFT into user space. A client we wrote using this system call displays the full DFT in tabular form.
   - `NMC_WRITE`: this command will overwrite arbitrary entries in the DFT.
   - `NMC_ADD`: add entries to the DFT.
   - `NMC_DEL`: remove entries from the DFT, given a pathname.
   - `NMC_CLEAR`: clear a single entry from the DFT, given a table index.
   - `NMC_RESET`: clear all the entries from the DFT.
3. `NMO_RFSI`: manage the RFSI, the *Replacement File System Information*. To facilitate the ability to query a filesystem and find a replacement for it, each vfs contains pointers to the filesystem it replaces and to the one that replaces it — the fields `vfs_replaces` and `vfs_replaced_by` in the vfs structure; see Figure 2.1. The operations allowed on the RFSI are identical to those for `NMO_DFT`.

4. NMO_DFT_SIZE: Size of the DFT. Allowed operations are:
   - NMC_READ: return the current number of entries in the DFT.
   - NMC_GETMAX: return the size of the allocated DFT. This is the maximum number of entries that could fit in the DFT.
   - NMC_SETMAX: set the maximum size of the DFT. This could be used to extend or truncate (via kernel "realloc" routines) the length of the DFT.
5. NMO_RFSI_SIZE: Size of the RFSI. The only allowed operation is NMC_READ which returns the current size of the RFSI. The maximum size of the RFSI cannot be controlled externally. It is equal to the number of vfs structures that exist in the kernel, and grows or shrinks with them.
6. NMO_MGMT: NFS management flag per filesystem. The only three operations allowed here are NMC_READ, NMC_SETON, and NMC_SETOFF. They return the current value of this flag, set it to on, or turn it off, respectively.
7. NMO_MEDIAN_SET: Median set is the long-term queue of measured round-trip times of the nfs_lookup operations. Allowed operations are:
   - NMC_READ will read the current median value of the long queue.
   - NMC_GETMIN will read the current number of medians stored in the queue. The queue gets reset each time a replacement is made, and no new replacements are made until the queue is full again. This tells us how many data points we have already accumulated.
   - NMC_GETMAX tells us the value of the most recent median entered, the one at the very top of the queue.
8. NMO_MEDIAN_SUBSET: Median subset is the short-term queue of measured round-trip times. Allowed operations are identical to those allowed on the full-size median queue (NMC_MEDIAN_SET).
9. NMO_TRIGGER_RATIO: Trigger ratio between the median of the short-term queue and the long-term one. Allowed operations are NMC_READ for checking the current ratio, and NMC_WRITE for changing it.
10. NMO_SWITCH_NOW: Forced switching flag. This controls a bit in the vfs structure's field vfs_nfsmgr_flags, as described in Section 3.4.1.1. The only three operations allowed here are NMC_READ, NMC_SETON, and

NMC_SETOFF. They return the current value of this flag, set it on, or turn it off, respectively. Setting this flag to on will force a switching of this filesystem the next time an NFS lookup operation is performed on it.

The last field of nfsmgr_ctrl() is used not just to return values to the user, but to pass to the kernel whatever necessary information it requires. Many of the operations listed above are specific to a particular filesystem, such as DFT operations. For these operations, the name of the filesystem (mount point) must be passed to the kernel. The system call will search for the vfs with the same name in the vfs_mnt_path field, and if found, will apply the operation requested to the DFT of the vfs in question.

4.3 Debugging Facilities

This nfsmgr_debug() system call is very simple. It passes an integer to the kernel, and returns a status back. The integer is a bit-mask for turning on debugging (mostly using printf()s) for at various parts of our code. Of course, prior to this mechanism working, we had to wrap parts of debugging code with the appropriate bit-mask tests.

Chapter 5

Evaluation

This system was implemented and received use on a limited number of (Sun-4, 40MHz SparcStation II) machines.

The goal of this work is to improve overall file system performance — under certain circumstances, at least — and to improve it enough to justify the extra complexity. For this method to really work, it must have:

1. Low overhead latency measurement between switches.
2. A quick switch.
3. Low overhead access to the replacement after a switch.
4. No anomalies or instabilities, like ping-pong switching.
5. No process hangs due to server failures when a replacement is available.
6. No security or administrative complications.

We have carried out several measurements aimed at evaluating how well our mechanism meets these goals.

The *overhead between switches* is that of performance monitoring. The added cost of timing every `rfscall()` we found too small to measure. The cost of computing medians could be significant, since we retain 300 values. But we implemented a fast incremental median algorithm that requires just a negligible fraction of the time in `nfs_lookup()`. The kernel data structures are not so negligible: retaining 300 latency measurements costs about 2KB per file system. The reason for the expansion is the extra pointers that must be maintained to make the incremental median algorithm work. The extra fields in the `struct vfs`, `struct vnode`, `struct file` are small, with the exception of the DFT, which is large. The current size of each (per-filesystem) DFT is 60 slots which occupy a total of 1KB-2KB on average.

Our measured *overall switch time* is approximately 3 sec. This is the time between the request for a new file system and when the new file system is mounted (messages 1-8 in Figure 3.3). Three seconds is comparable to the time needed in our facility to mount a file system whose location is already encoded in Amd's maps (about 1-2 seconds in our environment), suggesting that most of the time goes to the mount operation.

