AN ABSTRACT OF THE THESIS OF

Robert O. Eifrig for the degree of Master of Science in Computer Science presented on March 12, 1986.

Title: Object files in UNIX

Abstract Approved: Redacted for Privacy
William S. Bregar

Object files are named active entities (processes) in the UNIX file system graph which provide services. Services are solicited by using the object file's pathname in any system call. Other continuing services can be obtained by opening an object file (which creates an communication path to the object) or by changing the current directory of a process into the object file.

An object file is not necessarily a single named entity. A full namespace may be implemented by the object file which is *mounted* on the path to the object file node. Entities in this namespace are referenced by the suffix string which follows the path to the object node. These entities can be any anything which is managed by the object process (e.g. new types of files, devices on a network, windows, ...). The object process is the common site which is contacted for all manipulations of these entities.

Communication between the object and client processes is handled by a remote procedure call (RPC) protocol which passes parameters of the following types: data, channels and identification. When used between two processes on different machines, this protocol provides the basis for a distributed file system. The remote file manipulation occurs by executing system calls on a file passed as a channel parameter to a site which is not the native site of that file. This RPC protocol uses IP as its transport layer for communication between the two sites. In a network environment, object files insulate the object and client processes

from nameservers, network addresses, and protocols.

The basic interface between the client and object is the system call protocol (SCP). This layer of software converts individual system calls into RPCs. It is the individual SCP operations which form the alphabet in which the object process reads service requests and writes replies to these requests.

This thesis will describe the UNIX implementation of object files and how object files can be used to implement software which previously would have required extensive kernel modification.

© Copyright by Bob Eifrig
March 12, 1986

All Rights Reserved

Object Files in UNIX

by

Bob Eifrig

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed: March 12, 1986

Commencement June 1986

APPROVED:

Redacted for Privacy

Associate Professor of Computer Science in charge of major

Redacted for Privacy

Head of the department of Computer Science

Redacted for Privacy

Dean of the Graduate School

Date thesis is presented: March 12, 1986

Acknowledgement

I wish to thank the Tektronix Corporation for allowing me to use their computer resources which were not available at Oregon State University. Further, I want to thank Earl Ecklund who provided the interfacing with the Tektronix organization and donated his Magnolia workstation for use in the development of this thesis.

TABLE OF CONTENTS

1 Introduction 1

2 Server processes 5

3 Object files 8
- 3.1 Object file design goals 11
- 3.2 Object files as a file namespace extension mechanism 13
- 3.3 Names, addresses and routes 16
- 3.4 Intersecting operations 21
- 3.5 Channel and directory operations 23
- 3.6 The object exec operation 24
- 3.7 Partially indirect operations 25
- 3.8 The object select operation 26
- 3.9 The two pathname operations 27
- 3.10 The SCP operations 28
- 3.11 Extensibility 31

4 The remote procedure call mechanism 33
- 4.1 Parameter types and type checking 36
- 4.2 Structures of interacting processes. 38
- 4.3 Error notification 39
- 4.4 Caller exception handling 39
- 4.5 Request and reply parameter 0 40
- 4.6 Size limits and fragmentation 41
- 4.7 Asynchronous signals, process groups and controlling terminals 42
- 4.8 Inode binding 43
- 4.9 Closing RPC channels 43
- 4.10 The process level state machine 44

5 Network RPC communication 48
- 5.1 Establishing a host-to-host connection 49
- 5.2 Distributed file system via RPCs 49
- 5.3 Formal and actual channels 51
- 5.4 The local file server 55
- 5.5 Keepalive messages and passive closes 56
- 5.6 The remote open channel transfer procedure 57
- 5.7 The remote level RPC state machine 58
- 5.8 Heterogeneous machines 67
- 5.9 Data copying and storage allocation 68
- 5.10 Distributed process groups 70

6 Example applications of object files 72

	6.1	A simple spooling system	72
	6.2	Object mail system	73
	6.3	A resource allocator	74
	6.4	A foreign network interface	75
	6.5	A foreign file system	76
	6.6	An ISAM database	76
	6.7	The checkin – checkout paradigm	76
	6.8	A window manager	77
	6.9	Transaction processing	77
7	Implementation status		78
8	Related work		79
9	Future work		83
	9.1	A library of routines useful to object programs	83
	9.2	Support for shared text object exec operations.	84
	9.3	An identity server	84
	9.4	Improvements to the RPC communication protocol	85
	9.5	Increase the generality of object files	86
	9.6	The RPC mechanism and process migration	86
	9.7	Inter-object hard links	86
	9.8	Objects with multiple intersection points	87
	9.9	RPC forwarding	88
	9.10	Improvements to the exception service model	89
	Bibliography		90

LIST OF FIGURES

Figure		Page
1	The internal view of an object file	9
2	The external view of an object file	15
3	A nameserver query	19
4	The pathserver scenario	20
5	User and kernel states	31
6	A RPC protocol	34
7	The local RPC data structures	35
8	Caller's process level state machine	46
9	Callee's process level state machine	47
10	Remote level data structures	54
11	Caller's remote level state machine	65
12	Callee's remote level state machine	66
13	RPC message queued for network output	69

Object Files in UNIX

Chapter 1

Introduction

Object files represent a combination of (1) the object paradigm, (2) the client – server model of distributed computation and (3) the UNIX file system. The basic idea is to send messages to the object files (or objects) by executing I/O operations on a name in the file system. All of the binding and naming issues occur in the context of the file system where well established mechanisms for dealing these issues exist.

Object files are objects in the sense of the class-object-instance paradigm which are named entities in the (distributed) file system. The function of such objects is to provide services to other processes (including other objects) on request. Services are requested and delivered by means of messages between the object and its client. The object file's methods are implemented as a normal UNIX process with kernel support for message communication and creation of the object process. The class of an object file is defined by the object program which becomes the object process on demand. The instance of class represented by a particular object is defined by the object directory node in the file system graph. Instance variables consist of both data in the object process's address space and any disk resident information in the object directory. Note that object files represent large scale objects (with the granularity of an entire process) compared with objects in object oriented programming languages which have the granularity of a single procedure.

The naming, binding and rendezvous is accomplished by means of the pathname to the object directory. When a client wants to send a message to a particular object, it must use a pathname containing the object's object directory in a system call. Note that there are no network addresses or process identifiers involved in this operation. The message sent to the object specifies the (1) the type of system call executed by the client, (2) the identity of the client, (3) the suffix of the pathname (the part of the pathname beyond the object directory node) and (4) any other parameter required by the object process. The reply by the object process to this request is exactly the result required to complete the system call.

The set of UNIX input/output system calls is the common interface shared by all object files. This interface has the property that all existing software can function in the client role without modification. There is, of course, a means to extend this interface to provide arbitrary messages to objects.

Consider the following example. A print server is implemented as an object file with the object directory path of "/obj/lpr". To use this object file, a client could execute

pr foo.c > /obj/lpr

The shell, in response to this output redirection, will execute a *creat* system call on the "/obj/lpr" path. This action will cause the kernel to detect an object file operation when the pathname traversal routine encounters the object directory node (the object directory is structurally a UNIX directory which is uniquely tagged to distinguish it as an object directory). Next, the kernel will arrange for the object process to be created if it does not already exist. Once the object process is created, a message specifying that the parameters of the *creat* system call is delivered to the object process.

