NFS Extensions for Transparent Access to Remote Devices

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology

by
Avinash Vyas

to the
Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

May, 2001

Certificate

This is to certify that the work contained in the thesis entitled "*NFS Extensions for Transparent Access to Remote Devices*", by *Avinash Vyas*, has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

May, 2001

(Dr. Deepak Gupta)	(Dr. Rajat Moona)
Department of Computer Science & Engineering,	Department of Computer Science & Engineering,
Indian Institute of Technology,	Indian Institute of Technology,
Kanpur.	Kanpur.

Abstract

Sharing resources increases their utilization and availability in a computing environment. Solutions exist for transparent sharing of resources, such as file sharing through SUN Network File System (NFS), printer sharing through Line Printer Daemon Protocol (lpd). These sharing services are in the form of resource specific protocols. A generalized framework is needed that can provide transparent sharing of all classes of devices. Such a framework is also desirable in Distributed Systems so that all the resources of a workstation are transparently accessible from any other. A resource sharing framework is easy to develop if there exist services that provide transparent access to remote devices.

In this thesis, we propose extensions to of Network File System (NFS) to transparently access remote devices in addition to files. Using this extended NFS protocol, clients will be able to access the local devices of the server. The proposed extensions conform to the design goals and properties of the NFS. They are not specific to any device and can be used to provide remote access to most devices in an easy manner. We have also proposed and implemented an architecture exemplifying the use of extended NFS protocol in providing transparent access to remote devices. Our implementation allows the clients to access remote consoles of the server. We have also described the strengths and limitations of our protocol.

Acknowledgments

I take this opportunity to express my sincere gratitude toward my supervisors Dr. Deepak Gupta and Dr. Rajat Moona for their invaluable guidance. I would never have thought of taking such a project to completion without their encouragement and support. Their discipline and innovative ideas gave me the right direction whenever I needed. I consider myself extremely fortunate to have had a chance to work under their supervision.

I also wish to thank whole heartily all the faculty members of the Department of Computer Science and Engineering for the invaluable knowledge they have imparted to me. I would like to specially thank Head of the Department Dr. Pankaj Jalote for his constant efforts of improving the postgraduate program. I also extend my thanks to the technical staff of the department for maintaining an excellent working facility.

My stay at IITK was unforgettable to say the least, and the biggest reason for it being my classmates of the great mtech99 batch. Specifically I would like to thank my brother Ashutosh, and friend Ashish Gupta for providing me a constant helping held. I cannot forget the times I spent with my wing mates Raju, Maloo, Mayank and Saugata. I thank my juniors Gaurav, Alok, Mukul and Neeraj for being so supportive. I would like to give special thanks to my seniors Atul, Rahul, Venkat, Godha and Rajeev for listening to my stupid problems patiently. Atul boss is the person from whom I have learned the most.

I can never thank enough my parents and my little sister Anshu for being a constant source of love and affection throughout. I am eternally grateful to them for always being with me whenever I needed them. Finally, I thank God for being kind to me and driving me through this journey.

Contents

Acknowledgments **i**

1 Introduction **1**

1.1 Resource Sharing . . . 1

1.1.1 Advantages . . . 1

1.1.2 Remote Access . . . 2

1.1.3 File Sharing . . . 2

1.1.4 Device Sharing . . . 3

1.2 Resource Sharing in Distributed Systems . . . 3

1.2.1 NFS . . . 4

1.2.2 AFS . . . 4

1.2.3 LOCUS . . . 4

1.2.4 RFS . . . 5

1.2.5 SPRITE . . . 6

1.2.6 JINI . . . 6

1.2.7 Windows NT . . . 7

1.3 The Scope of Our Work . . . 8

1.4 Our Approach . . . 9

1.5 Organization of the Report . . . 9

2 NFS protocol and implementation **10**

2.1 Characteristics of NFS . . . 10

2.2 Introduction . . . 11

2.3 The Protocol Suite . . . 12

2.3.1 Network File System Protocol 12
2.3.2 Remote Procedure Call Protocol 14
2.3.3 Extended Data Representation Protocol 15
2.3.4 Mount Protocol 16
2.4 Implementation and Control Flow 16
2.4.1 Overview 17
2.4.2 File Handles 17
2.5 UNIX Semantics and Performance 18
2.5.1 Deletion Of Open Files 18
2.5.2 Exclusive Use 19
2.5.3 Client-Side Caching 19
2.5.4 Retransmission Cache 20
2.6 NFS Version 3 20

3 NFS Extensions **22**

3.1 Design Goals 22
3.2 Design Issues 23
3.2.1 Mounting 23
3.2.2 Major Device Number 24
3.2.3 Device State 25
3.2.4 Consistency in the Device State 26
3.2.5 Ioctl 26
3.2.6 Blocking Input-Output 27
3.2.7 Data Buffering 28
3.2.8 Response time vs Network load 29
3.2.9 Asynchronous Notification 29
3.3 New NFS Procedures 30
3.3.1 Readdevice 30
3.3.2 Writedevice 30
3.3.3 Ioctl 31
3.4 Limitations 31
3.4.1 Disk-less Workstations 31

3.4.2 Exclusive Use of Devices 32
3.4.3 Asynchronous Notification 32
3.4.4 Crash Recovery 32

4 Implementation Architecture 34

4.1 Overview 34
4.2 Client Side Implementation 37
4.2.1 Device Independent Modules 37
4.2.2 Device Specific Modules 40
4.3 Server Side Implementation 41
4.3.1 Service Model 41
4.3.2 Device Independent Modules 43
4.3.3 Device Specific Modules 45
4.4 Terminal Specific Implementation 47
4.4.1 D-state 47
4.4.2 Server part 49
4.4.3 Client Part 50

5 Performance Evaluation 58

5.1 Experimental Setup 58
5.2 Functional Evaluation 58
5.3 Performance Evaluation 59
5.4 Results 61

6 Conclusions 63

6.1 Future Work 64

Bibliography 67

A New NFS Procedures 68

A.1 Readdevice 68
A.2 Writedevice 70
A.3 Ioctl 72

List of Tables

5.1 Overhead with Algorithm 1 61
5.2 Overhead with Algorithm 2 61

List of Figures

2.1 NFS Implementation 17

4.1 Overall Architecture 35
4.2 Terminal Driver Structure 51

5.1 Timeout values used in Algorithm 1 60
5.2 Timeout values used in Algorithm 2 60

Chapter 1

Introduction

A computer can be viewed as a collection of different resources such as hardware devices (e.g. CPU and memory), peripheral devices (e.g. keyboard and disks) and software abstractions such as files. The operating system provides *process* as the basic computing abstraction that uses the resources such as CPU time, memory and other peripheral devices to perform the required computation. The operating system acts as a resource allocator for these processes.

1.1 Resource Sharing

The concept of resource sharing was introduced by *Time sharing systems.* The processes on a single system share the CPU, memory and other devices to execute concurrently. This increased the utilization and the efficiency of the systems. The concept of resource sharing was extended across the computers by *distributed operating systems.* If a number of different computers are connected by a communication network, then a user or specifically a process at one computer is able to use the resources available at another.

1.1.1 Advantages

Devices like printers, plotters and scanners etc. are too expensive to be provided individually for every user. Shared use of these devices keeps the installation cost down and increases their availability and utilization. Additionally it allows for easy administration and better control of these devices. In distributed systems, sharing of the CPU among connected computers results in substantial computational speedup. Similarly sharing of files in these systems enable users to access them from different machines; this facilitates collaborative working. Additionally, file sharing allows user mobility as user can work on any machine and access his files transparently.

1.1.2 Remote Access

A mechanism for remote access to resources is the basic facility required for providing resource sharing among different computers. Such a remote access mechanism has to deal with network failure, crash of a client or a server, heterogeneity of the client and the server machines, while providing complete access transparency to the users. Access to a remote resource requires data to be transferred between local and remote machines. The most common way to achieve this transfer is through a *remote service* mechanism. In remote service, a request for access to remote resource is delivered to the remote machine, which performs the actual operation and returns the results to the requesting machine. The *remote procedure call* paradigm [10] is a representative of this remote service.

1.1.3 File Sharing

The classical UNIX operating system allows multiple users on a single machine to share files stored locally on the machine. The network connectivity enabled the file sharing between different computers. The early efforts in this direction were restricted to copying files from one machine to another such as UNIX-to-UNIX Copy program [12] and File Transfer Protocol (ftp) [14]. These solutions were far from fulfilling the vision of being able to access files on remote machines as local files. *Distributed File Systems* extend the sharing of files among users on different machines interconnected by a communication network. They hide the dispersion of file systems to provide a uniform view and transparent access to files across all the machines. A distributed file system can be implemented either as a part of a distributed operating systems or, in the form of a software layer which manages the communication between conventional operating systems and file systems to provide access to remote file systems. Remote File system (RFS) [15], Sprite file system [22] are examples of the former type of implementation and Network File System (NFS) [16], Andrew File System (AFS) [17] are the examples of latter type of implementation.

1.1.4 Device Sharing

The need to share devices in a computer system arises due to economics or the nature of applications. Sharing of devices is not restricted to just remote access, it has to deal additionally with issues such as access control, maintenance of job queue, priority of jobs and exclusive use of the device for a limited time. To provide transparent sharing of expensive devices, device specific application layer protocols have been developed. Examples are *Line printer daemon protocol* (lpd) [9] which provides transparent remote printing service and Remote Magtape protocol (rmt) [4] which is used for manipulating magnetic tape drives from a remote machine. In distributed systems, sharing of CPU is is implemented by *process migration*, primarily for *load sharing* among different machines. Transparent process migration requires remote access to files and devices. For example, migration of an interactive process requires remote access to the local terminal of the system, where it was executing before migration.

1.2 Resource Sharing in Distributed Systems

Several systems have been implemented which provides resource sharing. The resources that can be shared in these systems vary from files to devices such as terminals. In this section we will look at some of these systems and examine their capabilities of sharing different resources. We mainly emphasize on transparency and support for remote device access along with local device access. First we describe the systems which provides resource sharing on conventional operating systems, and then those in distributed operating systems.

1.2.1 NFS

Network File System (NFS) [16] is the *de-facto* standard for remote file sharing on UNIX systems. It is targeted for small environment such as LANs with limited number of clients. It allows sharing of a complete file system of file server or its subtree, among the client machines. There is no notion of a globally shared file system in NFS. Each client is independent to configure its own file system name space, so it is not necessary that all machines promise a common view of the shared file system. Chapter 2 discusses important characteristics and architecture of NFS in more detail.

1.2.2 AFS

Andrew File System (AFS) [17] is a distributed file system developed at CMU's Project Andrew and currently owned and supported by Transarc Corporation. In contrast to the NFS, AFS is capable of scaling to thousands of users. Clients are presented with a partitioned space of file names: a *local name space* and a *shared name space*. The uniform shared name space is provided by the servers, while the files constituting local name space are stored on the local disks of the clients. AFS uses file caching for better performance. Consistency is guaranteed by using *callback* mechanism to the client. AFS is only meant for shared access to remote files and does not provide access to the remote devices.

1.2.3 LOCUS

The LOCUS operating system is a distributed version of UNIX [13]. The heart of the LOCUS architecture is its distributed file system. The LOCUS file system presents a single tree-structured naming hierarchy that covers all objects in the file system on all machines. Every node of the distributed system is given a subtree in the file system hierarchy for its local file system. Local file system contains the device files for accessing the devices of that system. To provide transparent access to devices, standard file names were dynamically linked with site specific device files. LOCUS was the first of the distributed systems to support transparent access to remote named pipes and devices. Since it was developed before introduction of the VFS/Vnode architecture [7], remote device access was coded within each device driver instead of a separate implementation. For implementing an input-output of blocking nature, the request is blocked at the client side and the server uses *callback* mechanism for asynchronous notification from the device to the process.

1.2.4 RFS

AT&T introduced the *Remote File Sharing* (RFS) [15] file system in SVR3 UNIX to provide access to remote files over a network. Similar to NFS, RFS is based on a client-server model. The server exports directories and the clients mount them. The RFS provides transparent access to remote files, devices and named pipes. RFS also uses remote service for providing access to remote resources. RFS calls these mode of operation as *remote system call* model. For each system call that operates on the remote file or device, the client packages the argument to the system call, as well as information about the client process's environment into an RFS request. The server recreates the client's environment and executes the system call. The client process blocks until the server sends back a response message, containing the result of the system call. The client then interprets the results and completes the system call before returning control to the process. The server process executing the system call on behalf of the client may block for a long time, waiting for device or pipe input-output. Thus the number of such server processes becomes the bottleneck of this scheme. In case of too many requests blocked at the server, some incoming request may be denied service due to unavailability of the resources at the server. This results in loss of transparency. The implementation of RFS requires the state to be maintained both at the client as well on the server side. For this reason, the RFS has a complex mechanism for the crash recovery and a strong cache consistency protocol.