The *overhead after a switch* consists mostly of doing equivalence checks outside the kernel; the time to access the vfs of the replacement file system and DFT during `au_lookuppn()` is immeasurably small. Only a few milliseconds are devoted to calling *checksumd*: 5-7 msec if the checksum is already computed. This call to *checksumd* is done once and need not be done again so long as a record of equivalence remains in the DFT. If checksums have to be computed, it would take about 5-6 msec more per 1MB of data already loaded in memory, for the comparisons to complete.

A major issue is how long to cache DFT entries that indicate equivalence. Being stateless, NFS does not provide any sort of server-to-client cache invalidation information. Not caching at all ensures that files on the replacement file system are always equal to those on the master copy; but of course the repeated comparisons somewhat defeat the purpose of using the replacement. We suppose that most publicly-exported read-only file systems have their contents changed rarely, and thus one should cache to the maximum extent. Accordingly, we manage the DFT cache by LRU.

As mentioned above, *switching instabilities* are all but eliminated by preventing switches more frequently than every 5 minutes.

In one experiment we performed, an *Emacs* process was started from a filesystem on a slow server to be replaced. We simulated the slowing of the server by artificially raising the round-trip times of the NFS lookup operations going to this server. In the midst of our editing within *Emacs*, the trigger ratio was reached. A flurry of activity ensued, and a replacement was found and mounted. Within only a few seconds the open vnode for the *Emacs* process was replaced for us by one on the replacement filesystem, and our process was released from its hung state for us to resume editing.

Since we map the uid of all outgoing NFS calls to replacement filesystem to that of "nobody," we avoid potential security problems with users being able to access files owned by others in a different administrative domain. We chose suitable defaults for the variables of our system such that no changes need be made; however, if necessary, privileged users and system-administrators can use our control facilities to tune system parameters.

5.1 Experience

5.1.1 What is Read-Only

Most of the files in our facility reside on read-only file systems. However, sometimes one can be surprised. For example, *Emacs* is written to require a world-writable lock directory. In this directory *Emacs* writes files indicating which users have which files in use. The intent is to detect and prevent simultaneous modification of a file by different processes. A side effect is that the "system" directory in which *Emacs* is housed (at our installation, `/usr/local`) must be exported read-write.

Deployment of our file service spurred us to change *Emacs*. We wanted `/usr/local` to be read-only so that we could mount replacements dynamically. Also, at our facility there are several copies of `/usr/local` per subnet, which defeats *Emacs*' intention of using `/usr/local` as a universally shared location. We re-wrote *Emacs* to write its lock files in the user's home directory since (1) for security, our system administrators wish to have as few read-write system areas as possible and, (2) in our environment by far the likeliest scenario of simultaneous modification is between two sessions of the same user, rather than between users.

Note also that it is not necessary that the whole filesystem being switched be exported read-only, only the parts that are requested. That depends on the ability of the system to allow this. For example, Solaris 2.x [SMCC90c] allows arbitrary parts of a filesystem to be exported with different permissions, but SunOS 4.x [SMCC90d] only allows sibling subdirectories of a filesystem to be exported with different permissions.

5.1.2 Suitability of Software Base

5.1.2.1 Kernel

The vfs and vnode interfaces in the kernel greatly simplified our work. In particular, hot replacement proved far easier than we had feared, thanks to the vnode interface. The special out-of-kernel RPC library also was a major help. Nevertheless, work such as ours makes painfully obvious the benefits of implementing file service out of the kernel. The length and difficulty of the edit-compile-reboot-debug cycle, and the primitive debugging tools available for the kernel were truly debilitating. Recent developments in kernel technologies such as layered kernel modules in Solaris 2.x [SMCC92a] and multi-server systems such as the GNU Hurd [Bushnell94] or the CMU "US" server would have been tremendous to us.

5.1.2.2 RLP

RLP was designed in 1983, when the evils of over-broadcasting were not as deeply appreciated as they are today and when there were few multicast implementations. Accordingly, RLP is specified as a broadcast protocol. A more up-to-date protocol would use multicast. The benefits would include causing much less waste (i.e., bothering hosts that lack an RLP daemon) and contacting many more RLP daemons. Not surprisingly, we encountered considerable resistance from our bridges and routers when trying to propagate an RLP request. A multicast RLP request would travel considerably farther.

RLP uses UDP by default. Since UDP does not guarantee delivery of packets, there is a realistic limit on how large a single packet could be for reliable delivery; large packets are more likely not to get delivered correctly and will require a retransmission. Also, most applications which use UDP avoid exceeding 8KB, because most UDP implementations have been written to match the default buffer sizes of NFS, and hardly ever allow you to reach the protocol limit of 64KB as described in [Stevens94]. The information necessary for our attribute-guided search for filesystems could have reached these limits.[1] A better protocol, using TCP for example, will remove these limitations. It would also be necessary to send out only minimal information, and then exchange further information on a "need to know" basis, and if asked, with remote resource servers; a hierarchical organization such as that successfully used in [Dyer88, Mockapetris87a, Mockapetris87b, SMCC93, Noor94] might be more suitable.

Finally, having exclusively used RLP's "catch-all" message format (see Section 4.1) and not what it was primarily designed for is further evidence of its unsuitability for this work.

5.1.2.3 NFS

NFS is ill-suited for "cold replacement" (i.e., new opens on a replacement file system) caused by mobility, but is well suited for "hot replacement" because of its statelessness.

NFS' lack of cache consistency callbacks has long been bemoaned, and it affects this work since there is no way to invalidate DFT entries. Since we restrict ourselves to slowly-changing read-only files, the danger is assumed to be limited, but is still present. Most newer file service designs include

[1] Under SunOS, the maximum path name length alone, MAXPATHLEN, is 4096 bytes long.

cache consistency protocols. However, such protocols are not necessarily a panacea. Too much interaction between client and server can harm performance, especially if these interactions take place over a long distance and/or a low bandwidth connection. See [Tait92] for a design that can ensure consistency with relatively little client-server interaction.