The shell (or the client process in general) is blocked within the *creat* system call until the object process responds to the *creat* message. The object process must return an open file descriptor (or an error) to the client to satisfy the pending *creat* operation. There are no restrictions on the type of descriptor which can be returned, however a pipe or disk file (for holding the data to be printed in the output queue to the print device) are probable choices. At this point the *pr* process is created, inheriting the descriptor returned by the object process as its standard output. The *pr* process will execute a sequence of write system calls followed by a close. The object process will be involved in the data path if an interprocess communication descriptor (such as the pipe) is returned, however this is not necessary.

The same scenario as described above works if the object directory resides on a different machine from the client. In this case the pathname to the object process must go through a distributed file system which supports the object file abstraction. In this case the *creat* message is formed on the client's machine, transferred through the network to object's machine (the object's machine is always the machine which contains the object directory in its local file system). The kernel on the object's machine will create the object process (if necessary) and deliver the *creat* message. The reply is transferred back through the network to the client process.

Another point is that all parameters required by the print operation are communicated to the object via the file name string. For example

pr foo.c > /obj/lpr/laser1/"copies=3 banner=foo.c"

shows a likely syntax for the print server. In this case the pathname suffix, "laser1/..." is passed as a string to the object process. The object process can parse this string to get the parameters requested by the client.

Notice that the "/obj/lpr" object files is now implementing more than one logical object. The logical objects are the printer queues for each distinct printer. The particular object being invoked is determined by the pathname suffix (laser1 in the example). In other words, the object file implements a namespace of objects (which can, in UNIX terminology, be considered mounted on the UNIX file system with the object directory being the mount point). This is a natural consequence of the large scale object implementation: combine collections of related objects into a single object file (and thus a single object process). Interactions among the objects within an object file are cheap compared to external interactions which are more costly. This concept is more clearly illustrated by an object file which is a foreign file system driver. The internal objects are the foreign files themselves. The foreign files are accessed by name via UNIX system calls and the naming conventions are particular to the type of file system being emulated. In general there is a complete structured directory system which is used to translate the pathname suffix into the target file being accessed. An example of the interactions among objects within the object file would be the deletion operation where both the target file and the directory holding its name are affected.

As a result of the addition of object files to UNIX, one can consider the UNIX file system to be a directed graph of objects. The original UNIX files (regular files, directories, ...) can be considered primitive objects. The non-primitive objects must be synthesized from object file implementations of those objects. Further, pathnamed references to objects can extend from a directory which is inside of one object file back into the public (as opposed to the private, on-disk memory of the object file implemented within the object directory) UNIX file system. In fact, such a reference can extend into yet another object file.

The discussion above is intended to give the reader informal description of what object files are and how they can be used. The next chapter deals with the existing facilities for server processes and which improvements are needed. Chapters 3, 4 and 5 are devoted to all

the gory details of the motivation, design and implementation of object files. Chapter 6 describes (very briefly) additional applications of object files. The status of what has actually been implemented is described in chapter 7. The next chapter describes related work. The last chapter describes areas which need additional investigation.

Chapter 2

Server processes

The motivation for the design of object files[1] is the concept of a *server process*[2]. A server process is a request interpreter where the requests originate from other processes (as distinguished from requests typed directly by a user (like an editor) or which are stored on fixed file (like a shell script)).

In a multi-processor computer system servers provide one of the basic mechanisms for the utilization of multiple processors. Even in single processor computers, a multi-process application is often structured about the idea of a server to provide (1) a common site for a particular computation, (2) segregation of function (i.e. a convenient logical structure) and (3) as a serialization or mutual exclusion mechanism (i.e. a monitor). The usefulness of the server concept will be assumed as a premise in this paper so it will not be justified further.

The implementation of server processes under UNIX[3] is a difficult programming task for the following reasons:

(1) All of the interprocess communication facilities are based on the idea of passing data or sharing data between processes. The conversion of service requests into messages requires the use of a protocol which must be designed for each application. The protocol implements the details of the packaging of messages plus the synchronization between client and server. Although this scheme provides a general and flexible approach, it is idiosyncratic and error prone. In addition, the protocol routines have to be duplicated among all of the client(s) and server(s).

---

[1] The term *file* will be used to mean a named disk resident I/O object. Regular files, directories, special files are all files. A socket is not a file in this sense, even if it is bound to an inode as in the AF_UNIX sockets, because the disk based information alone cannot create the socket. Likewise a 4.2 BSD pipe is not a file, however a System 5 FIFO (a named pipe) is a file.

[2] Definitions and terminology will be introduced in *italics*

[3] UNIX [Rich74] is a trademark of AT&T. In this paper the term UNIX will refer of any the UNIX or UNIX-like systems unless a specific version is mentioned. All discussion of implementation is done under a 4.2 BSD UNIX derivative system.

(2) The invocation of the server processes is also a source of difficulty. The first problem is avoiding the race condition which exists between multiple clients in starting a single server process. A second problem is to insure the server is completely independent from the client which created it. To avoid these problems, many of the server processes are created at boot time and live forever. Clearly this approach is wasteful of system resources.

(3) Another problem is the reliable authentication of the client's identity by the server. In the case that the server is to provide remote execution of arbitrary software or manipulate arbitrary files, it is crucial to the security of the system that the server determine exactly which access rights belong to the current client. This must be done in a way which prevents the client from stating a false identity. Under UNIX the access rights are more complex than saying "the request originated from a client process run by a user who logged in as foo". Processes can have multiple and transient identities due to execution of set-user/group-ID files and by manipulation of the set of groups to which this process belongs. All of this information must be communicated to the server.

(4) In addition to request and reply parameters which can be represented by data structures in memory, some services require the use of open *channels*. A channel is an open I/O connection which can be shared among processes; channels have a one-to-one correspondence with entries in the UNIX kernel's file[] table. In UNIX, the semantics of open channels is complex and cannot, in general, be duplicated by re-opening files. Consider, as examples (a) a file which has been unlinked, (b) a socket (pipe) which does not have any name and (c) a terminal where the client is a member of the terminal's process group. The only solution is to pass the channel to the server by reference where all manipulations of the channel done by the server are done in the client's context.

(5) The development and debugging of multi-process systems involving servers is difficult. One difficulty is the inability to use interactive debuggers on the server processes without disturbing their context. The problem which occurs is that the process being debugged (i.e. the server) inherits too many attributes from the debugger which is restricted to be the immediate parent of the server. As a result, debugging of servers relies on more primitive techniques such as diagnostic output placed in log files.

(6) The rendezvous between client and server is accomplished by means of both processes having knowledge of a common address. Obtaining agreement on the address means

consulting a table which maintains the correspondence between the logical function of a server and the address. Some related issues are (1) how to initially allocate a unique address for a new server and (2) how to prevent un-authorized use of that address (how to keep this address from being stolen by another server).

Chapter 3

Object files

The implementation of object files is an attempt of solve some of the difficulties described above with the use and construction of servers. In short, an object file is a server. As is implied by the name *object file,* it is also a named entity in the file system.

The term *file system* in the case of a distributed computer system refers to a distributed file system which is the union of the local file systems on all of the sites in the distributed computer system. This distributed file system is implemented with ease using the object file mechanism and will be discussed later in the paper.