1.2.5 SPRITE

Sprite [22] extends the ideas of LOCUS for accessing the remote devices and preserving the UNIX semantics. It provides the notion of a file server, being different from the device server. In sprite a file server is a machine where files corresponding to devices are present, while device server is a machine where the actual devices are present. Each device file stores the information about the device server of the device corresponding to that file. This ensures a system wide uniqueness for devices. For accessing the local devices every system uses the same files, called the *localhost* device files. These device files map to the client's instances of the devices. For implementing blocking input-output, servers uses callback mechanism to the client, while the client blocks for the request. This scheme requires complex crash recovery protocol by virtue of it being a state based implementation.

1.2.6 JINI

A JINI system [18] is a distributed system based on the idea of federating users and the resources required by those users. The goal of the JINI system is to make the network a dynamic entity which enables its users to share services and resources over a network. It is intended to provide easy access to resources anywhere on the network while allowing the network location of the user to change. The main emphasis of JINI is on the dynamic joining and leaving of services, devices and users in the system rather than transparency of their use. The JINI system extends the JAVA application environment from a single virtual machine to a network of machines. JINI names all its resources as *services*. A service is an entity that can be used by a person, or by a program or by another service. The JINI system allows sharing of these services. The backbone of the JINI framework is the JAVA Remote Method Invocation (RMI) mechanism. RMI is a JAVA programming language enabled extension similar to the remote procedure call mechanism. RMI allows not only data to be passed from object to object around the network but also full objects, including code. It is tightly coupled with the JAVA programming environment and assumes that its components are implemented in JAVA. It is good for developing new distributed applications and sharing devices/services. Existing applications are not benefited with the framework because its main goal is platform independence and not transparency. Since JINI services need to be implemented over user level Java Virtual Machine, they are considerably slow.

1.2.7 Windows NT

Windows NT [1] is an operating system designed primarily for personal computers and its design is different from operating systems of UNIX domain. Its design incorporates several different models of operation, for providing various services. It uses *client-server* and *micro-kernel* model for providing multiple operating system environment. Most of the operating environment and operating system services are implemented as user level processes. The clients use these services by passing messages to them using the message passing primitives provided by the micro-kernel. It uses an object model for uniformly managing all system resources. Thus in Windows NT, resources such as files, processes, ports and semaphores etc. that can be shared, named, or made visible to user mode programs, are implemented as objects.

The kernel mode portion of the Windows NT is known as *NT Executive*. It consists of a series of components that implement basic operating system services like virtual memory management, file system and interprocess communication etc. The input-output system of Windows NT is one of the component of the NT executive. Windows NT has a packet driven input-output system, in which every input-output request is represented by an *I/O request packet* (IRP), as they move from one I/O system component to another. One of the I/O system component, called *I/O Manager* defines an orderly framework within which these IRPs are delivered to file system and devices. The I/O manager passes the IRP denoting an input-output request to the correct driver. The driver performs the operation specified by the IRP and returns the IRP containing the result back to the I/O manager. I/O system has uniform structure of all its drivers, with every input-output being implemented through these drivers. The file system being a part of I/O system, is also implemented through a *driver* which has structure and interface identical to other drivers.

The Windows NT has support for access to all remote resources. It views networking as a means to provide access to remote resources such as files, devices and ultimately processors. The networking software is also largely implemented as extension to the input-output system. Windows NT's support to access remote resources is built through two major components, *network redirector* and *server*. Both of these are implemented as file system drivers and therefore are the part of the input-output system of the NT executive. The redirector is the network component responsible for sending input-output request across a network when the file or device to be accessed is not local. The server on the remote machine (where the file or device is physically located) receives and serves such requests. The redirector and the server communicates using the SMB protocol. The object model provides the network transparency in case of remote resource access, but the naming scheme itself differentiates local and remote resources.

1.3 The Scope of Our Work

Our work is aimed at providing transparent access to remote devices, in a heterogeneous UNIX environment. The machines may have different architecture and may run different flavors of UNIX. The uses of such a system span a number of different audiences. It can easily be extended to provide sharing of expensive resources such as printers, plotters and scanners. Another area where such a system can be used is distributed operating systems. In these systems, process migration requires remote access to files and devices for greater transparency.

Our basic approach is to extend an existing remote file accessing mechanism to provide access to remote devices transparently. SUN NFS is one file system which is widely used for remote file sharing in distributed systems and which is designed for heterogenous environment. In this work, we extend NFS to allow transparent access to remote devices.

1.4 Our Approach

In our approach of providing access to remote devices using NFS, we have avoided changes to the existing NFS procedures. Since the existing read and write procedure cannot handle the complexity of the device input-output, we have added three new procedures in the NFS protocol. Two of these protocols are used by the clients for reading and writing to devices. The third procedure corresponds to the ioctl system call, which is used for modifying device properties. For implementing blocking input-output, the requesting process is blocked at the client. The client keeps retransmitting the request to the server till the desired operation completes successfully.

In order to make server stateless and facilitate transparent crash recovery, the state of the device is also maintained at the server. An NFS request should contain complete information required for processing it at the server. Hence the device state is also included in every request on remote device. We have also changed the service model of the these requests at the server.

1.5 Organization of the Report

The rest of this report is organized as follows. **Chapter 2** describes the SUN Network File System's design goals and its implementation in UNIX kernel. We also describe the protocols associated with the NFS like Mount, Network Lock Manager in this chapter. The design issues considered for the extension of NFS are discussed in **Chapter 3**. The implementation of Extended NFS in Linux kernel to support remote access to terminals is described in **Chapter 4**. In **Chapter 5**, we discuss some of the performance issues. Finally, in **Chapter 6** we conclude this thesis with a brief summary of the work and possible future enhancements.

Chapter 2

NFS protocol and implementation

The **Network File System (NFS)** is both an implementation and a specification of software system for accessing remote files [8]. It has become the de-facto standard for remote file sharing in UNIX systems. Till date SUN has released two versions of the NFS specification which are named as *version 2* and *version 3*. Version 3 removes some of the limitations of version 2 and improves its performance.

2.1 Characteristics of NFS

The NFS is designed for a network of heterogeneous machines. It is useful for sharing files among workstations in a small network. The design of the NFS has certain features as outlined here:

Operating System Independence: The NFS is designed to be independent of operating systems and machine architectures. Its implementation is not restricted to only UNIX systems and several other operating systems implement NFS server as well as NFS client.

Stateless Server: The unique characteristic of NFS is its statelessness. NFS does not require a server to retain any information about the state of the NFS clients between two of their successive requests. Each request is treated independent of all previous requests. In the NFS protocol the request sent by a client contains all the information needed to process it at the server end.

NFS, therefore does not provide any *open*, *seek*, or *close* requests as these operations necessarily require to maintain the state at the server side. The implementation of these functions is therefore private to the NFS clients.

Crash Recovery: The NFS protocol is very rugged against the crash of the server or that of the client. In case the client crashes, no crash recovery is required at the server as it keeps no persistent information about its clients. In case the server crashes, client keeps retransmitting the request at a certain frequency until the response is received from the server. When the server boots after the crash, it processes the request and sends the response back. It is possible because in the protocol all requests are independent of each other. Thus the NFS client can not distinguish between a slow server and a rebooted server.

Transparency: The NFS provides the fundamental property of *network transparency* as clients are able to access remote files using the same set of operations as applicable to the local files [8]. The name of the file does not reveal its physical location, hence it is *location transparent*. Change in storage location of a single file results in a change of its name, however transfer of the exported file system or directory has no effect on the name space at client side. The changes are made only in the client's mount table using the *mount* protocol. Thus the NFS also provides *location independence* at granularity of the component (i.e., exported file system or directory).

2.2 Introduction

NFS is based on a client-server paradigm. A file server exports a file system or subtree thereof. Clients are the machines that remotely access the files exported by the server. Clients can mount the entire file system exported by the server or a subtree of that onto any directory in existing file hierarchy.

Clients and servers communicate via remote procedure calls, which are synchronous calls. NFS uses the **Remote Procedure Call (RPC) Protocol** [20] to define the format of all interactions between the client and server. In fact, the NFS protocol [21] is defined as a set of remote procedure calls. SUN RPC and hence NFS uses Extended Data Representation (XDR) [19] to represent the data sent across the network in a standard machine independent format.

2.3 The Protocol Suite

The primary protocols in the NFS suite are RPC, NFS, and Mount. They all use XDR for data encoding.

2.3.1 Network File System Protocol

NFS defines a set of remote procedure call, their arguments and results which are used by the clients to operate on the remote files at the server. These are defined using the RPC language which is basically the XDR language extended with program, version, and procedure declarations. All procedures in the NFS protocol are assumed to be synchronous. The most important argument to these NFS procedures is the file handle, which is used by the clients to reference a file at the server. An outline of the NFS protocol version 2 procedures is given below.

null() returns ()

This procedure does nothing and is used to measure the round-trip time between the client and the server.

lookup(dirfh, name) returns (fh, attr)

This procedure returns the file handle corresponding to the file whose name is given as argument. The other argument is the file handle of the directory in which the file is present.

create(dirfh, name, attr) returns (newfh, attr)

This procedure creates a new file and returns the file handle and attributes of the created file. The other arguments are the file handle of the directory in which the file is to be created and the attributes of the file.

remove(dirfh, name) returns (status)

This procedure removes a file from the directory. The arguments to the procedure are the name of the file and file handle of the directory in which the file is present.

getattr(fh) returns (attr)

This procedure return the attributes of the file whose file handle is given as argument.

setattr(fh, attr) returns (attr)

This procedure set the attributes of a file to the given one. The file attributes which can be modified are mode, uid, gid, size, access time and modify time.

read(fh, offset, count) returns (attr, data)

This procedure is used to read data from a file whose file handle is the argument. The *offset* gives the starting byte, from where the data is read up to *count* characters from the file .

write(fh, offset, count, data) returns (attr)

This procedure is used to write data to a file. The *offset* gives the offset of the first byte within the file. The *count* number of bytes are written from *data* in the file. The *fh* provides the file handle for the file.

rename(dirfh, name, tofh, toname) returns (status)

This procedure renames a file *name* in the directory represented by its file handle *dirfh* to *toname* in the directory represented by its file handle *tofh.*

link(dirfh, name, tofh, toname) returns (status)

This procedure creates a hard link *toname* in the directory represented by its file handle *tofh* to the file *name* in the directory represented by its file handle *dirfh.*

symlink(dirfh, name, string) returns (status)

This procedure creates a symbolic link *name,* in the directory represented by *dirfh,* with value *string.* The *string* argument is not interpreted at the server.

readlink(fh) returns (string)

This procedure returns the string associated with the symbolic link represented by its file handle *fh*.

mkdir(dirfh, name, attr) returns (fh, newattr)

This procedure creates a new directory *name* in the directory represented by its file handle *dirfh*. It returns the file handle of the newly created directory and its attributes.

rmdir(dirfh, name) returns (status)

This procedure removes an empty directory *name* from the parent directory represented by its file handle *dirfh*.

readdir(dirfh, cookie, count) returns (entries)

This procedure returns up to *count* bytes of directory entries from the directory represented by its file handle *dirfh*. Each returned entry consists of file name, file id and pointer to next entry called the *cookie*. The returned *cookie* is used in the subsequent call to readdir, in case all directory entries were not read in the first request itself.

statfs(fh) returns (fsstats)

This procedure returns the file system information such as block size, number of free blocks, etc.

2.3.2 Remote Procedure Call Protocol

The idea of Remote Procedure Call (RPC) was mooted in mid 70's, but the first framework actually came in early 80's [23] [10]. Today there are many commercial RPC implementations available such as Sun Microsystems RPC [20], Xerox Courier RPC [2], Apollo's Network Computing Architecture[3] and OSF's Distributed Computing Environment RPC[6]. The basic working model of RPC is based on the traditional procedure call model, used in programming languages. Procedure call allows for transfer of control and data within a program running on a single computer. RPC extends the idea to transfer of control and data across a communication network.

RPC is based on a client-server paradigm. A client is a machine that requests for a procedure to be invoked and a server is where the procedure is actually executed. When a remote procedure is invoked, the calling process at client is blocked until it receives the response back from the server. The parameters and information about the procedure to be called are passed across the network to the server where the procedure is to be executed. When the procedure finishes results are passed back to the blocked process at the client.

NFS is built over the SUN RPC protocol. This protocol specifies message format, transmission methods and authentication mechanism, for remote procedure calls. SUN RPC is fundamentally independent of transport protocol. It implements its own reliable datagram service by keeping track of unanswered requests and retransmitting them periodically until a response is received.