The primary drawback of using NFS for mobile computing is its limited security model. Not only can a client from one domain access files in another domain that are made accessible to the same user ID number, but even a well-meaning client cannot prevent itself from doing so, since there is no good and easy way to tell when a computer has moved into another uid/gid domain.

Our work was based on version 2 of the NFS protocol. Version 3 [Pawlowski94, SMCC94] of the protocol fixes some of the problems of the current version. For example, it allows for use of TCP, dynamically adjusting buffer sizes, and asynchronous writes — which would definitely improve its performance over wide-area networks.

NFS version 3 also provides better support for security:

- A new NFS operation, ACCESS, allows the client to request from the server a list of access rights per filehandle. The server will check the permissions — possibly through a set of ACLs and uid/gid mappings — and decide which of the access requests to grant and which not. This is a more flexible and finer grained method than the version 2 of the protocol, in which the only reliable way to determine if a client had access to the server's files was to try the operation and see if it failed.

- A new authentication model has been added, using the Kerberos authentication protocol [Steiner88, Lunt90, Bellovin91].

Chapter 6

Experiences

6.1 Experiences in Kernel Development

During the time we worked on this system, we have gained considerable experience developing and testing kernel code. It has proven to be a challenging task. These comments were borne out of working in the SunOS 4.x operating system, but they are valuable nonetheless to many other environments based on a monolithic kernel.

6.1.1 Debugging

The largest single problem when developing kernel code is debugging. Each time a new test had to be made, we had to edit kernel sources, rebuild the kernel executable (*/vmunix*), install it, reboot, wait for the machine to come up, and then start our tests. This whole cycle, for a SparcStation II averaged around 30 minutes for very small code changes. (That might explain why this work spanned over several years.)

6.1.1.1 `printf()`s

The best method for debugging kernels which we came to use was copious `printf` statements in small code sections that had to be debugged at the moment. We had to be careful about how many and where we placed these print statements. For example, busy sections such as the name resolution function (`au_lookuppn()`) are bad places to insert them, because the amount of output that will get generated by the kernel — which gets printed on the console and added to a syslog [SMCC90e] daemon — is so voluminous that the machine spends most of its time displaying debugging output, and user processes are pushed down the scheduling priority.[1]

However, not even using `printf` helped us at times. Output has to pass through a kernel buffer, out to the syslog mechanism, and then to the console (or wherever `/etc/syslog.conf` directs it to). When a kernel panic occurs, the kernel printf buffer almost certainly has some output that has not been flushed to the console. That output is lost when the machine panics. Unfortunately, that output is the most critical to have, because it is the debugging information just leading to the panic. The best ways to avoid these problems were to be extremely careful when writing kernel code. See Section 6.1.2.

Our solution was to introduce a new system call dedicated to turning kernel debugging on and off for any section of our code, and for querying or even changing information that is accumulated by the filesystem code. See Sections 4.2 and 4.3.

6.1.1.2 Kernel Debuggers

We have tried other methods for debugging kernels, such as using *kadb*. But we found these to be cumbersome and greatly lacking in flexibility as compared to user-level debuggers such as *gdb*.[2]

6.1.1.3 Source Browsing

SunOS 4.1.2's kernel sources number almost a half a million of C code lines. No one person could be an expert in every part of this large system. Our work mainly concentrated on less than one tenth of that amount, and on the whole, no more than one fifth of the code (about 73,000 lines) had to be looked at to achieve our goals. In one respect, this exemplifies just how much modularity there is in the kernel. On the other hand, we had to learn how the SunOS kernel operates all on our own, testing one section at a time. A lot of time was spent placing `printf` statements at various points in the kernel, and checking what output was produced.

That is how we learned the execution flow in the kernel. It is difficult to know at any given point how did the kernel get there. One of the main reasons is the so-called "object-oriented" programming style the SunOS kernel has. Many routines are not called directly, but as a consequence of a macro

[1] Console output is considered a high-priority event in SunOS 4.x.

[2] *Gdb* has the ability to debug kernels over the network and/or from processes, but it is only possible for micro-kernel based operating-systems, such as Mach 3.0 [Stallman94].

expansion on a field of a structure containing opaque data and generic structures full of pointers to functions. One of these functions is dereferenced, and then called on the actual data point it was passed. Here is an example from `<sys/vnode.h>` showing how this is achieved:

```
#define VOP_GETATTR(VP,VA,C) (*(VP)->v_op->vn_getattr)(VP,VA,C)
```

The operation might have been applied to any vnode, but at that level the knowledge of what filesystem that vnode belonged to was lost. The best way we found to recover that information was to compare the addresses of the pointers to the functions — in this example, comparing `*(VP)->v_op` with the global `&nfs_vnodeops`. That way we could tell the vnode in question is an NFS one.

Another problem was the lack of documentation specific to SunOS kernels or even more general about "modern" operating system resembling SunOS. The books available to us at the time were outdated, too broad, or inapplicable [Bach86, Leffler89, Tanenbaum87].

6.1.2 Coding Practices

When coding in the kernel, we found many of our assumptions and experiences accumulated over years of user-level programming to be false. These proved to be futile; the slightest problem in the kernel causes a panic, followed by a long kernel-dump of memory pages, and the obvious need to fix the code.