An object file consists of two parts: (1) a special type of node in the file system graph called an *object directory* and (2) the *object program* which is the server program. The object directory can be placed anywhere in the file system and it is structurally identical to a UNIX directory. Object directories are distinguished from ordinary directories by a unique inode type code. The object program is a UNIX executable binary file which can be produced by any of the compilers available under UNIX. The object program can also be a executable interpreter script. The internal structure of an object file is shown in figure 1.

When an object directory is referenced by any pathname parameterized system call, a server process is automatically created if it does not already exist. This process, called the *object process,* executes a copy of the object (or server) program. Once the object process is in existence (either the new object process or an existing object process), the system call executed by the client is communicated to the object process by a *remote procedure call* (RPC) protocol. The object process can then service this request and reply to the client. An object is *active* when its object process is in existence (and has the RPC channel open and ready to service a request) and an object is *passive* otherwise. Any system call which requires a RPC to the object process is said to *invoke* the object.

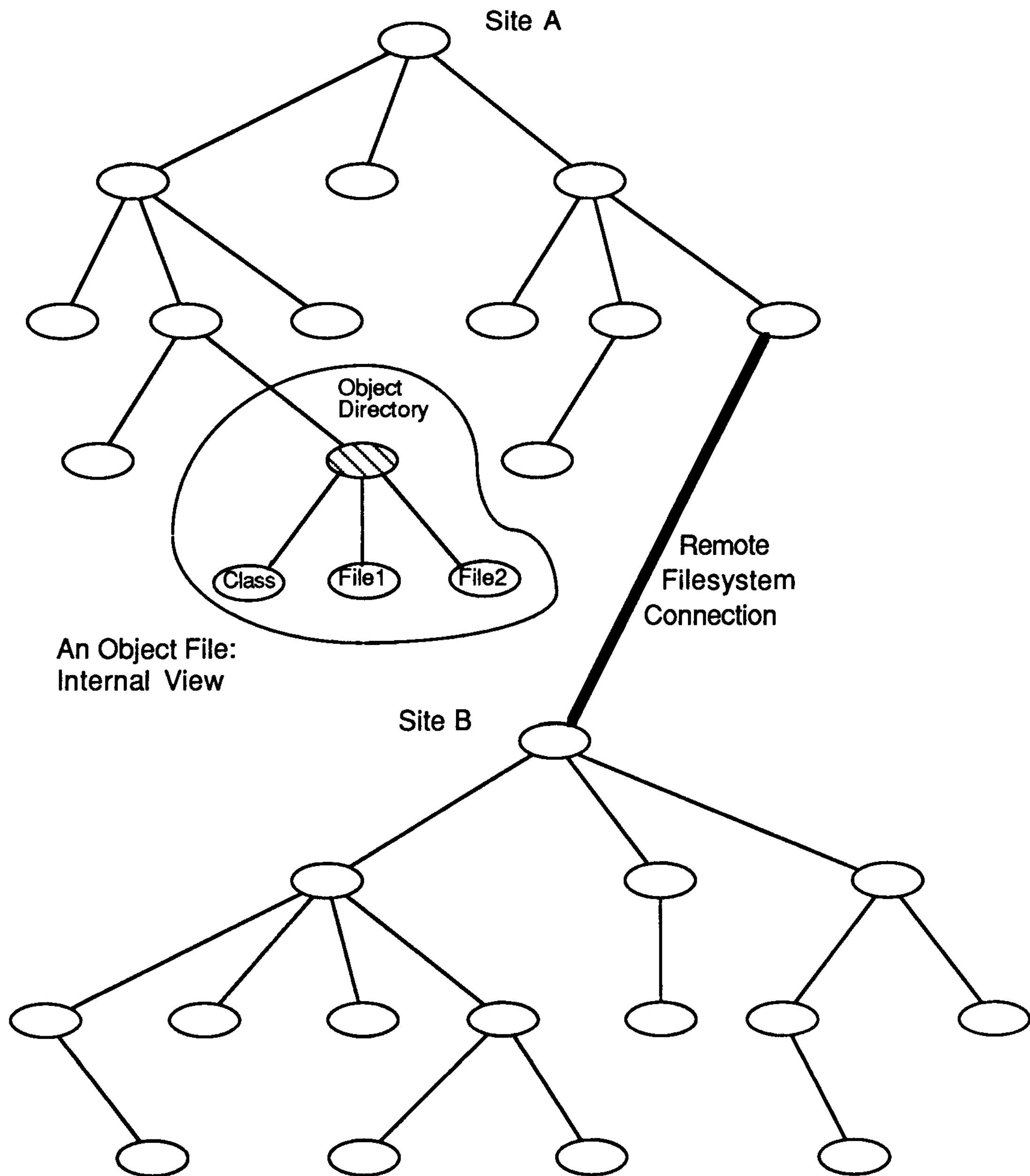

Figure 1: The internal view of an object file

The term object file will be used in the singular throughout this paper, however that is not necessarily the case. Additional object processes associated with a single object directory

can be used to provide better service (by implementing multiple servers) or to service individual requests or service individual RPC channels (see section 3.5). All of these additional processes are forked by the original object process created by the activation event.

When an object file is activated, the object program is located by uses of the name of the operation (e.g. "open" or "stat") in the object directory. If this name does not exist or this file cannot be successfully loaded, the generic name "class" is tried. If this fails, the entire object operation fails.

The term *system call* needs further explanation. System calls are those functions described in section 2 of the UNIX programmer's manual [Ucb83] which request a service of the UNIX kernel. For purposes of the discussion of object files, the system calls of interest are those functions which are parameterized with a file pathname or by a channel. The existing system calls are encoded into RPCs by the a *system call protocol* (SCP) which will be discussed in detail later. There are three types of object operations: (1) intersecting operations (see section 3.4), (2) channel operations (see section 3.5) and (3) directory operations (described in section 3.5). Each of these will be discussed later. The term *operation* is used to be synonymous with system call.

The name *object file* is taken from the object - class - instance paradigm. The object terminology (as used by Smalltalk [Gold83], for example) can be defined as follows. Object files are members or instances of a class of objects. All objects in the same class share the same object program. The algorithms in the object program define the object's methods. The interface to the object is the set of RPCs to which it will respond (meaningfully). Each object has its own private memory. The private memory of an object file is (1) the data address space of the object process and (2) the on-disk entities stored in the object directory plus its hierarchical descendants. Note that other object files can exist within the private memory of an object. Further, an object can invoke itself or other objects in the course of servicing a request. The analogy can be pushed further to include all non-object UNIX files as object file primitives. The non-object UNIX files will be called *primitive* objects. The UNIX system call interface is used as a common interface for the invocation of object files. Other object related notions (e.g. sub-class and super-class) do not have direct analogies with UNIX object files.

3.1 Object file design goals

(1) An object file should pose solutions to the list of problems with the implementation of servers as enumerated in the previous section. The solutions are:

(1.1) The complexity of the server - client protocol is avoided by the use of the RPC mechanism. A (remote) procedure call is a procedural interface which is much easier to utilize by procedural languages (such as C). Further, the system call interface is also a procedural mechanism. All of the existing IPC schemes available in UNIX require the client and server to communicate using a message passing paradigm. It is task of the protocol to do this conversion. In this sense the RPC discipline is a distinct mode of interprocess communication which is separate from (1) virtual circuits (byte streams), (2) datagrams and (3) shared memory.