2.3.3 Extended Data Representation Protocol

Computers in a heterogenous environment vary in architecture as well as operating systems. Each architecture has its own internal representation of data. These differences are in byte ordering, sizes of data types, and format of strings and arrays. Hence communication between machines with different architectures has to deal with these differences. In opaque data transmission, machines do not have to interpret data. The data is treated just as a byte stream. When data has to be interpreted by the receiver, both communicating machines have to agree upon a standard format. External Data Representation is one such machine-independent representation for data transmission. Data sent over the network is converted from the native to the XDR representation. Similarly, at the receiver, data is converted from the XDR to the native representation. XDR defines several basic data types and the rules for constructing more complex data types.

2.3.4 Mount Protocol

The Mount protocol allows NFS clients to mount the remote file systems exported by a NFS server. Using the mount protocol the client obtains the *file handle* of the root of the exported file system tree. Similar to the NFS, mount protocol is also described by a set of RPC procedures which use XDR for representing their arguments and results. It is a state based protocol which requires the server to maintain a list of all clients which have mounted a directory or file system exported by the server. This list is however not required for the usual operation of either the client or the server. The list is used only for advisory purposes like the server warning the clients before being shut down.

Version one of the mount protocol is used with the NFSv2. The only information communicated between these two protocols is the file handle of the root directory of the mounted file system.

2.4 Implementation and Control Flow

NFS has been ported to several non-UNIX systems such as MS-DOS and VMS. There are both user level as well as kernel level implementation of NFS for various operating systems. Our discussion restricts to the kernel implementation of NFS in conventional UNIX systems with VFS/Vnode[7] interface.

VFS/Vnode interface is based on object-oriented programming concepts and provides an architecture for accommodating multiple file system implementation in a single UNIX kernel. The VFS (virtual file system) abstraction represents a file system in the UNIX kernel and Vnode represents a file. They represent abstract base classes from which subclasses can be derived and implemented to provide support of different file systems. A typical Vnode interface in UNIX kernel consists of two parts. First part is the file system independent data and functions, which are used by other kernel subsystems to manipulate the file. Second part of the interface is the set of virtual functions which are implemented by specific file system and a private data structure that holds the file system specific data of the Vnode.

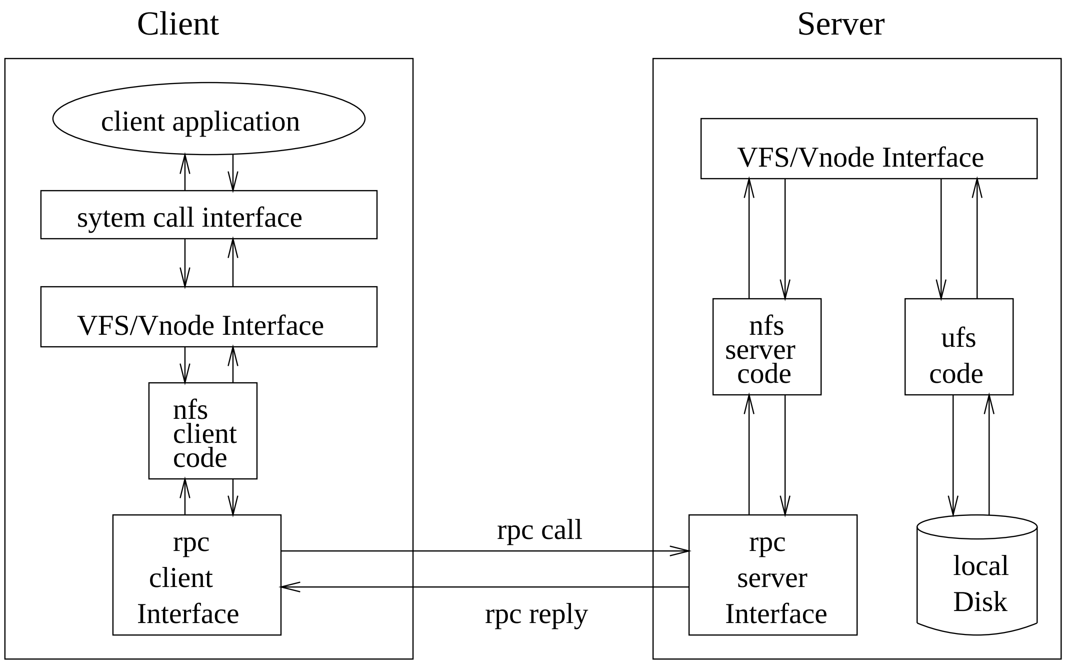

Figure 2.1: NFS Implementation

2.4.1 Overview

As shown in Figure 2.1, the server has exported a *ufs* file system, which is mounted by the client. When a process on the client opens a file mounted through NFS, after a name lookup, a file table entry and Vnode corresponding to that file is allocated at the client. The server as a result of a lookup on the file, returns a *file handle* corresponding to that file. This file handle, which is an opaque data object for the client, is sent by the client with every subsequent request to the server. The v_op field in the Vnode points to the vector of NFS client routines that implement the various Vnode operations. The server processes the requests by identifying the Vnode corresponding to the local file and invoking the appropriate Vnode operation that is implemented by the local file system.

2.4.2 File Handles

The NFS protocol associates an object called a *file handle* with all files in the exported directory. The server generates this handle when the client creates a remote file or issues a lookup on a remote file. The server returns the file handle to the client in the reply to the request, and the client can subsequently use it in other operations on that file. It is used by the server to identify the file that the client want to access. The file handle is an opaque 32 byte object for the client and the client should not make any attempt to interpret its contents, which are specific to the server. For most of the UNIX implementation, the file handle contains the *file system ID*, *inode number* and the *generation number* of the inode. The generation number is added to the inode to solve the problem of *stale* file handles. In absence of the generation number, if the server deletes a file and reuses its inode, while the file is being used by a client, the file handle at the client will refer to the newly created file. To eliminate this possibility, server needs to identify that, file handle sent by the client is stale. Since the generation number of inode is incremented each time the inode is freed, server can compare the generation number in the file handle and that in the inode of the file, to identify the stale file handles.

2.5 UNIX Semantics and Performance

NFS was primarily intended for UNIX clients, hence it is important that UNIX semantics be preserved for remote access to file access. The statelessness of NFS does not allow clients to maintain information about open files at the server, which leads to a few incompatibilities with UNIX.

2.5.1 Deletion Of Open Files

In UNIX, if a process deletes an open file (opened by another process or itself), the kernel does not actually delete this file. The kernel simply marks the file for deletion and removes its entry from the parent directory. Now no new process can open this file, while those that have it open can continue to access it. The kernel physically deletes the file, only when the last process that has the file open closes it.

In NFS this semantics cannot be implemented because the server does not know which files are open at the clients. In NFS a process will get unexpected error if another process deletes the file it is using. The problem can be solved partially at clients, as the clients are aware of the opened files. When the client detects an attempt to delete an open file, it changes the operation to rename the file, giving it a long and unusual new name which is unlikely to conflict with the existing files. This scheme solves the problem only when the two processes, the one using the file and the one deleting the file, are on the same client. It can not solve the problem when the two processes are on two different machines. Additionally if the client crashes after renaming the file and before actually deleting it, a garbage file is left on the server.

2.5.2 Exclusive Use

NFS cannot provide for record level or file level locking as provided by the UNIX for local files. As a result, a process can not access a file exclusively. Due to fixed size of a RPC request, a large *read* or *write* operation may span over several RPC requests. Hence, if two processes at two different clients issue write operation on the same file, at roughly the same time, overlapping writes at the server can occur. The Network Lock Manager (NLM) [11] protocol allows cooperating process to lock entire file or its portion, but it is only *advisory locking.* A process can always bypass the locks and access the file.

2.5.3 Client-Side Caching

If every operation on a remote files required one or more NFS request to the server, NFS performance would be intolerably low. Hence most NFS clients uses caching of both data blocks and file attributes to avoid sending NFS request to the server for every operation on the file. The file blocks are cached in the buffer cache and file attributes are cached in Vnode corresponding to the remote file. In order to avoid use of stale data at the clients, these cached contents must be refreshed on each change to the cached data or file attributes. In NFS, clients has to take measures for refreshing cached data. NFS clients maintains an expiry time indicating the time for which the attributes have been cached. If these attributes are accessed after a time quantum expires, clients fetch them from the server again. Before using the cached file blocks clients compare the *modify time* of the file attributes with the time when cached data was read from the server. If the file was modified after the data blocks were cached then cached data is flushed and request is sent to the server. All these mechanisms reduce, but do not eliminate the consistency problems of the cached data and file attributes.

2.5.4 Retransmission Cache

In order to provide reliable transmission, RPC clients retransmit requests until they receive a response. These retransmissions occurs due to the loss of a request or a response on the network or because the response is delayed due to the loaded server. The server needs to handle such duplicate requests correctly. NFS requests can be divided into two classes, *idempotent* and *nonidempotent*. Idempotent requests can be executed twice without any ill effect, while nonidempotent requests may result in incorrect behavior if repeated. Re-processing of duplicate requests not only results in incorrect behavior but also increases server load.

In order to detect and handle duplicate requests, the server keeps a cache of recent requests and reply messages. This cache is known as *retransmission cache*. Each entry in this cache additionally contains a *state* field and a *timestamp*. If the server finds an incoming request in the cache, and its state is *in progress*, the request is discarded. If the state of the cached request is *done* and the response has been sent just recently, the request is discarded. But if the duplicate request arrives at the server after some time, cached reply is retransmitted to the client. This approach requires a large cache, capable of saving the whole of the reply messages, which can be large for a request such as *read*.

2.6 NFS Version 3

NFS *version 2* (NFSv2) became enormously popular, which helped in highlighting its shortcomings. While some of the problems were addressed by clever implementations, many problems were inherent to the protocol itself. Thus NFS *version 3* (NFSv3) was introduced, addressing several important limitations of the older version. Major performance bottlenecks in NFSv2 are its synchronous procedures, which require the server to commit all modifications to stable storage before replying. NFSv3 introduces asynchronous writes in the protocol that allows the server to reply before committing the modification to stable storage. The data is finally written to the disk when the process exits or closes the file at the client and the client kernel sends a special request (COMMIT) to the server. This require that the client kernel holds on to data, until the process closes the file. NFSv3 supports greater file sizes by increasing the size of fields for specifying file size and offsets in read and write, to 64 bits. Additionally the number of over-the-wire packets for a given set of file operations are reduced by returning file attributes on every operations.

Chapter 3

NFS Extensions

In this chapter, we describe our proposed extensions to the NFS protocol. We first describe the design goals and the various design alternatives that could be used to meet these goals. Then we describe the procedures that we have added to the NFS protocol. Finally we describe some limitations of our proposed extensions.

3.1 Design Goals

The design of the proposed extensions to the NFS protocol is based upon the following goals.

Transparent Remote Access: The primary goal of our work is to support transparent access to the remote devices. The application processes must be able to use the remote devices as if they were local. Therefore the scheme should provide location transparency for devices, as NFS provides for the files.

Minimal Changes: Another important aim while developing the enhanced NFS protocol is to keep the changes and additions in the existing NFS protocol, its associated protocols, and device drivers to the minimum.

Device Independence: Enhancements made to the NFS protocol should be crafted with the aim of developing a generalized framework for transparent access to the remote devices. The protocol should not be specific to any device type and it should be possible to provide the remote access to new devices in an easy manner.

Conformation with NFS characteristics: Extensions and modification should be in conformation with the characteristics of the NFS. In particular, a strong emphasis should be laid on preserving the stateless nature of the protocol.

Heterogeneity: The work is aimed at providing transparent access to remote devices, in a heterogeneous UNIX environment. The machines may have different architecture and may run different flavors of UNIX. The enhancements to the NFS protocol should be independent of the operating system and machine architecture heterogeneity.

Preserving UNIX semantics: The framework for remote device access should try to preserve the UNIX semantics for device input-output. Although to preserve the stateless nature of the NFS protocol, the scheme may have to compromise on some issues. This could lead to a few incompatibilities with UNIX semantics.

3.2 Design Issues

In this section, we discuss the issues in the design of our NFS extension to support transparent access to remote devices.

3.2.1 Mounting

The physical disk unit typically consists of several logical partition each of which usually contains a file system. Mounting allows the users to view these different file systems as components of a single file system. Mounting integrates two file systems by making an association between the mount point directory and the root directory of the mounted file system. To access remote files through NFS, the clients are required to mount the remote file system. Similarly to access remote devices, the clients will be required to mount them. Ability to export an individual device increases the flexibility at the server, as each device can be exported selectively to the clients, using the same device file. Ability to mount an individual device increases transparency at clients. This is because, clients can mount individual remote device at the mount point that refers to the local instance of that remote device. The existing mount protocol is able to mount only file systems and directories. Mounting an individual device requires changing the mount semantics and thus its implementation.