These are some of our recommendations when writing kernel code:

- **pointers:** Be very careful with pointers. Don't ever assume that the value of a pointer you are passing around is what you thought it was. Always check to make sure. When your code is working flawlessly, you can remove extraneous checks to speed it up. Many times, due to memory allocation and/or alignment problems, pointers and their data get corrupted. Also important is to initialize values of any allocated data (static, automatic, etc.). Most user-level compilers these days will make sure values of stack or heap allocated storage is zeroed first. That is not so in the kernel (for speed reasons), so you must initialize all values yourself. Besides, initialization is just good programming practice.

- **memory:** In the kernel you don't have infinite amounts of memory, not even virtual memory. Every byte you use comes out of a fixed amount of physical memory the kernel takes away from the rest of the system when it starts up. You can easily run off the end of the kernel memory by calling too many nested functions (recursion is not a good idea either), by using many (or large) automatic or static variables, and of course, by forgetting to call the proper kernel *free* routine for something you have allocated. For example, using large automatic strings allocated for deeply nested functions accounted for several days of frustrating debugging, faced with the obscure "watchdog reset — rebooting" messages.

- **output:** as mentioned in section 6.1.1.1, it is not recommended to generate too much output to the console for several reasons: you want the kernel print buffer to flush in time, too much output in frequently called kernel functions may slow the system manyfold, and it is easier to look at less debugging output at the critical moment than many pages of useless information. Other methods we used included "timed output". That is, we turn on verbose output from some code section for only a short period of time, which automatically turns itself off.

- **backup kernel:** always keep a backup kernel image in available for use. We always left a known working kernel in `/vmunix.good`, which we were able to specify at boot time in case our newly installed kernel failed to boot or crashed frequently.

6.2 Vendor Bugs

Even if you are an experienced C programmer and wrote bug-free kernel code, you may still get kernel panics. Kernel code of vendors is hardly bug-free. In our case, hundreds of patches exist for various versions of SunOS 4.x.

The system administrators at our site have installed most of these patches, and were constantly installing new ones. However, the sources we were working from were those of the original unpatched system. That meant that the kernels we were building from original sources did not include any bug fixes. We had our kernels crash several times due to known bugs which we had no source fixes to.

In a few occasions, we tried to install binary kernel patches to object modules for those files which we knew we were not modifying, and working under the assumption that it is better to fix some bugs than none at all. That assumption only worked half of the time. Often, large patches are distributed in collections known as "Jumbo Kernel Patches". These are such extensive patches that they span many kernel modules, and make incompatible changes that must be coordinated among different code sections. Installing only a few of them, and expecting the rest to be generated from sources often did not work. If we were lucky, the kernel would not build due to missing symbols. If we were unlucky, the kernel linked, but failed to run at some stage, sometimes several days after the system was rebooted.

Chapter 7

Related Work

Underlying our work is the idea that in order for mobile computing to become the new standard model of computing, adaptive resource location and management will have to become an automatic function of distributed services software. The notion of constantly-networked, portable computers running modern operating systems is relatively new. Accordingly, we know of no work other than our own (already cited) on the topic of adaptive, dynamic mounting.

The Coda file system [Satyanarayanan90] supposes that mobile computing will take place in the form of "disconnected operation," and describes in [Kistler91] a method in which the user specifies how to "stash" (read/write) files before disconnection and then, upon reconnection, have the file service run an algorithm to detect version skew. Coda can be taken as a point of contrast to our system, since the idea of disconnection is antithetical to our philosophy. We believe trends in wireless communication point to the ability to be connected any time, anywhere. Users may *decide* not to connect (e.g., for cost reasons) but will not be *forced* not to connect (e.g., because the network is unreliable or not omnipresent). We call this mode of operation *elective connectivity.*

An obvious alternative to our NFS-based effort is to employ a file system designed for wide-area and/or multi-domain operation. Such file systems have the advantages of a cache consistency protocol and a security model that recognizes the existence of many administrative domains. Large scale file systems include AFS [Howard88] and its spinoffs, Decorum [Kazar90] and IFS (Institutional File System) [Howe92]. Experiments involving AFS as a "nation-wide" file service have been going on for years [Spector89]. This effort has focused on stitching together distinct administrative domains so as to provide a single unified naming and protection space. However, some changes are needed to the present authentication model in order to support the possibility of a mobile client relocating in a new domain. In particular, if the relocated client will make use of local services, then there should be some means whereby one authentication agent (i.e., that in the new domain) would accept the word of another authentication agent (i.e., that in the client's home domain) regarding the identity of the client. Such could be made possible by cooperating *Identification Servers* [Johns93a, Johns93b]. For example, a client's RLP request could cause a server receiving it to call the client's `identd` server to find the identity of the user who initiated the request. The server may decide to deny the request if identification could not be made, or perhaps choose to ask another server (perhaps a master identification server back at the client's home base) for identity confirmation. Once an identity is confirmed, the server may log that information for future reference or for tracking in case of suspected break-in attempts.

The IFS project has also begun to investigate alterations to AFS in support of mobile computers [Honeyman91]. Specifically, they are investigating cache pre-loading techniques for disconnected operation and transport protocols that are savvy about the delays caused by "cell handoff" — the time during which a mobile computer moves from one network to another.

Solaris 2.3's CacheFS [SMCC92b] allows for effective caching and synchronization of data between a client and NFS server. The main benefits of such a caching mechanism is the ability to use smaller, lighter, and less power-consuming disk drives — especially important for mobile computers.