(1.2) The SCP layer within the kernel automatically handles the creation of the object process on an on-demand basis. Further the object process is free to die at will (usually when it remains idle for a certain timeout period). When the object process is created, it has the following state:

- The owner and group IDs are those of the object directory. The object program file can, in addition, use the set-user/group-ID bits in the object program file. The object process has all restrictions and privileges of its ID, except the resource limits set by setrlimit() are infinite.
- The parent of the object process is /etc/init (see (1.5) below).
- The object process' current directory is the object directory. The root directory is always the local root.
- The object process receives only one channel (open descriptor) which is the conduit by which the RPCs are received.
- All other process state information is set to default values.

(1.3) Each SCP operation contains an identification parameter which is provided by the kernel on behalf of the client which specifies the real and effective

user and group IDs plus the client group vector. Identification parameters are always defined by the kernel (unless the client is the super user) which makes them impossible to forge. The tagged parameter nature of the RPC interface (described in detail later) makes it impossible for any other data to masquerade as an identification.

(1.4) The RPC mechanism supports passing channels between processes. There are no restrictions on the type of channel that can be passed.

(1.5) Debugging of the object process with the full facilities of the ptrace system call is supported. If a traced process attempts to execute an object exec system call, the process executing the exec will become the object process in the state described in (1.2) above except that the parent process attribute is maintained. This only works if the process executing the exec has the same UID as the object directory and the object is passive.

(1.6) The rendezvous between client and server is accomplished by the use of the object directory's pathname by the server. The acquisition of the common address between client and server is reduced to the problem of the client knowing the pathname to the object directory. All of the techniques used to specify pathnames are available to the client (specification of a sequence of directories to search is a common example). The allocation of an address for a new server (i.e. choosing a new object directory name) is much simpler due to the hierarchical nature of the file system. The final component of the object directory's pathname need be unique only among the members of the immediate parent directory.

(2) Object files should provide a mechanism for transparently simulating all of the semantics of UNIX I/O system calls. If the object process were designed to do so, it should be impossible for the object's client to detect any difference between invocations of the object and access of any non-object file in UNIX. This property is crucial to the implementation of a distributed file system based on object files.

(3) All processes which do channel I/O can interact with objects. Further, this must be done without modifying any of the process level software (which excludes relinking with new libraries). In most of the software systems involving servers there are only

certain processes which have the necessary interface to communicate with the server. By use of the SCP in place of or in addition to the application specific RPCs, many of the existing UNIX tools and utility programs can be used with any server process written as an object file. In this way the UNIX I/O operations form a basis for a uniform set of operations which can be used with all objects and is independent of the specifics of any particular object.

(4) The implementation of objects should hide the differences between (a) client and server executing on the same processor, (b) client and server executing on different processors connected via a network, and (c) client and server executing on different processors connected via a shared memory.

(5) The RPC mechanism should be usable as an IPC facility without using object files.

(6) The object file mechanism can be used as a means to migrate software into and out of the kernel itself. The basic idea is that each object operation (RPC) between the object and client process would have a one-to-one correspondence to a real procedure call within the kernel if the object were "kernelized". The main benefit is that the object interface is very close to the user process. The means that an object in the kernel would have maximum functionality but it would be difficult to share both procedure and data structures with other parts of the kernel (e.g. the inode table or buffer cache).

3.2 Object files as a file namespace extension mechanism

When a pathname operation invokes an object, the *pathname suffix* is passed to the object. The pathname suffix is the remaining part of the pathname following the "/"(s) which terminate the object directory component name. This property is extremely useful for the following reasons:

First, parameters to be used by object files can be encoded into the suffix. This is a transparent method of passing parameters since almost all existing UNIX software assume no structure to the name except for components separated by slashes.

Second, this suffix can be used to append new namespaces to the UNIX file namespace using the object directory as a *mount point* (analogous to the UNIX mount operation). A

UNIX mount operation connects a new file system to the existing file system by binding the root of the new file system to an existing node (called the mount point) in the original file system. The root of the new file system (and it hierarchical descendants) is referenced as if one were accessing the node to which it was bound.

To elaborate on the second point: The file namespace is considered to be the domain of a function which maps strings into entities. The strings have the existing syntax where a full pathname is composed of a sequence of component names separated by slashes. An entity is anything which deserves a name. Entities can be physical (e.g. computers and I/O devices) or logical (e.g. files, processes, messages). The collection of entities includes as primitive entities all UNIX non-object file types (regular files, directories, special files, symbolic links, and fifos). All other entities are implemented by means of object files. Figure 2 show the external view of an object file where the entities are shown as rectangles. Compare this figure with figure 1.

A generalized directory is any mapping between component names and entities. This structure gives the total namespace the character of a directed graph where entities are nodes and the edges indicate the node referenced by the generalized directory. A generalized directory would be any node without degree greater than zero. Note that a symbolic link (or any other reference by name method) is not an edge in this digraph. Objects which implement directories of entities within that object (i.e. the object process is the manager/manipulator of those entities) can store directories in any way. When referenced by a client, a directory entity is assigned the status (and other characteristics seen through the system call interface) of a UNIX directory. The entries in that directory are returned in the structure of a UNIX directory entry when the generalized directory entity is read.

Hard out-links to primitive entities outside of the an object file can be implemented by storing the hard links in the private on-disk memory within the object directory. Hard in-links (either between entities or from UNIX directories outside the entity) cannot be easily simulated. More work is needed here.