In our design we wanted to keep the changes to existing system minimum. Therefore we decided against changing the mount protocol. Instead we use the existing mount protocol to mount a directory of the server containing device files. The device files are then used to access the remote devices of the server from where the directory is mounted.

3.2.2 Major Device Number

The kernel identifies each device by the device type (block or character), and a pair of numbers, called the *major and minor device numbers*. The major number identifies the device driver for a class of devices, while the minor number identifies a specific instance of a device in that class. Allocation of the major number to devices is specific to the operating system. In a heterogeneous environment a device may have major number at the server different then the major number at the clients. In order to provide operating system independence, we chose to standardize the major numbers for the devices in the extended NFS protocol. Further to minimize changes to the existing operating systems a mapping is maintained between native major numbers and protocol standardized major numbers. Before sending a request, the client converts native major numbers to standardized major numbers. Similarly upon receiving the response the client does the reverse conversion. Server upon receiving a request, maps the standardized major number to the native one. The server communicates responses with the client using the standardized major numbers. Thus all communication between the server and the client uses the standardized major numbers. No such mapping is required for minor numbers as their interpretation is specific to the server.

3.2.3 Device State

The devices have state and the operating system needs to know about these states for most kind of device operations. Thus the device operations are stateful in contrast to the file operations. The operating system maintains the device specific state for each device *(d-state)*. The d-state comprises of data structures to store configurable device parameters. It is encapsulated within the device driver of the device. In remote operations also the d-state has to be maintained. The d-state can be maintained either at the client side or at the server side or at both sides. Maintaining it only at the client side and using its device driver makes the server completely stateless. This option however requires duplication and incorporation of device driver code into the NFS server to interact with the actual devices. A lot of changes and additions are required in server code to provide remote access to each new device, limiting the flexibility and generality of the server. A protocol for ensuring consistent d-state in case of simultaneous access by multiple clients is also required. Such a protocol becomes very complex due to the stateless nature of NFS.

The choice of maintaining d-state at the server side facilitates the use of existing device drivers without any changes and provides a cleaner interface between the device and the server code. A crash recovery protocol would be required as the server crash would result in loss of d-state maintained at the server.

In our approach, the d-state is maintained at the client side as well as at the server side and the existing device driver at the server side is used. The d-state maintained at the client side is embedded in every NFS request on devices along with the other arguments. Thus, each request becomes self contained and the process of crash recovery is simplified. The notion of d-state kept at the server is different from the NFS notion of *state* and does not conflict with the stateless nature of the NFS server. In normal operation, NFS server does not require any information from the previously served requests in order to serve the current request. Moreover, the d-state maintained at the client side comprises of only the state which can be modified by the clients explicitly by *ioctl* system call or implicitly by other system calls. This state information depends on the type of remote device being accessed.

3.2.4 Consistency in the Device State

Maintaining the device state at two different locations has an inherent problem of inconsistency in the two states. The d-state kept at a client provides the state of the actual device to the client. In case of simultaneous access to the same device from two different clients, existence of two d-states can be a potential cause of inconsistency. This would mean different state of the same device at different clients. We solve this problem wherein the most recent image of the d-state from the server is sent to the clients along with each response. The clients use this state received along with the response to make their d-state consistent with that at the server. Although this scheme may not guarantee state consistency at all instants, it eventually makes the d-state across all the clients consistent with that of the server.

3.2.5 Ioctl

The *ioctl* system call is the generic entry point for modifying the user controlled d-state and to configure the devices. This is a highly versatile call through which one can support arbitrary operations on the devices. The arguments to this system call vary in number and type, depending upon the device and the type of request. For a device like terminal, a Linux implementation supports around 60 different ioctl commands. This generality makes *ioctl* very difficult to support in remote environment as it would require different arguments to be sent for each command. In order to handle the large number and complexity of different ioctl commands we classify them into following three categories:

1. Ioctl commands to retrieve some state information of the device.
2. Ioctl commands that modify the state of the device.
3. Ioctl commands that require some function to be invoked in the device driver of the actual device.

Ioctl commands which require only the retrieval of state information and passing it to the applications are handled at the client side itself. Such information is provided from the d-state present at the client side. Thus keeping the d-state at the client side in addition to the server side reduces the network traffic. A new procedure *ioctldevice* is added to the NFS protocol for the ioctl commands that modify the state of the device. Sending of individual ioctl commands and arguments to the server however increases the complexity of the NFS and XDR code. We used a different approach in our design. The execution of ioctl command modifies the d-state at the client side and then the entire d-state is sent to the server. At the server this received state is used to update and modify the actual device state. This allows the design of ioctl procedure to be uniform and akin to the other two procedures for reading and writing on device. For the ioctl commands which require some function to be invoked at the server the *ioctldevice* procedure has the provision for sending the commands along with their arguments, and receiving the response back from the server. For example, in terminal devices, the ioctl command *TIOCSTI* is used to put a character in the read buffer of the terminal driver. This ioctl command requires sending of the character to the server and then storing it in the read buffer maintained by the terminal driver. Since such ioctl commands are very small in number, it would not make the XDR encoding-decoding function unmanageable.

3.2.6 Blocking Input-Output

The service time associated with the device operation is potentially indefinite (for example, in the case of terminals or other user input devices) in comparison to the one associated with the file operations which is typically more predictable. This blocking nature makes input-output on devices difficult to implement in comparison to that on files. There could be two options for implementing the blocking input-output for remote access, either block at client side or block at the server. When a request blocks at the server, it holds up system resources such as memory. In extreme cases some request may be denied service due to non availability of resources. In case the server crashes before the completion of the blocking input-output, loss of all blocked requests requires a complex crash recovery protocol. NFS uses the implicit acknowledgement model for RPC requests, where response to a request indicates that the request was correctly received at the server. Thus in absence of response from the server, it is impossible for the clients to distinguish between a lost request and a blocked request at the server. When there is no response from the server within a time period, the client has to retransmit the request assuming it was lost. So a blocked request will cause unnecessary retransmission of requests, increasing network traffic and server load.

The option of blocking at clients, requires the server to store information about the client and the operation being performed. Later when the requested operation can proceed without blocking, the server can notify the client to retry its request. This *callback* scheme that is used in many other systems, is unsuitable for extended NFS because of the stateless property of the NFS protocol.

In our approach we implement all operations as non-blocking at the server, while blocking is implemented at client side. In this scheme, if a device operation cannot completed immediately, a special status value is returned by the device input-output handler at the server. The server returns this special status to the client who retries the operation with increasing time intervals until it is completed successfully.

3.2.7 Data Buffering

The device drivers buffers the input data, before it is read by a process and buffers the output data before it is written onto the physical device. The reason of data buffering in most of the device driver is to increase performance, as input-output operations on physical device (e.g. disks) are costly operations. In some device (e.g. terminals) apart from performance reasons, the input data has to buffered as it is generated asynchronously. There could be two options to buffer this data, either at the server side or at the client side.

If the data is buffered at the server side then every input-output request on a remote device at the client, will be carried out by involving network traffic and thus will increase the processing time of the request. On the other hand if the data is buffered at the client side then we need to ensure consistency of the data across all the clients using the same device. In case of NFS since the server does not store any information about the clients, consistency of the data buffered at the clients cannot be guaranteed. Different schemes can only reduce the problem but can not eliminate it. In case of interactive devices, data cannot be buffered at the clients as such devices generate data asynchronously. The server has to store this data until any client sends a request to read it.

The decision of data buffering is specific to the device which is accessed, but in the absence of a ideal buffer consistency scheme, we suggest the buffering to be done at server. This is also the case for the interactive devices.

3.2.8 Response time vs Network load

The trials of a request for an operation from the clients incur heavy network and server load. The time interval between the re trials determines the response time for an operation. If this time is high than the user experiences a delay in the response and hence transparency is lost. Keeping this interval low results in a heavier load on the network. For a tradeoff between the two, the time interval and the algorithm for re-trial needs to be fine tuned. We explain this algorithm for our implementation in Chapter 5.

3.2.9 Asynchronous Notification

Our scheme (and for that matter any model adhering to the design principles of NFS) cannot provide asynchronous notification from the device to a client process as it would involve the initiation of communication from the server to the client. This cannot be achieved without compromising the stateless nature of the server. Hence using only this framework we cannot support devices which require asynchronous notification to processes. An example of such a device is the controlling terminal, which requires that the terminal generated signals be delivered to the foreground process group. Although an external remote signaling mechanism can provide such support, it is out of the scope of this work.

3.3 New NFS Procedures

Our design necessitates sending of d-state along with input-output requests and responses. The *read* and *write* procedures of the existing NFS cannot handle the complexity of device specific input-output. Hence, we propose to add three new procedures *readdevice*, *writedevice* and *ioctl* in NFS, for reading from, writing to and changing the properties of the device. Apart from these additions no changes have been made to any procedure of the existing NFS protocol. We here describe the three new procedures added in the NFS protocol, their arguments and their results. The protocol enhancements have been made in version 2 of NFS and should be consistent with NFSv3 as well.

3.3.1 Readdevice

Readdevice is the procedure to read data from a device at an enhanced NFS server. The arguments to readdevice procedure consist of the file handle, offset, count and d-state of the device which is accessed. The file handle represents the file through which the device is accessed, offset provides the position on the device, from where to start reading (in case of some devices it is unused) and d-state is the device state maintained at the client. The response of the readdevice procedure contains the data read from the device and the recent d-state of the device at the server, along with the attributes of the device file after operation.

3.3.2 Writedevice

Writedevice is the procedure to write data onto a device at an enhanced NFS server. The arguments to writedevice consist of the file handle, offset, count, data and d-state of the device which is accessed. The file handle represents the file through which the device is accessed, offset provides the position on the device, from where to start writing (in case of some devices it is unused), and d-state is the device state maintained at the client. The response of the writedevice includes the recent d-state of the device at the server along with the file attributes.

3.3.3 Ioctl

Ioctl is the procedure to change the state of the device at an enhanced NFS server. The arguments to the ioctl procedure consist of the file handle, ioctl number, ioctl arguments and d-state of the device which is accessed. The file handle represents the file through which the device is accessed, ioctl number identifies the ioctl that needs to be executed at the server (this remains unused for most of the calls), arguments needed for the ioctl command (applicable only when ioctl number is used) and the d-state is the modified state of the device maintained at the client. The response of the ioctl includes the result of the ioctl command that is executed at the server (used only when the ioctl number in arguments was used), along with the attributes of the device file after operation.

3.4 Limitations

In this section we discuss the applicability and various limitations of the design of the extended NFS. Since the main emphasis is on preserving the design goals and properties of NFS and keeping changes to the minimum, various hard issues had to be resolved.

3.4.1 Disk-less Workstations

In case of disk-less workstations, NFS is often used for mounting the root file system. This file system also includes the /dev directory, the files of which are used to access the local devices on the disk-less workstation. With the scheme proposed, the device files in a directory mounted by the clients, are used to access the devices of the server. This scheme therefore is of limited use for the disk-less workstations. Several solutions can be used in such a case to enable disk-less workstation to access their local devices. At the boot time, a RAM disk can be used in which the device file can be created to access the local devices. Alternatively another file server can be used to mount a directory containing the local device files to provide access to local devices. Another possibility would be to extend the mount protocol to export and mount a device, rather than just a directory.

3.4.2 Exclusive Use of Devices

Simultaneous use of the same device by two or more processes may lead to problems. For example, if a printer is used simultaneously by two process such that their write requests are interleaved, printer's output will also be interleaved and of no use. We need a mechanism with which a process can exclusively use a device. This problem of exclusive use of a device is out of the scope of this framework. This is because to provide exclusive access to a device, the server needs to retain some state for the device which would violate stateless nature of the server. The support for exclusive use of devices can however be provided through a separate protocol like Network Lock Manager, which provides file locking in NFS.

3.4.3 Asynchronous Notification

In NFS, asynchronous notification of any sort from device to process is not possible due to the stateless nature of the servers. This scheme therefore does not provide any mechanism for asynchronous notification from device to process. For example this scheme can not be used for delivering device generated signals to the process. It would require some external mechanism for delivering device generated signals remotely.