Plan 9's *bind* command has been designed to make it easy to mount new file systems. In particular, file systems can be mounted "before" or "after" file systems already mounted at the same point. The before/after concept replaces the notion of a search path. Plan 9 also supports the notion of a "union mount" [Pike91, Presotto92]. Several filesystem could be unified into one large one. Whenever files are identical, a client host might get parts of these files from any number of servers used to form the union. If one such server becomes inaccessible, but others still do, the client will continue to receive uninterrupted file service.

The Plan 9 bind mechanism is a more elegant alternative to our double mounting plus comparison. However, a binding mechanism — even an unusually flexible one such as that of Plan 9 — addresses only part of the problem of switching between file systems. The harder part of the problem is determining *when* to switch and *whom* to switch to.

Chapter 8

Conclusion

We have described the operation, performance, and convenience of a transparent, adaptive mechanism for file system discovery and replacement. The adaptiveness of the method lies in the fact that a file service client no longer depends solely on a static description of where to find various file systems, but instead can invoke a resource location protocol to inspect the local area for file systems to replace the ones it already has mounted.

Such a mechanism is generally useful, but offers particularly important support for mobile computers that may experience drastic differences in response time as a result of their movement. Reasons for experiencing variable response include:

1. Moving beyond the home administrative domain and so increasing the "network distance" between client and server.

2. Moving between high-bandwidth private networks and low-bandwidth public networks (such movement might occur even within a small geographic area).

While our work does not address how to access replicated read/write file systems or how to access one's home directory while on the move, our technique does bear on the problems of the mobile user. Specifically, by using our technique, a mobile user can be relieved of the choice of either suffering with poor performance or devoting substantial local storage to "system" files.[1] Instead, the user could rely on our mechanism to continuously locate copies of system files that provide superior latency, while allocating all or most of his/her limited local storage to caching or stashing read/write files such as those from the home directory.

Our work is partitioned into three modular pieces: heuristic methods for detecting performance degradation and triggering a search; a search technique coupled with a method for testing equivalence versus a master copy; and a method for force-switching open files from the use of vnodes on one file system to vnodes on another (i.e., "hot replacement"). There is little interrelationship among these techniques, and so our contributions can be viewed as consisting not just of the whole, but also of the pieces. Accordingly, we see the contributions of our work as:

1. The observation that file system switching might be needed and useful.
2. The idea of an automatically self-reconfiguring file service, and of basing the reconfiguration on measured performance.
3. Quantification of the heuristics for triggering a search for a replacement file system.
4. The realization that a "hot replacement" mechanism should not be difficult to implement in an NFS/vnodes setting, and the implementation of such a mechanism.

8.1 Future Work

There are several directions for future work in this area.

The major direction is to adapt these ideas to a file service that supports a more appropriate security model. One part of an "appropriate" security model is support for cross-domain authentication such that a party from one domain can relocate to another domain and become authenticated in that domain. Another part of an appropriate security model should include

[1] One might suppose that a "most common subset" of system files could be designated and loaded. However, specifying such a subset is ever harder as programs depend on more and more files for configuration and auxiliary information. This approach also increases the user's responsibility for system administration, which we regard as a poor way to design systems. One possible solution is a caching filesystem such as [SMCC92b]. With a caching filesystem, only a small working set of files most frequently used are stored on a smaller local disk, alleviating the need to go to a remote server for file access. Only when rarely used files are requested, a file search on remote hosts could be conducted, perhaps using our switching mechanism.

accounting protocols allowing third parties to advertise and monitor (i.e., "sell") the use of their exported file systems. Within the limited context of NFS, a small step in the right direction would be a mechanism that allows clients (servers) to recognize servers (clients) from a different domain. The most recent version of Kerberos contains improved support for cross-domain authentication, so another step in the right direction would be to integrate the latest Kerberos with NFS, perhaps as originally sketched in [Glover93, Steiner88, Lunt90, Bellovin91].

Another desirable idea is to convert from using a single method of exact file comparison (i.e., *checksumd*) to per-user, possibly inexact comparison. For example, object files produced by *gcc* contain a timestamp in the first 16 bytes; two object files may be equal except for the embedded timestamps, which can be regarded as an insignificant difference. Another example is that data files may be equal except for gratuitous differences in floating-point format (e.g., 1.7 vs. 1.7000 vs. 1.70e01). Source files may be compared ignoring comments and/or white space. Intelligent comparison programs like *diff* or *spiff* [Nachbar88] know how to discount certain simple differences.

Other extensions and improvements to our work include:

- Adding an absolute measure of performance to the trigger function. Currently it will only switch if a relative change for the worse had occured, but not if persistent yet bad performance exists. More generally, we believe that a better trigger function could be found.

- Converting RLP from a broadcast protocol to a multicast protocol.

- Reimplementing RLP in an environment that supports out-of-kernel file service implementations (e.g., multi-server Mach 3.0).

- Upon first access to a replacement filesystem, retrieve the checksum information for the whole filesystem to the local client. This is usually a small index file which will not take much to store locally on the home system. It would improve performance because for each new pair-comparison, there would be no need for `checksumd` to send a message to the host of the master copy — already determined as unsuitable — to get the checksum information for the file in question. The checksum information will reside locally.

- Investigate the possibility of breaking processing out of hung RPCs. If we could do that, and find a replacement to the hung fileserver (without of course accessing it again), we could switch the vnodes of these processes to replacement servers as well.

- Using MD5 [Rivest92] checksums rather than MD4, because they are slightly more secure.
- Finish the implementation of RLP, for the sake of completeness.