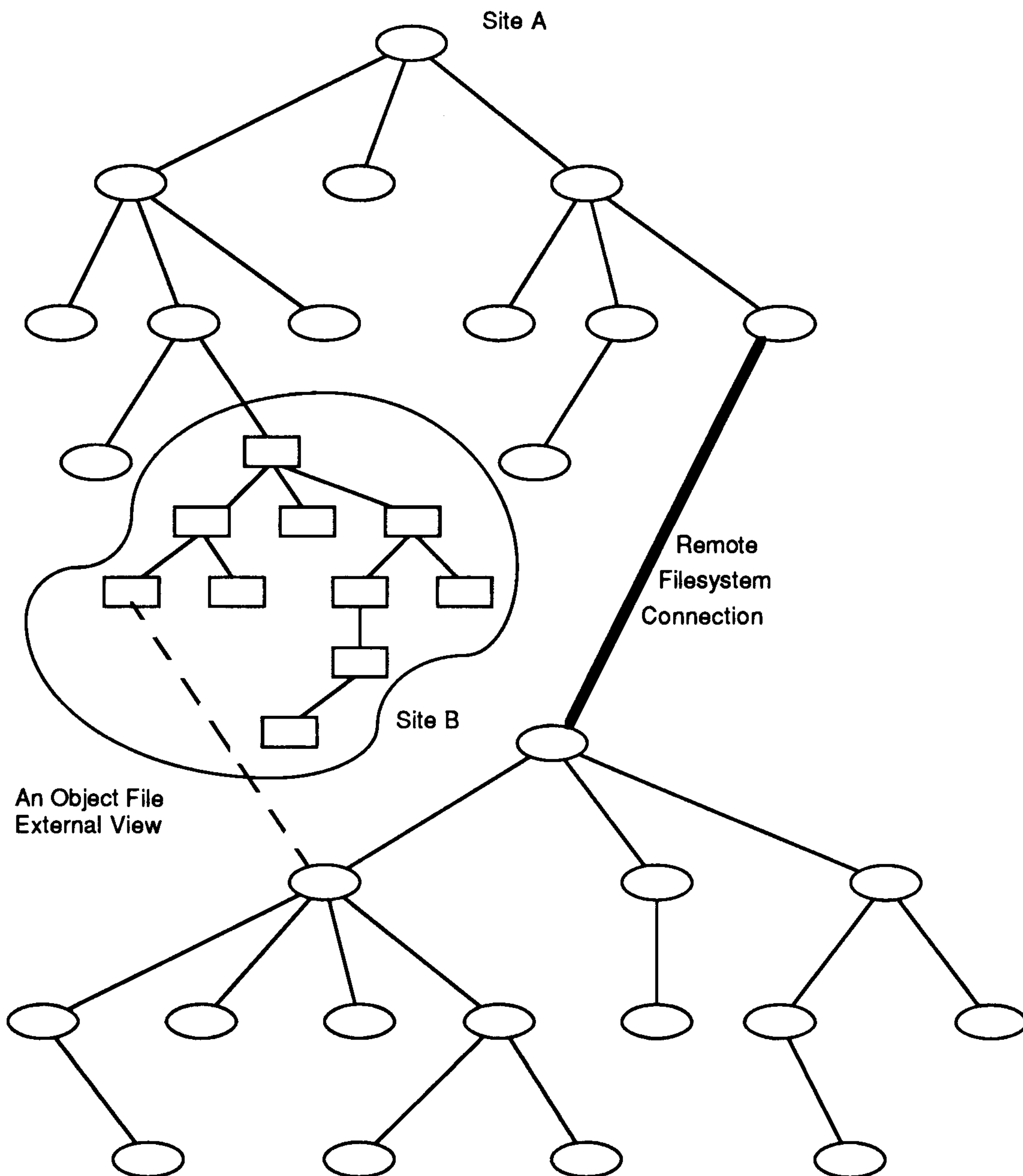

Figure 2: The external view of an object file

In order to make use of existing UNIX software whenever it is applicable the following UNIX name conventions should be used even within an object files namespace. These are

"soft" conventions which permit the concept of hierarchies (as seen by the UNIX utilities) to be extended into object files:

(1) Slash must be used to delimit hierarchies.

(2) "." is a self reference for a directory entity.

(3) ".." is a reference to the parent directory.

3.3 Names, addresses and routes

Distinctions between names, address and routes must be drawn. A *name* indicates the entity we want. It usually has properties such that it is human-readable, mnemonic and independent of network location. The name is also the mutually known (by client and server) item of information which is needed to accomplish a rendezvous.

The *address* of an entity is its "physical location" at the current instant. Note that an entity is not guaranteed to have a fixed address. It can "move" between activations. In a local UNIX file system the address of a primitive entity was the device code, inode number pair. By analogy in a distributed entity context, the address of an entity is now the device code, host address, inode number triple. For an object file, the object process can obtain a unique (unique to the host on which it is executing) major device code. The minor device code can be assigned at the discretion of the object process. The inode number is any integer which has a one-to-one correspondence with all entities implemented by the object. The host address is a unique net host address used in the network context.

The only useful property of addresses it that if the addresses of two entities are the same while they are both open, then the two entities are the same (not identical copies but the same entity). Addresses cannot be used to access entities. The stat/fstat/lstat system calls return the address of an entity.

A *route* is a description of how to get to an entity. In the context of names, routes and addresses, a pathname is a route. The object file mechanism leads directly to the the remote mount style (all pathname references to a particular remote machine pass through a single node in the file system graph called the mount point) of distributed file system. The way this works will be discussed in more detail later. In this way the route to a remote site (and to all services available on that site) means knowing the pathname through the mount point.

In a network operating system, the network is made visible to the user. The command language and programming interface in a network operating system is typified by "execute function x on site y using input z from site w". Clearly the object file mechanism provides this because the object programs are always executed on the site containing the object directory and the particular site of execution is implied by the pathnames used.

A distributed operating system is one where all sites in the network run the same operating system which presents the illusion of a single machine (the network is completely invisible). The object file mechanism will not make a distributed operating system, however it can go beyond the network operating system model by making the network transparent to the acquisition of services provided by object files.

The most important characteristic of a name which is not provided by the pathname or route is location independence. Clients can be moved. However if an object changes its path, all clients must be changed (or their configuration parameters or symbolic links to the destination object, et cetera). One reason for moving an object is to reflect a change in the up/down status of a site while continuing to provide a particular service in the distributed environment. The location of a site is usually encoded into the pathname to the mount point. To avoid using pathnames which include this location dependence, the location dependence can be placed in search paths or symbolic links. As an example, let "/services" be a directory containing the names of services, each of which would be directories of symbolic links to the object files which are the providers of the service. The site of the servers would be contained in the contents of each symbolic link (i.e. the route). A client would access the directory of providers of the service, trying each until the service can be provided.

The example above suffers from two problems. First, exhaustively trying the various providers of services is expensive and is duplicated by each client. Second, the mapping of names to routes done by the symbolic links is static. Both of these problems can be solved if the directory of providers of a services were replaced by a single symbolic link to a single server which was known to be available. This single symbolic link would be adjusted by a nameserver-like program, called the *pathserver*. The pathserver is a mixture of both nameserver functions, higher level routing services and an object file interface. The pathserver adjusts the symbolic links in "/services" according to (1) the up/down status of the service site, (2) the communication costs and load balancing concerns among the service providers and (3) dynamic registration and de-registration of servers.

Note that "/services" itself could have been made into an object directory where the pathserver would be the object process, but this method is rejected because each object operation would be indirect. Such a method would allow the pathserver to be more integrally involved and should be used whenever access through the symbolic link in "/services" fails. In this scheme, the pathserver would be the object process of the object directory "/Services". Further, an identical hierarchy beneath these top-level directories would be used to identify the particular service.

A new service would be installed or deleted by attempting to create or delete a symbolic link in "/Services". The pathserver will propagate these changes to other sites and to the local "/services" directory. Additional directory levels in "/services" and "/Services" can be used to maintain private services and servers (the access permissions for these subdirectories can be used to control access without actually invoking the servers). Other logical subdivisions, such as experimental, can also have separate subdirectories.

The typical role of a nameserver in the client server model is shown in figure 3. Compare this to figure 4 which shows the corresponding scenario for the pathserver. Note that the "/services" directory and its descendants constitute a cache of the pathserver's translations so that the pathserver need not be involved in each transaction. Objections to the object file mechanism on the grounds of the inefficiency of a file system based mechanism should include the overhead of the nameserver query in their analysis. Except in small systems, the namserver's database is disk resident and the nameserver itself resides in virtual memory. Both object files and the typical client-server system using a nameserver cache the disk resident entities so that commonly used bindings need not reference disk.