3.4.4 Crash Recovery

The state recovery of a device, after a server crash is done when the first request comes from the client to the server after the crash. This leaves a race condition which possibly could result in a device state after the crash different from the one before the crash at the server. A possible scenario where this may happen is when two clients, say A and B, simultaneously open the same device. Suppose the server crashes just after client A changes the state of the device, so the client B's d-state becomes stale. If the first request after server-reboot comes from the client B, device state at the server will be restored to stale d-state. The client A who made the changes to device state prior to the crash, will also change its d-state to this stale state after its subsequent requests and responses. This will results in loss of the last modification made by the client A. Although this differs from the traditional uni-processor UNIX semantics a bit, it ensures consistency in the d-state of the device at all the clients.

Chapter 4

Implementation Architecture

In this chapter, we describe our implementation of the proposed NFS extensions. We have implemented both the client and the server parts of the design within the Linux 2.2.9 kernel. We first give an overview of our implementation architecture. We then describe the server side and the client side implementation respectively. Finally we describe the device specific implementation aspects for terminal devices.

4.1 Overview

Our architecture cleanly separates the device independent code from the device dependent code, at both the client and the server end. The overall implementation architecture is shown in Figure 4.1. The client side implementation consists of *the remote device driver*, *NFS client extensions* and *the kernel poll thread*. The NFS client extensions comprise of the functions implementing the NFS protocol procedures and XDR functions for encoding and decoding their arguments and results. These XDR functions use the device specific XDR functions to encode and decode the d-state of the device, associated with every request and response. The remote device driver translates the input-output operation on the remote device to NFS requests, which are sent to the server for execution. It also maintains the d-state of the device at the client side for the purpose of crash recovery. The kernel polling thread at the clients polls the remote devices on behalf of the requesting processes.

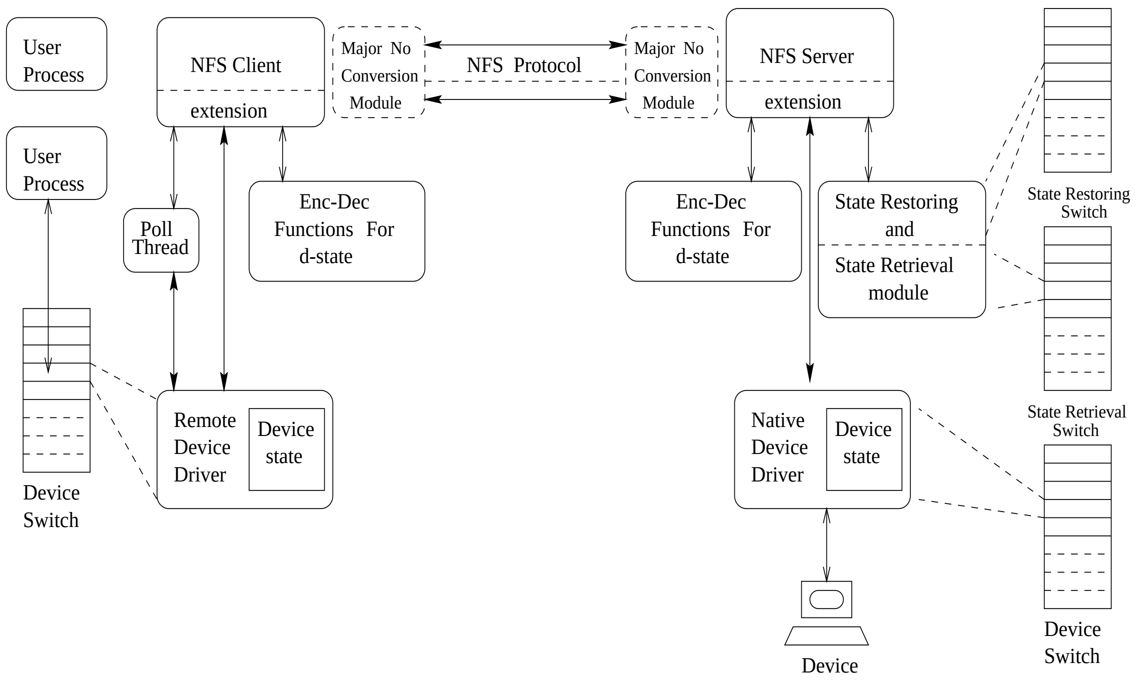

Figure 4.1: Overall Architecture

This scheme for polling is essential for implementing the select system call and is explained in detail in Section 4.2.1.

The server side implementation consists of *NFS server extensions* and *the device state restoration and retrieval module.* The NFS server extensions comprise of the implementation of NFS protocol procedures and their corresponding XDR function-s. These XDR functions use the device specific XDR functions for encoding and decoding the d-state of the device as done at the client side as well. The NFS server implementation uses the VFS/Vnode interface to interact with the actual device. In case of a server crash, the device state restoration and retrieval module at the server, is responsible for restoring the state of the device to the one prior to the crash. It also associates the latest d-state of the device with the results of every input-output request.

Together, these five components constitute the complete framework of transpar-ent remote device access. These modules are explained in detail in later sections. We now provide a brief overview of the functioning of the remote device access system.

To access the devices of a server, the clients mount the directory containing device files of a server through the mount protocol associated with NFS. The server does not expose local major numbers of the devices to the clients. The clients therefore see the major numbers of the devices as modified by the major number conversion module. At the client side all input-output operations on the remote device are handled by remote device driver for that device. When a process at the client opens a device file of a remote device, the NFS *lookup* request is sent to the server to obtain the file handle of the device file. After obtaining the file handle, the *open* function of the remote device driver at the client is invoked. The open function is used to construct the d-state of the device at the client. Subsequent operations on the remote device requested by the process are translated by its remote device driver into one or more NFS requests to the server. Along with the file handle and arguments, the d-state of the device is also embedded in such NFS requests. The inclusion of the d-state makes each NFS request self contained and independent of previous requests for the operation on the same device.

At the server the device requests are handled by the major number conversion module. This module translates the major number in the request to the local device major number. Further processing of the request depends upon the state (open or closed) of the device for which the request is made. A closed device indicates that either no request was made for the device earlier or the device was not accessed for a long period (and therefore was automatically closed by the NFS server) or the server was rebooted after a crash. In these situations, the *Device state restoration and retrieval module* opens the device and restores its state to the one embedded in the request. For an already opened device, the request does not require the reopening of the device.

The device requests are then translated through the VFS/Vnode interface into the corresponding device driver functions. After processing the request, the results and the latest d-state of the device are sent back to the client. On the client side, the device driver either returns control to the user process or retransmits the request after some time, depending upon the response.

4.2 Client Side Implementation

The extensions made to the NFS protocol at the client side have two parts. The device independent part and the device dependent part. The device independent part consists of the NFS procedures, their XDR functions and the kernel poll thread. The device dependent part comprises of the remote device driver and the XDR functions for encoding and decoding the d-state of the device.

4.2.1 Device Independent Modules

We now describe in brief the device independent parts of the implementation of NFS client extensions.

▮ *NFS Procedures*

There are three function corresponding to the newly added procedures of NFS protocol viz., *readdevice*, *writedevice*, and *ioctl*. The remote device driver (as explained later) uses this interface to send requests for the remote device to the server and receive responses from the server.

▮ *XDR functions*

There are two functions for each of the three newly added NFS procedures. One function is used for encoding the arguments of NFS request to XDR representation and second is used for decoding the response in XDR representation to native form. These functions use the device specific XDR functions to encode and decode the d-state of the device embedded in every request and response.

▮ *Kernel Poll Thread*

The *poll* system call allows a process to check whether a device or socket is ready for input-output, without actually requesting an input-output operation. This allows a process to read from a device only when data is present, otherwise continue its regular processing. Polling is particularly helpful when a process is monitoring many devices for input. However polling is wasteful is terms of processing power, as the process itself has to repeatedly check the device status. Its variant, the *select* system call allows a process to check data availability on multiple devices or sockets without repeatedly checking the device status. In case of select system call, the kernel notifies the process whenever the device status changes.

Our implementation of poll and select calls uses a kernel thread named *kernel poll thread* and a linked list called *nfs_poll_issued.* The poll system call for a remote device, places the information required for polling a remote device as an entry in the nfs_poll_issued list. The kernel poll thread uses the information stored in the nfs_poll_issued and sends a NFS poll-ioctl request (described later) for polling the remote device. The poll mask received in the response is stored in the nfs_poll_issued, which is later returned to the requesting process.

The entries in the nfs_poll_issued list contain the complete information required for sending NFS request to poll remote device. Some of the important fields of an entry in nfs_poll_issued list are described below.

nfs_server: This structure contains the information (e.g. hostname) of the NFS server hosting the remote device to be polled.

nfs_fh: This structure is the *file handle* of the device file of the remote device to be polled.

device: This integer stores the major and minor numbers of the device being polled. The major in this list is the native major number of the clients operating system for that remote device. It is converted to the NFS wide major number before sending the request to the server.

timeout: The time interval between two successive polling requests. The polling thread retries a poll request only after the expiration of this counter. The timeout value is increased each time the NFS poll request for the remote device is unsuccessful. This field is used by the implementation of the select system call.

wait_address: This field represents the *wait_queue* on which the processes sleep, while the kernel poll thread polls the remote device. This field is also used in the implementation of the select system call.

completed: This flag indicates whether the poll request on the remote device has completed successfully or not. A request is not successful because the device has no data to be returned.

nos_proc: This field indicates the number of processes polling on the remote device represented by this entry.

properties: This field points to the d-state of the polled remote device.

Poll System Call: As explained earlier the extended NFS does not include a procedure for polling a remote device. Request to poll a remote device are sent as NFS ioctl request to the NFS server for execution. We refer such an NFS request as *poll-ioctl* request. The poll system call is implemented using the kernel poll thread. The kernel poll thread uses the NFS poll-ioctl request to poll the remote device as explained above.

Select System Call: As mentioned earlier select is a blocking system call and requires a notification from the kernel to the process. The kernel poll thread is introduced to implement the select system call. In implementation of other blocking operations (e.g. read), the device driver retries the NFS procedure in the context of the requesting process. The same mode of operation for select makes it equivalent to poll. Hence in our approach the select system call is implemented using kernel poll thread.

When a process makes a select system call with a remote device as one of its arguments, it is handled by the poll function of remote device driver. This function places a structure containing file handle, server address, poll arguments, time when to issue next poll and other information (as described earlier) into the nfs_poll_issued linked list and sleeps on the wait-queue associated with this entry.

Upon the timeout, the kernel poll thread re-sends an NFS poll-ioctl request to the server, using the information from the entry. On success the completed flag is updated and the waiting process is woken up. In case of unsuccessful poll, the time interval for retransmission is set by the retransmission algorithm specific to that device, and the entry is placed in the linked list again. When the sleeping process wakes up, it checks the entry in the linked list. If entry is marked completed, then it removes the entry from the linked list and returns with the *poll mask* stored. If entry indicates unsuccessful poll and the application specified timeout for select has occurred, the device driver retries the NFS poll-ioctl request and returns with the poll mask returned by the server in response.

The retransmission algorithm used by the kernel poll thread for deciding the timeout for retransmissions depends on the kind of device being polled. The kernel poll thread uses *retransmission timeout switch* and the major number of a device for deciding the timeout period. The retransmission timeout switch is an array of pointers indexed by the major number of devices. Its each entry refers to a table describing successive retransmission timeout values for each kind of device.

4.2.2 Device Specific Modules

The device dependent modules of the NFS client extensions consists of device specific XDR functions and the remote device driver, which are explained in detail below.

▮ *Device Specific XDR functions*

The device specific XDR encoding-decoding routines are used for two purposes. First, they are used for conversion of device specific arguments (for example those of ioctl) and results of NFS procedures between the native and the XDR representations. Secondly, they are used to encode and decode the d-state of the device associated with each NFS request and response.

▮ *Remote Device Driver*

There is one remote device driver at client corresponding to each type of remote device. The remote device driver hides the physical location of the device and makes the access to remote device transparent. The main purpose of the remote device driver is to translate the input-output operations on remote device to one or more NFS requests to the server. Upon receiving the response the remote device driver decides to retry the NFS request after some time interval or return to the process, depending upon the input-output semantics of the device and nature of the request. It also implements blocking semantics of device input-output for the clients. The blocking semantics is implemented by retransmitting the request until it is executed successfully at the server.

The remote device driver also encapsulates the d-state of the device that is required for crash recovery of the server and is sent with every NFS request on the device. It maintains this d-state using the state received with each response from the server.

4.3 Server Side Implementation

In this section we describe the implementation of the server side NFS extensions. At the server, a new service model is introduced for the newly added procedures of extended NFS protocol. The new service model is a modified form of the service model of original NFS protocol.