Acknowledgments

This work was supported in part by the New York State Science and Technology Foundation's Center for Advanced Technology in Computers and Information Systems; by a National Science Foundation CISE Institutional Infrastructure grant, number CDA-90-24735; and by the Center for Telecommunications Research, an NSF Engineering Research Center supported by grant number ECD-88-11111.

We thank an anonymous member of the Usenix program committee for the suggestion to use file checksums. We thank the Usenix program committee, especially David Rosenthal and Matt Blaze, for valuable advice about presentation and emphasis.

Bibliography

[Accetta83] M. Accetta. Resource location protocol. Request for comments 887. ARPA Network Working Group, December 1983.

[Bach86] M. J. Bach. *The Design of the Unix Operating System*. Prentice Hall, 1986. ISBN 0-13-201757-1.

[Bellovin91] S. M. Bellovin and M. Merritt. Limitations of the Kerberos Authentication System. *USENIX Conference Proceedings* (Dallas, TX), pages 253–67. USENIX, Winter 1991. FTP - research.att.com:/dist/kerblimit.usenix.ps.Z; local - kerblimit.usenix.ps.

[Bhagwat93] P. Bhagwat and C. E. Perkins. A mobile networking system based on Internet Protocol (IP). *Proceedings USENIX Symposium on Mobile & Location-Independent Computing* (Cambridge, Mass.), pages 69–82. USENIX, August 1993.

[Blaze92] M. Blaze and R. Alonso. Issues in massive-scale distributed file systems. *Proceedings of the Usenix File Systems Workshop*, pages 135–6, May 1992.

[Bushnell94] M. I. Bushnell. The HURD: Towards a New Strategy of OS Design. *GNU's Bulletin*. Free Software Foundation, January 1994. Copies are available by writing to `gnu@prep.ai.mit.edu`.

[Callaghan89] B. Callaghan and T. Lyon. The Automounter. *USENIX Conference Proceedings* (San Diego, CA), pages 43–51. USENIX, Winter 1989.

[Carlberg92] K. G. Carlberg. A Routing Architecture that Supports Mobile End Systems. *MILCOM Conference Proceedings*, volume 1, pages 159–64. MILCOM, October 1992.

[Cox91] D. C. Cox. A Radio System Proposal for Widespread Low-power Tetherless Communication. *IEEE Transactions on Communications*, **39**(2):324–35. IEEE, February 1991.

[Dyer88] S. P. Dyer. The Hesiod Name Server. *USENIX Conference Proceedings* (Dallas, TX), pages 183–9. USENIX, Winter 1988.

[Glover93] F. Glover. A Specification of Trusted NFS (TNFS) Protocol Extensions. Technical report. Internet Engineering Task Force, 1 March 1993.

[Hitz94] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. *USENIX Conference Proceedings* (San Francisco, CA), pages 235–46. USENIX, Winter 1994.

[Honeyman91] P. Honeyman. Taking a LITTLE WORK Along. Technical report. CITI, University of Michigan, Ann Arbor, MI, U.S., 1991.

[Howard88] J. H. Howard, Michael L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, **6**(1):51–81, February 1988.

[Howe92] J. Howe. Intermediate File Servers in a Distributed File System Environment. In *CITI Report 92-4*. CITI, June 1992.

[Ioannidis91] J. Ioannidis, D. Duchamp, and G. Q. Maguire. IP-based Protocols for Mobile Internetworking. *SIGCOMM'91* (Zurich, Switzerland), pages 235–45. ACM Press, 1991.

[Ioannidis92] J. Ioannidis, D. Duchamp, S. Deering, and G. Q. Maguire. Protocols for Supporting Mobile IP Hosts. Draft RFC. IETF Mobile Hosts Working Group, June 1992.

[Johns93a] M. St.Johns. Identification Protocol. Requests for Comments 1413. Network Working Group, February 1993.

[Johns93b] M. St.Johns and M. Rose. Identification MIB. Requests for Comments 1414. Network Working Group, February 1993.

[Johnson93] D. Johnson. Ubiquitous Mobile Host Internetworking. *4th Workshop on Workstation Operating Systems (WWOS-IV)* (Napa, U.S.), 1993.

[Juszczak89] C. Juszczak. Improving the Performance and Correctness of an NFS Server. *USENIX Conference Proceedings* (San Diego, CA), pages 53–63. USENIX, Winter 1989.

[Juszczak94] C. Juszczak. Improving the Write Performance of an NFS Server. *USENIX Conference Proceedings* (San Francisco, CA), pages 247–59. USENIX, Winter 1994.

[Kazar90] M. L. Kazar, B. W. Leverett, O. T. Anderson, V. Apostolides, B. A. Bottos, S. Chutani, C. F. Everhart, W. A. Mason, S.-T. Tu, and E. R. Zayas. DEcorum File System Architectural Overview. *USENIX Conference Proceedings* (Anaheim, CA), pages 151–64. USENIX, Summer 1990.

[Keith90] B. E. Keith. Perspectives on NFS File Server Performance Characterization. *USENIX Conference Proceedings* (Anaheim, CA), pages 267–78. USENIX, Summer 1990.

[Keith93] B. E. Keith and M. Wittle. LADDIS: The Next Generation in NFS File Server Benchmarking. *USENIX Conference Proceedings* (Cincinnati, OH), pages 111–28. USENIX, Summer 1993.

[Kistler91] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Asilomar, Pacific Grove, CA), pages 213–25. Association for Computing Machinery SIGOPS, 13 October 1991.