The following observations can be made about the object file approach to the distributed client - server model: (1) The use of a name agent has been effectively eliminated. (2) All of the UNIX utilities are available (provided that SCP is spoken, it may be that SCP is only used for diagnostic and configuration purposes). (3) The pathserver is optional without sacrificing the name - route - address distinction. (4) Addresses are never used. (5) The client (including all libraries down to the lowest level functions in the client's address space) is never involved in communication, synchronization and other protocol issues. Clients totally avoid direct contact with nameservers, addresses and communication protocols. These tasks are assumed by the operating system in a manner exactly analogous to the way physical disk block addresses are hidden from processes in the operating system.

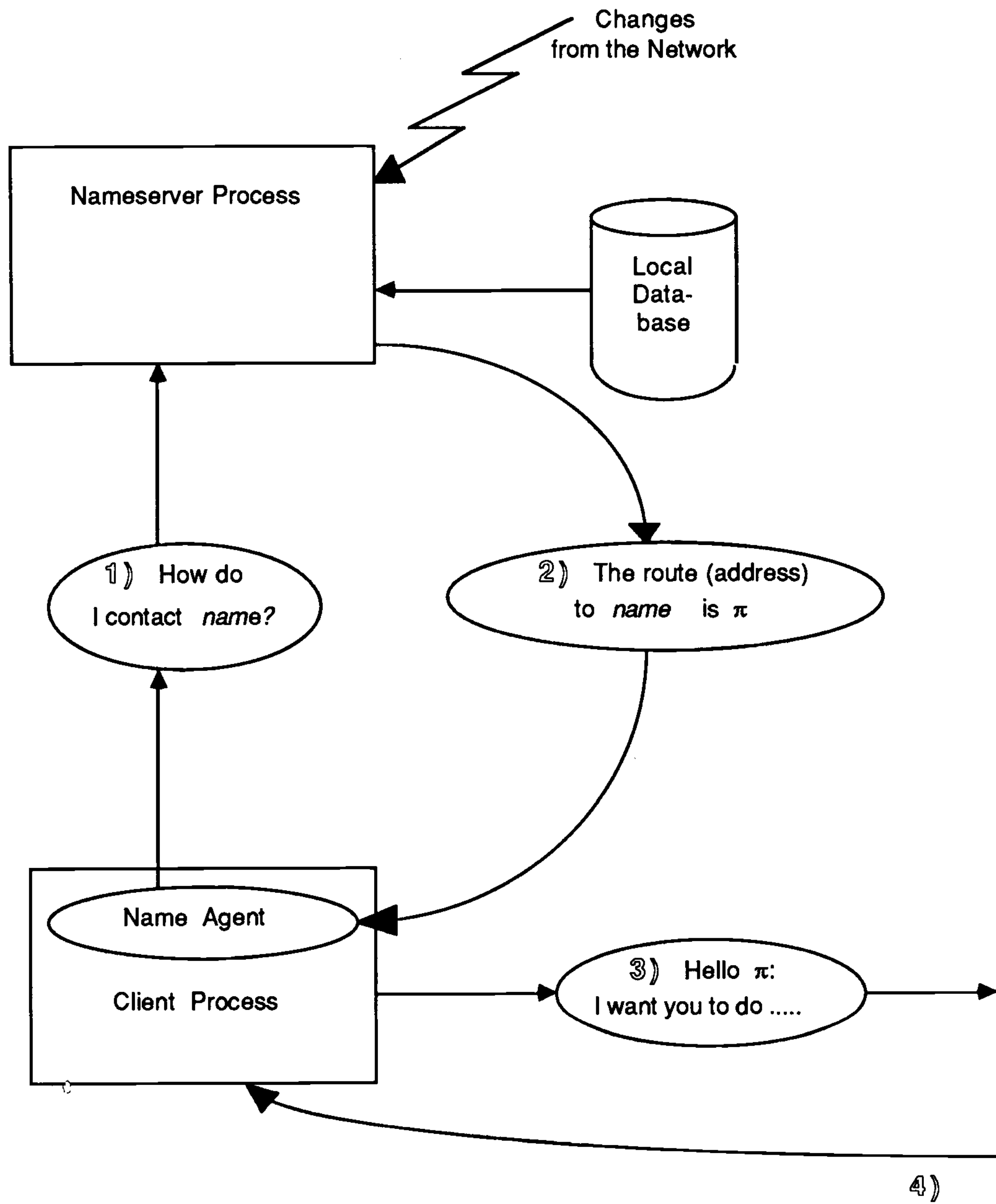

Figure 3: A nameserver query

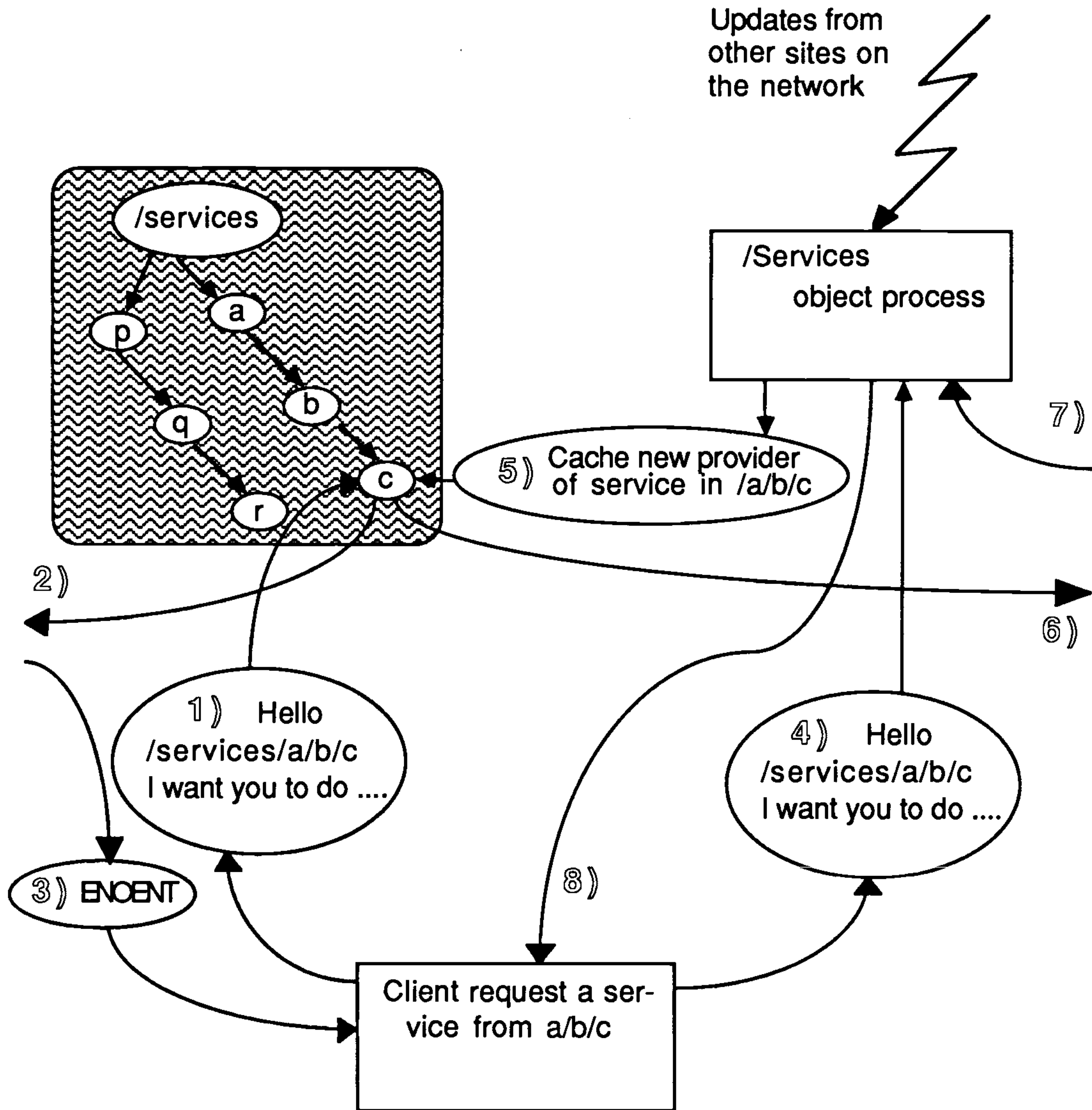

Figure 4: The pathserver scenario

Nameservice for host addresses used in the establishing of the remote mount connections is handled at a lower level (described in chapter 5).

3.4 Intersecting operations

Intersecting operations are pathname parameterized operations where a pathname walk originating from outside (explained later) an object intersects the object directory. An intersecting operation is always the activating operation. When the object is activated, the only descriptor provided to the object process is the callee channel of the RPC conduit used to communicate intersecting operations. This descriptor (channel) is called the *intersecting descriptor (intersecting channel).* If the intersecting channel is closed (or if the object process exits) the next intersecting operation will create a new object process. If there are several processes associated with an object, the object is considered active as long as at least one process has the intersecting channel open. Also note that non-intersecting operations (see section 3.5) can still be serviced by a passive object. If the intersecting channel is closed before a intersecting request can be read by the object process, the kernel will try (a limited number of times) to activate the object by creating a new object process and re-sending the request.

There are times when an object directory (and its hierarchical descendants) must be referenced without invoking the object. Referencing the object directory without invoking the object is called *suppressing* the object. When an object is suppressed, part of its private memory not in the object process' address space is being accessed. This can be done by using the at-sign (@) as an escape character within the pathname which intersects the object directory as specified below. Normally an object directory can only be suppressed by its owner, however if the ISGID bit is set in the object directory any process in the owner's group can suppress. If the ISUID bit in the object directory is set, then anyone can suppress the object.

The syntax of the at-sign is overloaded and described by the following rules which apply in order:

(1) Two consecutive at-signs are mapped into a literal at-sign.

(2) If a pathname begins with an at-sign followed by an unsigned decimal integer followed by a slash, then the decimal integer (descriptor) selects a channel which should be a directory that is used as the starting reference point for pathname traversal. If there are no components following the first slash, then the channel need not be a directory however it must be either an inode or RPC type. This allows channels to serve as additional

working directories that can be used for the reference point at which a pathname traversal begins. A more important feature is that current directories can be passed through the RPC mechanism as channels.

(3) If an at-sign immediately follows a slash and the last member of a path is a symbolic link, then the substitution of the symbolic link is suppressed. The idea here is to avoid the fixed, per-system call decision as to whether or not to expand the last symbolic link.

(4) All at-signs used to suppress an object directory must occur before or during the object directory's component name.

(5) If an at-sign immediately precedes a slash, then an indefinite number of object directories following the slash along the path can be suppressed.

(6) An at-sign in any other context will suppress a single object directory. Multiple at-signs can be used to achieve a fixed maximum depth of suppression and excess at-signs are ignored.

(7) At-signs which occur in the pathname suffix of an intersecting operation will be passed literally to the object process.

The addition of redundant "/." and "/.." sequences will permit the at-sign to be used in any context above.