4.3.1 Service Model

The NFS server is primarily designed to handle the requests for the remote file operations. A possible stateless algorithm for handling remote file requests is described as follows.

```
service_request(request) {
    open file
    execute the requested operation
    send response to the client
    close file
}
```

This service model is however not useful for implementing the device operations over NFS. This is primarily because the closure of the device file with each request on the server will destroy the device state at the server. We require the device state at the server even when no request is being served. For example, the terminal driver when put in *non echo* mode should not echo any keystrokes even if the application has not made any request to read the data. To implement such semantics we changed the service model for NFS requests on devices. A device is opened by the NFS server when the first NFS request for its access is received. At this time, the state of the device is set using the d-state included in the request. The device is not closed after an operation so that device state persists across two device operations. The modified service model is described as follows.

```
service_request(request) {
    if (the requested device is not open) {
        open the device
        set the state using the d-state in request
    }
    execute the requested device operation
    send response back to the client
}
```

This scheme however has a small drawback in which the device once opened will never be closed. Note however, that this does not introduce any inconsistency in the device behavior. For resource optimizations we implemented a scheme where in the device is closed by the server after absence of requests on it for a significantly large amount of time. Thus the memory used by various data structures of the device driver will be freed when a client process ceases to access the device.

4.3.2 Device Independent Modules

The device independent part of the server implementation of the extended NFS protocol comprises mainly of the server procedures, their XDR functions and routines that provide interface to the device driver functions.

▮ *Device Interface*

As discussed in the design, the NFS requests at the server are translated through VFS/Vnode interface into corresponding device driver functions. In order to keep the interface between NFS server and native device driver clean and minimize the required changes, we have provided four functions – *opendevice*, *readdevice*, *writedevice* and *ioctldevice*. These functions provide the interface to the existing device driver functions. They use a file table entry to store the pointer to device driver functions and its private data structures. The file table entry once allocated, is used by these procedures to serve subsequent requests on the device. We now describe these functions briefly.

Opendevice: As discussed earlier, the service model of the new procedures requires a device to remain open once a request for that device is received at the server. The processing of the request on a device depends upon whether the device is found in open or close state. This function is used for the device state dependent processing of a request. It is responsible for obtaining the file table entry corresponding to the device file on which the operation is to be issued. To provide the needed functionality it maintains a linked list called *open_devices*, which stores information about devices that are currently open. A typical entry in *open_devices* contains the major and minor numbers and the file table entry of the file corresponding to the device. Before any NFS request on device is served, this function searches the *open_devices* linked list for an entry corresponding to that device. If an entry for that device is found, the corresponding file table entry is returned. If the entry is not found in the *open_devices*, a new file table entry is allocated. The entry is initialized and stored in the *open_devices* list. In this case the function also invokes the device specific functions to restore the device state. The d-state in the request is used for restoring the device state. Finally a pointer to the file table entry is returned.

Readdevice: This function is the interface to Vnode function for reading from a device. It uses the implementation of the Vnode function provided by the existing device drivers. Using the *opendevice* function it obtains the file table entry corresponding to the file through which the device is accessed and invokes the Vnode *read* operation of the native device driver. The read operation of the device driver is issued in a non-blocking mode. If the operation completes without blocking, its result is encoded and sent back to the requesting client. Otherwise if the operation is required to block in the device driver, a special status is sent back to the requesting client. This ensures a bounded response time to the clients, even for blocking read operation. The result of the read operation and the d-state of device are returned to the NFS server procedure which are then sent back to the client.

Writedevice: This function, similar to the *readdevice* function, is the interface to Vnode function for writing to a device. It uses the implementation of the Vnode function provided by the existing device drivers. Using the *opendevice* function, it obtains the file table entry corresponding to the device file and invokes the Vnode *write* operation of the device. Similar to the read operation the write operation of the device driver is also issued in a non-blocking mode, thus ensuring a bounded response time to the clients. The result of the write operation and the latest d-state of device are returned to the NFS server procedure which are then sent back to the client.

Ioctldevice: This function modifies the d-state of the device at the server. Since the d-state and the ioctl commands sent with the request are specific to a device, the device specific functions are used to modify the d-state of the device. This function therefore uses the *device state restoration switch* for modifying the d-state of the device. The implementation and functioning of *device state restoration switch* is explained later. After modifying the device state, the d-state of the device and the results of ioctl commands are returned back to the NFS server procedure to be sent back to the client.

▮ *NFS Procedures*

There are three function corresponding to the newly added NFS procedures *readdevice*, *writedevice*, and *ioctl*. At the server, after getting the decoded arguments, these procedures invoke the corresponding device interface routines explained above. The results are converted into XDR representation before sending them to the clients.

▮ *XDR functions*

For each of the three new procedures, there are two XDR functions – one for decoding the arguments of the request and second for encoding the results. The request and the response include the d-state and some device specific arguments (in case of ioctl), which are specific to a device. Hence, these functions use the device specific XDR functions for encoding and decoding the d-state of the device and arguments of ioctl requests.

4.3.3 Device Specific Modules

The device specific modules at the server side include the device state restoration and retrieval module, and the XDR encoder-decoder functions for the d-state.

▮ *XDR Encoder-Decoder Functions for d-state*

These functions convert the d-state sent along with the request from XDR representation to native format and from native to XDR representation when d-state is sent along with the response. Some ioctl commands and their arguments can also be sent to the server to be executed. This function first decodes such ioctl commands and then convert their arguments from XDR representation to native one.

▮ *Device State Restoration and Retrieval Module*

This module provides a clean and well defined interface between the device independent and device dependent processing of an NFS request on a device at the server. It is responsible for restoring the d-state of the device to the one embedded in the request if the device is not found in open state. The same module is responsible for handling the device specific ioctl processing. The module also provides the latest d-state of the device which is sent along with the response to the client. There are three major data structures maintained by this module. These are used by the NFS server for handling the d-state of the device. The data structures and their use is described as follows.

Open Device List: This list called *open_ devices*, as discussed earlier, is used to identify the open instances of the devices at the server.

State Restoring Switch: The device state restoration part of this module maintains two *state restoring switches* similar to the *device switches*, one for character devices and second for block devices. The state restoring switch is an array of function pointers which is indexed by major number. Each entry points to a function that is used for modifying the d-state specific to the device with that major number. For example, the function for modifying the d-state of a device whose major number is i can be found at an index i of the array. This function uses the d-state embedded in the request and different ioctl commands provided by the native device driver to restore the d-state of the device.

As explained earlier, the NFS ioctl request includes the d-state of the device as its argument. This d-state is to be used to set the state of the device at the server. In our implementation the state restoring module also handles the NFS ioctl request. However for some ioctl commands the client sends the arguments along with the ioctl request, which are processed at the server. Thus the device specific state restoring function needs to handle these cases as well. This function uses equivalent ioctl commands provided by the native device driver at the server to perform the ioctl requested by the client.

State Retrieval Switch: The device state retrieval part of this module maintains two state retrieval *switches*, one for character devices and one for block devices. Similar to the state restoration switch, state retrieval switch is also an array of function pointers indexed by the major number. These functions are meant for retrieving the device specific d-state to be sent with the response. Each device has different d-state and hence has different methods to retrieve this state of the actual device. The suggested method of retrieving the d-state of a device, is using different ioctl commands (specific to the device) through the VFS/Vnode interface. The state retrieval function hides the device specific processing needed to retrieve the d-state of the device at the server.

4.4 Terminal Specific Implementation

In the extended NFS protocol, the device specific part of a device that can be accessed remotely is to be provided. We have implemented the support for the remote access to terminals using the extended NFS protocol. In this section we describe this implementation.

4.4.1 D-state

The most important thing in providing remote access to a device is to identify the d-state of that device. For terminals, the d-state comprises of three fields – *termios* structure, *winsize* structure and the *line discipline.* The contents of these structures and their use described is as follows.

Termios is the structure that contains all the characteristics of a terminal device that can be examined and changed. This is the most important part of the d-state of the terminals. It contains four sets of flags and an array of control characters. The declaration of this structure on Linux is shown below.

```
struct termios {
    tcflag_t  c_iflag;
    tcflag_t  c_oflag;
    tcflag_t  c_cflag;
    tcflag_t  c_lflag;
    cc_t      c_cc[NCCS];
};
```

The *c_iflag* stores the input flags that control the input of characters by the terminal device driver (strip the eight bit on input, enable input parity checking, etc.). The *c_oflag* stores the output flags that control the driver output (expand tabs to spaces, map newline to CR/LF, perform output processing, etc.). The *c_cflag* stores the control flags that control the RS-232 serial lines (odd or even parity, send one or two stop bits, etc.). The *c_lflag* are the local flags which affect the interface between the terminal driver and the user (echo on or off, enable terminal generated signals, visually erase characters, etc.).

Winsize structure keep tracks of the current terminal window size. This helps in notifying the foreground process group when the size of the terminal window changes. The fields of this structure are given below.

```
struct winsize {
    unsigned short   ws_row;
    unsigned short   ws_col;
    unsigned short   ws_xpixel;
    unsigned short   ws_ypixel;
};
```

The *ws_row* and *ws_col* fields indicate the number of rows and the number of columns in character unit for the terminal window. The horizontal size and vertical size in pixel units are indicated by the fields *ws_xpixel* and *ws_ypixel* respectively. In Linux, ws_xpixel and ws_ypixel fields are not used currently.

line discipline is a number that identifies the line discipline used by the device driver at the server. The line discipline is the part of terminal driver responsible for interpreting the input and output. Depending upon the mode of the terminal, the raw data sequence typed at the keyboard is converted to the desired form before it is given to a process. Similarly output sequences written by a process is converted to the format as desired by the user for the output on the terminal.

4.4.2 Server part

As mentioned earlier, the device specific part of the server implementation of extended protocol consists of the code that handles the d-state of the device sent along with the NFS requests and their response. There are two modules in the protocol implementation that handles the d-state of the terminal at the server.

- *Device State Restoration and Retrieval Module*

For the terminals the device state restoration function uses the ioctl function of the native terminal driver to set the d-state of the terminals using the one included in the request. Typical ioctl commands used are *TCSETS*, which sets the termios of the terminal and *TIOCSETD*, which changes the line discipline used by the terminal driver and *TIOCSWINSZ*, which sets the window size of the terminal.

As explained earlier, the device state restoration function also handles the device specific ioctl processing. The modified d-state with the ioctl request is used to set the state of the device as explained above. The ioctl commands for terminals that require arguments to be send to the server to be executed there, are the following.

TIOCCONS: This command is used to redirect the *console* input-output to a particular terminal.

TIOCSTI: This command is used to place a character into the read buffer of the terminal. The character is treated as if it is actually read from the terminal.

TCXONC: This command is used to suspend or start the output and/or input to a terminal.

TCFLSH: This ioctl command is used to flush the input and/or output buffers associated with the terminal.

TIOCOUTQ: This command is used to find out the length of the output queue associated with the terminal.

TIOCINQ: This command is used to find out the length of the input queue associated with the terminal.

POLL: This ioctl command is used to retrieve the poll mask of the device.

The server executes these ioctl commands using the ioctl function of the native terminal device driver and sends the results back to the clients.

The state retrieval function for the terminal also uses the ioctl function of the native terminal driver to get the latest d-state of the device after the operation. Typical ioctl commands used to obtain this d-state are *TCGETS*, which retrieves the termios structure of the terminal and *TIOCGETD*, which retrieves the number of line discipline used by the device driver and *TIOCGWINSZ*, which get the winsize structure of the terminal.

4.4.3 Client Part

There are two terminal specific modules at the clients - *remote terminal driver* and *XDR functions* for encoding and decoding of the d-state of the terminals. XDR functions have been described earlier.

▮ *Remote Terminal Driver*

The remote terminal driver implements the Vnode functions using the procedures of extended NFS and maintains the d-state of the terminals. The implementation structure of the remote terminal driver is kept identical to the Linux terminal driver and is shown in Figure 4.2.

Data Structures: The major data structures associated with the remote terminal driver are – *tty_driver*, *tty_ldisc*, *tty_struct*, *termios*, *winsize.* These data structures store the various information needed for device driver functioning and interfacing with the Linux kernel.