[Kleiman86] S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. *USENIX Conference Proceedings* (Atlanta, GA), pages 238–47. USENIX, Summer 1986.

[Leffler89] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *Design and implementation of the 4.3BSD UNIX operating system.* Addison-Wesley, 1989.

[Lunt90] S. Lunt. Experiences with Kerberos. *Second Security Workshop Program* (Portland, OR), pages 113–20. USENIX, 27 August 1990.

[Macklem91] R. Macklem. Lessons Learned Tuning the 4.3BSD Reno Implementation of the NFS Protocol. *USENIX Conference Proceedings* (Dallas, TX), pages 53–64. USENIX, 21-25 January 1991.

[Mockapetris87a] P. Mockapetris. Domain names – concepts and facilities. Request for comments 1034. ARPA Network Working Group, November 1987.

[Mockapetris87b] P. Mockapetris. Domain Names – Implementation and Specification. Requests for Comments 1035. Network Working Group, November 1987.

[Myles93] A. Myles and D. Skellern. Comparison of Mobile Host Protocols for IP. *Journal of Internetworking Research and Experience*, **4**(4):175–94, December 1993.

[Myles94] A. Myles, D. B. Johnson, and C. Perkins. A Mobile Host Protocol Supporting Route Optimization and Authentication. *IEEE JSAC*. IEEE, 1994.

[Nachbar88] D. Nachbar. SPIFF – A Program for Making Controlled Approximate Comparisons of Files. *USENIX Conference Proceedings* (San Francisco), pages 73–84. USENIX, Summer 1988.

[Noor94] A. Noor. Network Information Service+. *UNIX Review*, pages 47–54, October 1994.

[Ousterhout85] J. K. Ousterhout, H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. *Proceedings of 10th ACM Symposium on Operating Systems Principles* (Orcas Island, Washington). Published as *Operating Systems Review*, **19**(5):15–24, December 1985.

[Pawlowski94] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS Version 3 Design and Implementation. *USENIX Conference Proceedings* (Boston, Massachusetts), pages 137–52. USENIX, 6-10 June 1994.

[Pendry91] J.-S. Pendry and N. Williams. Amd – The 4.4 BSD Automounter. User Manual, edition 5.3 alpha. Imperial College of Science, Technology, and Medicine, March 1991.

[Perkins93] C. E. Perkins. Providing Continuous Network Access to Mobile Hosts Using TCP/IP. *Computer Networks and ISDN Systems 26*, pages 357–69, 1993.

[Pike91] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9, a distributed system. *Proceedings of Spring EurOpen Conference*, pages 43–50, May 1991.

[Presotto92] D. Presotto. Plan 9, A Distributed System. *USENIX Workshop on Micro-Kernels and Other Kernel Architectures* (Seattle, WA), pages 31–8. USENIX, 27-28 April 1992.

[rfc0768] J. Postel. User Datagram Protocol. Technical report 768. Internet Engineering Task Force, August 1980.

[Rivest91] R. L. Rivest. The MD4 message digest algorithm. *Advances in Cryptology — Crypto '90* (New York), pages 303–11. Springer-Verlag, 1991.

[Rivest92] R. L. Rivest. The MD5 Message-Digest Algorithm. RFC 1321. Internet Activities Board, April 1992.

[Rosen86] M. B. Rosen, M. J. Wilde, and B. Fraser-Campbell. NFS Portability. *USENIX Conference Proceedings* (Atlanta, GA), pages 299–305. USENIX, Summer 1986.

[Rosenthal90] D. S. H. Rosenthal. Evolving the Vnode Interface. *USENIX Conference Proceedings* (Anaheim, CA), pages 107–18. USENIX, Summer 1990.

[Ruemmler93] C. Ruemmler and J. Wilkes. UNIX Disk Access Patterns. *USENIX Technical Conference Proceedings* (San Diego, CA), pages 405–20. USENIX, Winter 1993.

[Sandberg85a] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. *USENIX Association Summer Conference Proceedings of 1985* (11-14 June 1985, Portland, OR), pages 119–30. USENIX Association, El Cerrito, CA, 1985.

[Sandberg85b] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. *USENIX Conference Proceedings* (Portland, OR), pages 119–30. USENIX, Summer 1985.

[Satyanarayanan90] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: a highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, **39**(4):447–59, April 1990.

[Schaps93] G. L. Schaps and P. Bishop. A Practical Approach to NFS Response Time Monitoring. *Systems Administration (LISA VII) Conference* (Monterey, CA), pages 165–9. USENIX, 1-5 November 1993.

[Shafer92] S. Shafer and M. R. Thompson. The SUP Software Upgrade Protocol. Unpublished notes. CMU, 1992. Available by ftp from `mach.cs.cmu.edu` in `/mach3/doc/unpublished/sup/sup.doc`.

[SMCC90a] SMCC. flock(2). In *SunOS 4.1 Reference Manual, Section 2*. Sun Microsystems, Incorporated, 21 January 1990.

[SMCC90b] SMCC. lockf(3). In *SunOS 4.1 Reference Manual, Section 3*. Sun Microsystems, Incorporated, 21 January 1990.

[SMCC90c] SMCC. share(1M). In *SunOS 5.3 Reference Manual, Section 1M*. Sun Microsystems, Incorporated, 5 July 1990.

[SMCC90d] SMCC. exportfs(8). In *SunOS 4.1 Reference Manual, Section 8*. Sun Microsystems, Incorporated, 7 October 1990.