References to an object directory by ".", "..", or by the current directory of a process established by a chdir system call parameterized by a suppressed object directory path do not result in intersecting operations. Instead, the object directory is treated as an ordinary directory in these cases, except that the inode type field will contain a new code which denotes an object directory (symbolically IFOBJ). When an object process is activated, the current directory of the object process is the suppressed object directory to permit easy access to the on-disk private memory.

Creation of a new object directory is done using the mkdir system call where the high order bits of the mode parameter must specify the object directory type code, IFOBJ. When a new object directory is created, the object program and any other files which make up the private memory of the object can be installed using suppressed pathnames into the object directory. The removal of an object directory is done by a rmdir with a suppressed pathname to the empty object directory.

3.5 Channel and directory operations

Channel operations are operations with invoke the object file by means of system calls on a channel created by the object. Directory operations invoke the object process because the current (or root; all future references to current directory all understood to refer to either the current or root directory) directory was established inside an object file. A directory operation has the property that it will invoke the object regardless of the specific pathname used, all that is required is the particular operation and the particular starting reference point in the file system graph. Channel and directory operations are similar because the creating operation (an object open or an object chdir) returns a channel to the client.

Operations parameterized by descriptors which invoke an object are always channel operations. Operations parameterized by a pathname which invokes an object can be intersecting, channel or directory operations. A pathname object operation is intersecting when a pathname originating outside the object intersects the object directory. A pathname channel operation begins with the "@<*fid*>/" sequence. A pathname directory operation starts from a directory inside the object. Note that for channel and directory pathname operations (a) the at-sign suppression mechanism does not work and (b) the entire pathname (excluding any "@<*fid*>/" or "/" prefixes) is passed to the object process in place of the suffix.

When a channel is returned by an object process, it can either return a RPC caller channel (the caller and callee channels associated with a RPC conduit will be discussed in chapter 4) or some other kind of channel. If this channel is an a not a RPC caller channel, then all operations on the channel are said to be *direct*. A direct operation does not invoke the object (which returned this channel or current directory to the client). Note that pipes, sockets, or pseudo-tty channels can be returned to client which connect the client and object processes. The client's operations on these channels are considered direct. Also an object can return a RPC caller channel of another object to its client. In this case, an operation by the client on the returned channel is direct with respect to the object which return the channel but is not direct with respect to the object whose RPC caller channel was returned. A direct operation is not an object operation; the term "direct" is only used to emphasize that the channel or current directory associated with the operation originated in a object operation.

If an object process wants to service each channel or directory operation following a open or chdir operation, then a RPC conduit where the caller RPC channel is returned to the client should be used. This channel can be dedicated to the client or shared among several

clients. Operations which require the object to respond to a RPC are called *indirect*. When a RPC caller channel is returned, all operations on this channel will result in RPCs via that conduit. Note that there is nothing in a SCP request or reply messages which identify (1) the client's descriptor, (2) the client process or (3) which channel to the object is being manipulated or (4) which entity is being manipulated. The object process must infer this from the channel used to read the request.

The terms inside and outside an object have been used informally above to describe the origin of a pathname with respect to an object directory. To be precise, a current directory (or root directory or open descriptor used as the starting point for a pathname walk) is *outside* an object if there exists a pathname relative to the origin directory such that no operation parameterized by this pathname will invoke the object. If directory is not outside an object, then it is *inside* the object. An object is *regular* if every directory entity which is inside the object is for one client is inside the object for all clients which can access that directory.