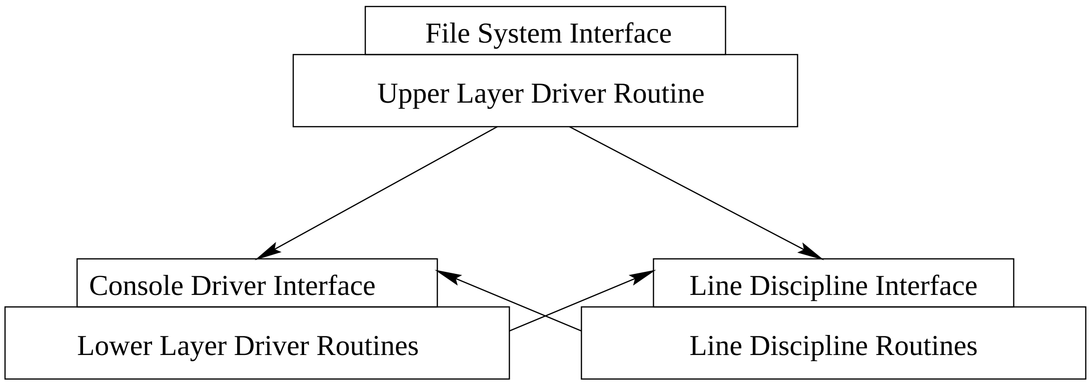
File System Interface
Upper Layer Driver Routine
Console Driver Interface
Lower Layer Driver Routines
Line Discipline Interface
Line Discipline Routines

Figure 4.2: Terminal Driver Structure

tty_driver: The remote terminal driver stores the tty_struct and termios structures corresponding to each remote terminal it is handling. This structure also defines the interface between the lower-layer terminal driver and the upper-layer terminal interface routines. The remote terminal driver need not implement any function of this interface, as the lower-layer driver functions are used to interact with the physical device interface.

tty_ldisc: This structure defines the interface between terminal line discipline and the upper-layer terminal interface routines. Since the actual line discipline processing is done at the server, the remote terminal driver need not implement most of the functions of this interface. However, it implements the ioctl and poll function of the interface, which are used for sending corresponding requests to the server, for execution.

tty_struct: This structure is used to store all of the state associated with a tty, while the tty is open. The main information includes, pointers to low-level terminal_driver interface, pointer to line discipline interface, pointer to termios and winsize structures. It also stores the count of number of processes that have opened this terminal, and major-minor number of the terminal device for which the structure is being used. This structure is allocated when a closed terminal is opened and deallocated when the last process which has the terminal open, closes it.

termios: At the client this structure is stored and maintained for crash recovery, it does not affect the actual processing. The termios structure of the device driver at the server is responsible for the actual processing of the input-output. The termios state is maintained even when the terminal is closed.

winsize: The *winsize* structure is maintained to keep track of the current terminal window size. This helps in notifying the foreground process group when the size of the terminal window changes.

Functions: Some of the Vnode operations are not applicable for terminals and are not implemented. For example, *lseek* function, returns error because a process cannot *seek* on a terminal.

Open: Opening of a remote terminal by an application at the client results in allocation and initialization of data structures of the remote terminal driver. The remote terminal driver does not send any NFS request to the server on opening a terminal. If a remote terminal is opened for the first time (after booting), the remote terminal driver allocates and initializes *termios*, *tty_struct* and *winsize* structures. If a closed remote terminal is opened, then only *tty_struct* and *winsize* are allocated and initialized. If an already opened remote terminal is reopened, the remote terminal driver returns a pointer to the *tty_struct* and winsize structures already allocated after incrementing their use count. This is used to keep a count of the number of open instances. The data structures are freed when the last process accessing the terminal closes it. For future reference, a pointer to the *tty_struct* is stored in the file table entry corresponding to the device file.

Read: The read procedure for terminals is complex as it has to handle various modes of terminal input. The procedure first validates the file handle and file attributes of the remote device file. It then allocates a kernel buffer for reading characters from remote terminal. This function uses different algorithms for reading from remote terminal depending upon the mode of the terminal (*canonical* or *non-canonical*). The data is read from the actual device at the server using the *readdevice* NFS procedure. Along with the readdevice request, it also sends the d-state of that terminal. On receiving the response of the request, the d-state embedded in the response is used to update the d-state of the terminal at the client.

Here we describe the various modes of the terminal input and their UNIX semantics. Then we explain how these modes and their semantics are implemented by the remote terminal driver.

Canonical In this mode terminal input is processed as lines. The terminal driver returns at most one line per read request, where a line is the sequence of characters up to a end-of-line character. If the number of characters entered by the user are more than requested in *read*, then only the requested number of characters are given to the read request. No characters are lost after the partial read and the next read starts where the previous read stopped. This mode recognizes and processes special input characters such as backspace, new-line and key combination for generating signals. This mode blocks the reading process till the driver receives the end-of-line character.

The remote terminal driver blocks the reading process and sends a NFS *readdevice* request to the server. The request either returns the requested number of characters (possibly less) or a special status to retry the operation. If the request returns a special status to the client, this functions waits (sleeps) and retry the operations with exponentially increasing time interval until the requested number of characters (possibly less) are read from the server. If the number of characters to be read is greater than the maximum size of a NFS request, then a single read request is broken into two or more NFS readdevice requests. In such a case, the client first sends only one request, of the multiple NFS readdevice requests. If the number of bytes in the response of the first NFS request equals maximum size of a response then only the second NFS request is sent. This ensures the semantics of *read*, even if the read request spans across multiple NFS requests.

Non-canonical In non-canonical mode input data is not assembled into lines. This mode also turns off the processing of special characters and signal generating key combinations. The read request returns depending upon the values of two variables MIN and TIME in *c_ cc* array in the termios structure. MIN specifies the minimum number of bytes before a read returns and TIME specifies the number of tenths-of-a-second to wait for data to arrive. There are four possible sub-modes depending upon the values of these two variables.

Case A: MIN > 0, TIME > 0

In this case TIME specifies value of a timer that is started only when the first byte is received. If MIN bytes are received before the timer expires, *read* returns MIN bytes. If the timer expires before MIN bytes are received, read returns the bytes received. This blocks the reading process until the first byte is received, thus at least one byte is returned to the reading process.

The remote terminal driver keep sending NFS *readdevice* requests until one or more bytes are received from the server. If the number of bytes read are equal to MIN then the *read* returns. Other wise it starts a timer which expires after TIME tenths-of-a-second. After the timer expires the client sent another *readdevice* request and tries to read at the server, if there are some more bytes to be read. It then returns with the number of bytes read up to now.

Case B: MIN > 0, TIME = 0

The *read* does not return until MIN bytes are received, thus indefinitely blocking the reading process.

The remote terminal driver keeps sending NFS *readdevice* requests to the server, with exponentially increasing time interval between successive retries, until MIN number of bytes are read from the server.

Case C: MIN = 0, TIME > 0

In this case, TIME specifies value of a timer that is started when *read* is called. The read returns when a single byte is received or when the timer expires.

In this case the remote terminal driver issues a NFS *readdevice* request to the server to read a byte. If the response contains some bytes read, *read* returns. Otherwise, if the server returns a status to try again, the remote terminal driver starts a timer for TIME tenths-of-a-second. After expiry of the timer the driver issues another NFS *readdevice* request to the server to read a single byte and returns with the response obtained.

Case D: MIN = 0, TIME = 0

In this sub-mode, if some data is available then *read* returns up to the number of bytes requested. Otherwise if no data is available *read* returns immediately.

The remote terminal driver in this sub-mode issues the NFS *readdevice* request once. If some bytes are read from the server, *read* returns with the read bytes else it returns zero.

Write: The implementation of *write* function is simpler than that of the read. It uses the NFS *writedevice* procedure to write data on to remote terminal. If the number of bytes to write is more than the maximum size of an NFS request, then two or more NFS *writedevice* requests are sent to the server. The algorithm for sending the multiple NFS *writedevice* requests corresponding to a single *write* request from application is same as the one for read. After the response of the first NFS request is received, the second NFS request is sent. This scheme is required for correct ordering of writes at the server. Before sending the *writedevice* request, this function validates the file handle and file attributes of the remote terminal file. After receiving the response of the request from the server, the d-state embedded in the response is used to update the d-state of the terminal at the client.

Ioctl: The terminals provide a large number of ioctl commands, which are handled at two levels – upper layer ioctl routine and line discipline ioctl routine. The ioctl commands that retrieve some state information are served at the client itself, using the d-state of the terminals maintained at the clients. The ioctl commands that modify the state of the terminals are made to modify the d-state maintained at the clients. The modified d-state of the terminal is then sent to the server in NFS *ioctl* request. The arguments of the ioctl commands, which are to be executed at the server, are sent to the server using the NFS *ioctl* request. Some ioctl commands are not applicable for the remote terminals accessed through this scheme. Examples of such ioctl commands are listed below.

TIOCEXCL: It marks the terminal for exclusive use. No other process (except with superuser privileges) can open the terminal after it is marked for exclusive use. This ioctl command is not supported by remote terminal driver because NFS cannot guarantee exclusive use of the terminal across different clients.

TIOCNXCL: It clears the terminal, marked for exclusive use. Since a remote terminal cannot be marked for exclusive use, this ioctl command is also not supported.

TIOCSCTTY: If the terminal is not the *controlling terminal* of a session, then this ioctl command sets it as the controlling terminal of the calling process. As explained earlier, this scheme does not allow for remote controlling terminals. Therefore this ioctl command is not supported in the remote terminal driver.

TIOCNOTTY: If the terminal is a *controlling terminal* of a session, then the session leader can use this ioctl command to disassociate itself from this terminal. This ioctl command is also not supported by the driver for the above mentioned reason.

TIOCSPGRP: This ioctl command is used to set the foreground process group-id of a terminal. The foreground process group-id of a terminal identifies the process group to which the terminal generated signals are to be delivered. It is not supported as remote signal delivery is not supported by this scheme.

TIOCGPGRP: This ioctl command is used to obtain the foreground process group-id of a terminal. Since a remote terminal is not associated with foreground process group, this ioctl command is not supported.

TIOCGSID: This ioctl command is used to obtain the session-id of the session with which the terminal is associated. This ioctl command is not supported due to above mentioned reason.

Close: The *close* function is used for clean up activity at the client. Similar to the *open* function, the remote device driver does not send any NFS request to the server on closing a remote device. If the closing process is the last process which has opened the terminal, this function releases the tty_struct and winsize structures, and removes pointers from the file table entry. If there are other processes which have the same terminal open, it only decrements the count maintained in the tty_struct structure. The termios structure for terminal is not released.

The terminal specific modules and the device independent parts of the implementation of NFS extensions, together provides the remote terminal accessing mechanism. Similarly by providing the device specific implementation for other devices, exemplified by the terminal specific implementation, one could easily provide remote access to them using the extended NFS protocol.

Chapter 5

Performance Evaluation

In this chapter we discuss the experiments conducted to test the functioning of the extended NFS system. Another objective of these experiments was to measure the overheads associated in accessing remote devices. The measure of the overheads provides a means to configure the retransmission algorithm used by the remote device driver.

5.1 Experimental Setup

In these experiments the client and the server machines used were both Intel Pentium–II PCs with 128 MB RAM, running Linux kernel 2.2.9. The two machines were connected through a 10 Mbps Ethernet LAN. The experiments were conducted in normal working conditions, i.e., average load on server and average traffic in LAN. A directory containing the device file */dev/tty8* corresponding to a virtual console of the server was mounted at the client. This allows the process at the client to access the console of the server and interact with the user sitting on that console.

5.2 Functional Evaluation

The functionality of the design and implementation of extended NFS system was evaluated by executing various existing applications, which make extensive use of terminals, over remote terminals accessed through this system. The system is able to support the complete functionality of the terminals while using all the options of these applications. The transparency provided by our framework is evident with the successful execution of these applications over the remote terminals, without any modifications.

The typical scenario of the experiments comprises of an application process executing at the NFS client, using the remote terminal of NFS server to interact with the user at the server. We executed applications ranging from simpler commands such as *ls*, *cat* etc. to complex applications such as *vi*, *shell* etc. on remote terminals.

The system recovered transparently even in the cases of server reboot. If the NFS server crashes while an application at the client is using a terminal of the server, the application is not able to use the terminal till the server reboots. While the server is rebooting the client continues to send its requests. After the server reboots, the state of the terminal is restored at the server (when the next request from the client is received) and the application at the client is able to use the terminal again.

5.3 Performance Evaluation

The appropriate performance measures for our system depend upon the kind of device being accessed. For terminals the primary measure of interest is the response time. The response time would determine to what extent the user experiences an "interactive" experience. The user of a terminal would expect a character to appear on the terminal very soon after he/she presses a key. If the terminal is in echo mode, the typed characters would be echoed by the device driver on the server itself and would therefore appear on the screen almost immediately. Consider, however a situation where the terminal is used in a non-echo mode by the application. In such a scenario the user would see the response only after the application receives the character typed and displays some output. A typical example of such a situation would be the use on an editor such as vi. The vi editor puts the terminal in raw mode and assumes the responsibility of echoing the typed characters (in the INPUT mode). In such a scenario, the response time would critically depend on the algorithm used for retransmission of read requests. Since there is no easy way to measure the response time, in our experiments we used the user's subjective evaluation of this response time to evaluate various retransmission algorithms.