[SMCC90e] SMCC. syslogd(8). In *SunOS 4.1 Reference Manual, Section 8*. Sun Microsystems, Incorporated, January, 1990.

[SMCC92a] SMCC. DDI and DDK. In *SunOS 5.3 Reference Manual, Section 9*. Sun Microsystems, Incorporated, April, 1992.

[SMCC92b] SMCC. cfsadmin(1M). In *SunOS 5.3 Reference Manual, Section 1M*. Sun Microsystems, Incorporated, August, 1992.

[SMCC93] SMCC. Overview of NIS+. In *SunOS 5.3 Reference Manuals: Administering NIS+ and DNS*, pages 18–25. Sun Microsystems, Incorporated, 1993.

[SMCC94] SMCC. NFS Version 3 Protocol Specification. Technical report. Sun Microsystems, Incorporated, 16 February 1994. Available via anonymous ftp from `ftp.uu.net` in `/networking/ip/nfs/NFS3.spec.ps.Z`.

[Spector89] A. Z. Spector and M. L. Kazar. Uniting File Systems. *UNIX Review*, **7**(3):61–71. Miller Freeman Publications Company, March 1989.

[Srinivasan89] V. Srinivasan and J. C. Mogul. Spritely NFS: implementation and performance of cache-consistency protocols. Research Report 89/5. Digital Equipment Corporation Western Research Laboratory, Palo Alto, CA, May 1989.

[Stallman94] R. M. Stallman and R. H. Pesch. The GNU Source-Level Debugger. In *User Manual, Edition 4.12, for GDB version 4.13*. Free Software Foundation, January 1994.

[Stein87] M. Stein. The SUN network file system. *Digest of Papers, COMPCON Spring '87: 32nd IEEE Computer Society International Conference* (23-27 February 1987, San Francisco), pages 6+. IEEE Comput. Society Press, Washington, DC, Cat. No. 87CH2409-1, 1987. Summary only.

[Steiner88] J. G. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An Authentication Service for Open Network Systems. *USENIX Conference Proceedings* (Dallas, TX), pages 191–202. USENIX, Winter 1988.

[Stern92] H. L. Stern and B. L. Wong. NFS Performance And Network Loading. *Systems Administration (LISA VI) Conference* (Long Beach, CA), pages 33–8. USENIX, October 19-23 1992.

[Stevens94] W. R. Stevens. Maximum UDP Datagram Size. In *TCP/IP Illustrated, Volume 1: The Protocols*, pages 159–60. Addison-Wesley, February 1994.

[Stewart93] J. N. Stewart. AMD – The Berkeley Automounter, Part 1. *;login:*, **18**(3):19. USENIX, May/June 1993.

[Sun85] Behind the Network File System. *Computer Design*, **24**(6):152, June 1985.

[Sun86] Sun Microsystems Incorporated. *Network File System: protocol specification*, Part number 800–1324–03, revision B, 17 February 1986.

[Sun89] Sun Microsystems, Incorporated. *NFS: Network File System protocol specification*, Technical report RFC–1094, March 1989.

[Tait91a] C. Tait and D. Duchamp. Service Interface and Replica Consistency Algorithm for Mobile File System Clients. *1st International Conference on Parallel and Distributed Information Systems*, 1991.

[Tait91b] C. Tait and D. Duchamp. Detection and Exploitation of File Working Sets. *Proceedings of the Eleventh International Conference on Distributed Computing Systems*, pages 2–9. IEEE, May 1991.

[Tait92] C. D. Tait and D. Duchamp. An efficient variable-consistency replicated file service. *Proceedings of the Usenix File Systems Workshop*, pages 111–26, May 1992.

[Tanenbaum87] A. S. Tanenbaum. *Operating systems: design and implementation*. Prentice-Hall, Englewood Cliffs, NJ, 1987.

[Teraoka90] F. Teraoka, Y. Yokote, and M. Tokoro. Virtual Network: Towards Location Transparent Communiction in Large Distributed Systems. Technical report. Sony Computer Science Laboratory Incorporated, Tokyo, Japan, 1990.

[Teraoka91] F. Teraoka, Y. Yokote, and M. Tokoro. A Network Architecture Providing Host Migration Transparency. Technical report. Sony Computer Science Laboratory Incorporated, Tokyo, Japan, 1991.

[Teraoka92] F. Teraoka. Design Implementation and Evaluation of Virtual Internet Protocol. *Proceedings of the 12^{th} International Conference on Distributed Computing Systems*, pages 170–7, June 1992.

[Teraoka93] F. Teraoka and M. Tokoro. Host Migration Transparency in IP Networks: The VIP Approach. Technical report. Sony Computer Science Laboratory Incorporated, Tokyo, Japan, 1993.

[Wada93] H. Wada, T. Yozawa, T. Ohnishi, and Y. Tanaka. Mobile Computing Environment Based on Internet Packet Forwarding. *USENIX Technical Conference Proceedings* (San Diego, CA), pages 503–17. USENIX, Winter 1993.

[Walsh85] D. Walsh, B. Lyon, G. Sager, J. M. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, and P. Weiss. Overview of the Sun Network File System. *Proceedings of Usenix Association Winter Conference* (Dallas, Texas), pages 117–24. USENIX Association, January 19845.

[Watson92] A. Watson and B. Nelson. LADDIS: A Multi-Vendor and Vendor-Neutral SPEC NFS Benchmark. *Systems Administration (LISA VI) Conference* (Long Beach, CA), pages 17–32. USENIX, October 19-23 1992.