Another performance measure, which is important for all kind of devices, is the network load. In case of terminals one can easily see the tradeoff between the response time and the network load. Reducing response time would require frequent retransmissions which would then lead to high network traffic. Thus one needs to tune the retransmission algorithm in a way such that while the users experiences an acceptable response time, the network traffic generated due to retransmission is also reasonable. The algorithm should also prevent excessive retransmission during long periods of user non-activity while always ensuring a reasonable bound on the response time.

We experimented with two retransmission algorithms used for terminals and compare them in terms of the network traffic generated and the quality of user experience. The two retransmission algorithms differed only in the timeouts used for retransmissions. Instead of a fixed timeout period (which would mean either a large response time or high rate of retransmissions even during the period of non-activity), we use a progressively increasing timeout value. The maximum timeout value is fixed to ensure a bound on the response time after a long period of non-activity. Figures 5.1 and 5.2 show the successive timeout values (in milliseconds) for the two algorithms used.

50, 50, 100, 100, 150, 150, 200, 200, 250, 250, 300, 300, 500, 500, 1000, 1000, 1000,.........

Figure 5.1: Timeout values used in Algorithm 1

50, 100, 150, 200, 250, 300, 500, 1000, 1000, 1000,.........

Figure 5.2: Timeout values used in Algorithm 2

We conducted the experiments with four users with different typing speeds. The users were asked to use *vi* to type for two minutes, without erasing any characters.

We measured the number of NFS requests transmitted (read, write and poll) during each typing session and also asked the users about the quality of their experience.

5.4 Results

User No	Total Chars	Read per char	Write per char	Poll per char
1	523	0.933	1.380	12.436
2	438	0.929	1.406	10.769
3	382	0.893	1.416	9.793
4	375	0.885	1.525	9.727

Table 5.1: Overhead with Algorithm 1

User No	Total Chars	Read per char	Write per char	Poll per char
1	519	0.844	1.272	8.283
2	447	0.850	1.351	8.887
3	414	0.837	1.391	9.174
4	382	0.829	1.418	9.756

Table 5.2: Overhead with Algorithm 2

Table 5.1 shows the the total number of characters typed by the four users in two minutes, the number of read, write, and poll requests sent to the server per character typed, using Algorithm 1. Table 5.2 shows the same data for the second experiment in which Algorithm 2 was used.

For algorithm 1, the first three users reported an acceptable degree of interactivity while the slowest user (user 4) reported an unacceptable quality of interaction. It can also be seen from Table 5.1 that the retransmission of the requests per character with this algorithm was substantially higher for the faster users.

For algorithm 2, the first three users experienced poorer interactivity as compared with algorithm 1, but was still within acceptable limits while the slow user

(user 4) reported unacceptable interactivity. Clearly, the number of retransmissions for all four users are nearly equivalent and quite less in comparison to the algorithm 1.

In both the algorithms, by the time the slow user types a character the retransmission timeout almost reaches the maximum possible value. Hence he/she experiences a noticeable delay in the response. We have adopted the algorithm 2 in our final implementation owing to the reduced network traffic. It was found that choosing larger timeout values, further decreased the number of retransmissions, but resulted in an unacceptable quality of experience, even for fast and average users.

Chapter 6

Conclusions

In this thesis we have presented a mechanism for transparently accessing the remote devices. We have extended NFS to access the devices of an NFS server from the clients. The extensions preserve the characteristic properties of the NFS, especially the statelessness of the protocol and transparent crash recovery.

The NFS protocol is extended to include three new procedures, viz., readdevice, writedevice and ioctl. No changes have been made to any of the existing NFS procedures. These procedures are used by the clients to access devices of the NFS server. We have also suggested a new service model for these procedures at the server. The requests and responses of these procedures also include the d-state of the device being accessed. In case of a server crash, this d-state is used by the server to set the d-state of the actual device.

To access the remote devices through NFS transparently, the clients use a remote device driver. The remote device driver of a device simulates the UNIX semantics of input-output for that device. It implements the input-output operations requested by a client application on the remote device using the new NFS procedures. It also maintains the d-state of the device, which is sent with every request for making each request self contained and independent of previous requests.

We have implemented the proposed protocol for terminals. Our implementation is primarily based on the Linux operating system. Experiments show that the response time for remote terminal accessed using this implementation is acceptable and the network traffic generated is also reasonable.

We have also integrated our extended NFS implementation with a process migration system [5]. This allows interactive process to migrate to other hosts. Since this process migration system supports transparent delivery of signals to remote processes, asynchronous notification from a terminal to a remote process also occurs transparently in this integrated system.

6.1 Future Work

We have tried to keep the extensions to the NFS protocol, device and operating system independent. But in order to validate the correctness and performance of the protocol, support for remote access to different types of devices need to be implemented using it. Additionally some of the implementation should be on different UNIX implementations. After experience with these implementations, if needed the protocol can be reevaluated and suitably modified.

To use this scheme for exclusive access to devices, an external protocol needs to be developed. Such a protocol will allow transparent sharing of even device like printers.

Bibliography

[1] Helen Cluster. *"Inside WINDOWS NT"*. Microsoft Press Publication, 1993.

[2] Xerox Corp. "Courier: The Remote Procedure Call Protocol". Technical Report XNSS 038112, Xerox Corp, Dec 1981.

[3] T.H. Dineen, P.J. Leach, N.W. Mishkin, J.N. Pato, and G.L. Wyant. "The Network Computing Architecture and System: An Environment for Developing Distributed Applications". In *Proceedings of the 1987 Summer USENIX Conference*, pages 385–398, Phoenix, Ariz., 1987.

[4] Peter Eriksson. "Standardizing/Extending the RMT (Remote MagTape) Protocol". IETF 1993 Archives, http://mlarchive.ima.com/ietf/1993/1041.html.

[5] Ashish Gupta. "Performance and Policy issues in a Process Migration Implementation". M.Tech. Thesis, Dept. of Comp. Sc. and Engg., IIT Kanpur, 2001.

[6] Jr. H.W., Lockhart. *"OSF DCE Guide to Developing Distributed Applications"*. McGraw-Hill, Inc., 1994.

[7] S. Kleiman. "Vnodes: An Architecture for Multiple File System Types in Sun UNIX". In *USENIX Conference Proceedings*, pages 238–247, Jun 1986.

[8] E. Levy and A. Silberschatz. "Distributed File Systems : Concepts and Examples". *ACM Computing Surveys*, 22(4), Dec 1990.

[9] L. McLaughlin. "Line Printer Daemon Protocol". *Request for Comments RFC 1179*, Aug 1990.

[10] B.J. Nelson and A.D. Birrell. "Implementing Remote Procedure Calls". *ACM Transaction on Computer Systems*, 2(1):39–59, Feb 1984.

[11] X/Open CAE Specification: "Protocols for X/Open Internetworking: XNFS", 1991.

[12] D.A. Nowitz. "UUCP Administration". *UNIX Research System Papers, Saunders College Publishing*, II, 1990.

[13] G. J. Popek and B. J. Walker. *"The LOCUS Distributed System Architecture"*. Computer Systems Series, The MIT Press, 1985.

[14] J. Postel and J. Reynolds. "File Transfer Protocol". *Request for Comments RFC 959*, Oct 1985.

[15] A.P. Rifkin, M.P. Forbes, R.L. Hamilton, M. Sabrio, S. Shah, and K. Yueh. "RFS Architecture Overview". In *Proceedings of the summer 1986 Usenix Technical Conference*, pages 248–259, Jun 1986.

[16] R. Sandberg, D. Goldberg, S. Klliman, D. Walsh, and B. Lyon. "Design and Implementation of the Sun Network Filesystem". In *USENIX Technical Conference Proceedings*, pages 119–131, Jun 1985.

[17] M. Satyanarayanan, J. H. Howard, D. Nicholas, R. Sidebotham, A. Spector, and M. Vest. "The ITC Distributed File System: Principles and Design". In *Proc. 10th Symposium on Operating System Principles*, pages 119–130, Dec 1985.

[18] Sun Microsystems, Inc. "JINI Architecure Specification". http://www.sun.com-/jini/specs/jini1.1html/jini-title.html.

[19] Sun Microsystems, Inc. "XDR: External Data Representation Standard". *Request for Comments RFC 1014*, Jun 1987.

[20] Sun Microsystems, Inc. "RPC: Remote Procedure Call, Protocol Specification, version 2". *Request for Comments RFC 1057*, Jun 1988.

[21] Sun Microsystems, Inc. "Network File System Protocol Specification". *Request for Comments RFC 1094*, Mar 1989.

[22] B. B. Welch. *"Naming, State Management, and User-Level Extensions in the Sprite Distributed File System"*. Ph.D. dissertation, Computer Science Division, Dept. of Electrical Engg. and Computer Sciences, University of California, Berkeley, 1990.

[23] J.E. White. "A high level framework for network-based resource sharing". In *Proc. National Computer Conference*, Jun 1976.

Appendix A

New NFS Procedures

A.1 Readdevice

Readdevice is a procedure to read data from a device at enhanced NFS server.

```
struct readdeviceargs {
    fhandle    file;
    unsigned   count;
    unsigned   offset;
    properties prop;
}

union readdeviceres switch (stat status) {
    case NFS_OK:
        fattr       attributes;
        properties  prop;
        nfs_data   data;
    default:
        void;
}
```

On entry the arguments in readdeviceargs are:

file	The file handle of the file corresponding to the device, from which data is to be read. This is used by the server to identify the file through which the device is accessed.
count	The number of bytes of data that are to be read. If the count is 0 then read will succeed and return 0 bytes. The value of count must be less than or equal to maximum read count provided by the server, in file system information.
Offset	Usually in case of devices, specially the character devices, the offset has no meaning. But in case of some block devices it may be required by the device driver to specify the offset from where to read the data from.
properties	This is the d-state of the device kept at the client. It is sent with every request to make it complete by itself. This depends upon the device being read from and include only the state which is modifiable by a process.

On success it returns readdeviceres which includes:

fattr	These are the file attributes after read operation is completed.
count	This is the total number of bytes actually read from the device. This can be less then the requested amount of data. As in case of many devices the exact count of data to be read is not known a priori.
properties	This is the most recent d-state of the device at the server. It may be different from what was sent with the request, if some other client had changed properties at the server. This must be used by clients to make their d-state of the device consistent with the whole system.
data	The data read from the device.

A.2 Writedevice

Writedevice is a procedure to write data onto a device at enhanced NFS server.

```
struct writedeviceargs {
    fhandle    file;
    unsigned   count;
    unsigned   offset;
    properties prop;
    nfsdata    data;
}

union writedeviceres switch (stat status) {
    case NFS_OK:
        fattr       attributes;
        properties  prop;
    default:
        void;
}
```

On entry the arguments in writedeviceargs are:

file	The file handle of the file corresponding to the device, on which data is to be written. This is used by the server to identify the file through which the device is accessed.
count	The number of bytes of data that are to be written. The value of count must be less than or equal to maximum write value that have been provided in file system information by the server.
Offset	Usually in case of devices specially the character devices the offset has no meaning. But in case of some block devices this may be required by the driver to specify the offset at which to write the data.
properties	This is the d-state of the device kept at the client. It is sent with every request to make it complete by itself. This depends upon the device being accessed and include only the state which is modifiable by a process.
data	The data bytes to be written on the device.

On success it returns writedeviceres which includes:

fattr	These are the file attributes after write operation is completed.
properties	This is the most recent d-state of the device at the server. It may be different from what was sent with the request, if some other client had changed properties at the server. This must be used by clients to make their d-state of the device consistent with the whole system.

A.3 Ioctl

Ioctl is a procedure to change properties of a device at enhanced NFS server.

```
struct ioctlargs {
    fhandle     file;
    properties  prop;
    unsigned    ser_ioctl;
    ioctl_param ioargs;
}

struct ioctlres {
    case NFS_OK:
        fattr        attributes;
        ioctl_return iores;
    default:
        void;
}
```

On entry the arguments in ioctlargs are:

file	The file handle of the file corresponding to the device, on which ioctl is to be issued. This is used by the server to identify the file through which the device is accessed.
properties	This is the d-state of the device kept at the client, which is modified by the ioctl. This d-state is used to modify the d-state of the actual device at the server to reflect the changes made by the client through ioctl.
ser_ioctl	Some ioctl commands does not modify the device properties but require some function to be invoked in the device driver of that device. This identifies the ioctl command to be issued at the server.
ioargs	These are the parameters to the ioctl commands that need to be executed at the server.

On success it returns ioctlres which includes:

fattr	These are the file attributes after ioctl operation is completed.
iores	This is the response sent by the server after executing one of the ioctl commands that require some function of the device driver to be invoked at the server.