3.6 The object exec operation

A related operation is the exec(2) system call which also returns a channel via the SCP reply. The function of an exec operation is to completely replace the user image of a process. In the case of an object exec, the new user image is obtained from an object file. In this sense, the object process must provide a mechanism for the kernel to obtain the data which comprises the new image. This is done by returning a channel from which the image can be read. This channel can be either an executable regular file (the operation is a direct exec in this case) or a RPC caller channel (the operation is an indirect exec). Further, the same two alternatives are available if the loading of a script interpreter results in an object operation.

When an object exec occurs, the kernel itself will issue the object exec followed by a sequence of object reads on the returned channel to load the data. Note that object exec operation returns, in addition to the channel from which the data can be read, the execute header information and some other attributes necessary to the UNIX exec logic.

There are several flavors of UNIX executable binaries. These include (1) non-reentrant (non-shared), non-write-protected, fully loaded, (2) reentrant, write protected, and (3) reentrant, write protected and demand paged. It is assumed that the overhead of synchronously reading data from an object process is too great to support demand paging. Further, as a

temporary restriction of this implementation, shared text loading is not done. This is due only to the implementational complexities in replacing the text structure's inode reference with a reference to a RPC caller file structure. As a result of these two restrictions, all object exec are internally considered to be non-reentrant, non-shared, non-write protected and fully loaded at exec time (see section 9.2 for a discussion of extensions).

The object exec operation is also used to debug the object process using a breakpoint debugger. As mentioned earlier, an object will be traced if all of the following are true.

(1) A non-suppressed object exec of the object program's path is executed.

(2) The process executing the exec is has the trace flag (STRC) set.

(3) The UID of the process executing the exec is the UID of the object directory.

(4) The object is passive at the time of the exec.

3.7 Partially indirect operations

As noted earlier, the object process implementor has a choice to make for those operations which return channels. He can decide (a) to give the client unrestricted access to an existing channel or (b) to involve the object process in each and every operation which may arise via the returned RPC caller channel connecting the client and object processes. This choice is too limiting. To avoid this problem, a third choice is provided.

Partially indirect channels provide a mechanism for the choice between direct versus indirect operations to be made on an operation-by-operation basis. For example, the object process can decide at the time the channel is returned to the client that all read operations are to be direct and apply to a pipe however all ioctl operations are to be indirect (via a particular RPC conduit) and the object process is to simulate these functions.

Partially indirect operations are set up using the split(2) system call which combines two channels and a mask into a new channel. The mask is considered a bit vector which specifies, for each operation, whether it is to be done on the first channel (if the bit is 0) or the second channel (if the bit is 1). Also note that the split operation can be applied repeatedly to form a binary tree of channels.

The split operation is useful outside of the context of object files. As an example, suppose that one wishes to make a full duplex buffered data path out of two pipes (one in each direction). The split mechanism can be used to separate the read from the writes in this case.

Note that the split channel is completely invisible since all operations are directed either to the left or right subtree. Also note that the mask chosen is static. One cannot alter the mask after the fact. Perhaps a mechanism to do this should be provided?

3.8 The object select operation

What does it mean to ask whether a caller RPC conduit is readable (without delay), writable (without delay) or whether an exception is pending? This is exactly the question which must be answered if an object select operation is to make sense. The point of view taken in the design of object files is that the read operation handling routine in an object program can be considered a subprocedure which is invoked when an object read operation occurs. The fact that the read routine exists in another process is only a technical difficulty.

The algorithm used in the UNIX kernel is to test all channels specified by the select masks for the ability to execute the requested non-blocking operation. This test is done by calling a channel dependent routine within the kernel which returns a Boolean value for that channel. If none of the channels can execute the non-blocking operation specified, select will sleep until a change in state on one of the channels is indicated by a select wakeup operation. The channel specific code must retain the information that the process executing the select should be notified when the appropriate change of state occurs. By analogy and according to design goal 6, the select operation with the read bit set for an object file in which read is an indirect operation should invoke a routine in the object process to answer the question: is this channel readable without delay?

This strategy means that a RPC must be issued on the read RPC conduit to ask this question. Unfortunately that the select operation will block, at least long enough to RPC the object process to test readability. The second half of the implementation of the select algorithm is a mechanism for the object process to notify the client process who may be sleeping in a select, that data has become available to satisfy the read. This mechanism is available through the RIOCASYNC (see section 4.7) ioctl call.

An additional point which must be clarified is what happens when a select operation occurs on a split channel. Inside the kernel, separate selscan tests occur to determine whether each channel (according the descriptor masks) is readable, writable or has an exception pending. If the channel is split, the read test is made on the leaf node from which data would be read. The same procedure is done for write. The exception test is routed through the split channel tree according to the selscan operation bit in the mask of each split file table entry.

3.9 The two pathname operations

There are several operations which are parameterized by two pathnames: mount, link and rename. In each of these operations, the first pathname parameter is always traversed first by the kernel. If this pathname results in an object operation, then the mount1, link1 or rename1 SCP operation is said to occur. In this case the second pathname is passed to the object process as a simple string parameter. Similarly, if the first pathname does not result in invoking an object and the second pathname results in an object operation, then the mount2, link2 or rename2 operation is used and the first pathname is passed in as a string.

In addition to passing the pathname which did not cause invocation of the object, the starting directory must be communicated to the object process. This starting directory is communicated as a channel. The object process should use the descriptor associated with this directory to form a "@<*fid*>/" prefix for the pathname before it is traversed. Note that this descriptor can be either a primitive UNIX directory or an RPC conduit indicating that the associated directory was inside an object (possibly the same object).

The problems with the two pathname operations occur when the first pathname causes invocation. In this case the second pathname can (1) refer to a primitive UNIX object (i.e. a non-object file), (2) reference an entity within the same object invoked by the first pathname, and (3) invoke yet another object. There is no procedure for handling any instances of the third case. The object-to-object protocol required here goes beyond the scope of SCP. Even if the object-to-object protocol exists, the problems of representing inter-object hard links have not been solved in an implementationally acceptable way. Clearly more work is required here.

The sub-problem of attempting to distinguish among the three cases above is not easy. At this time the only assistance provided by the kernel is to test for self-intersection (and

returns the pathname suffix if a self-intersection is detected). Unfortunately, the self-intersection test does not handle *pass-through* object operations. A pass-through operation occurs when an operation parameterized by a pathname invokes (either an intersecting operation or the path can originate within the object) an object, but later leaves that object via an out-link (hard or symbolic). A *symbolic out-link* (by analogy to UNIX symbolic links) is a reference to the target entity by pathname. A *hard out-link* is direct reference which is static (recorded on disk in such a way that it survives re-activations of the objects involved and reboots of the machines involved). Pass-through operations result in nested (see section 4.2) object invocation or require the object process to act as intermediary interpreter for the kernel (in the case of a pass-through primitive object operation). All remote operations (both object and primitive) can be considered pass-through operations where the object process being passed through is the local file server (see chapter 5).

3.10 The SCP operations

The SCP operations are the UNIX I/O operations which are parameterized by channels or pathnames. Each SCP operation is mapped to a RPC (the exec and select operation may require more than one RPC) to communicate this operation and its parameters to the object process. There are several points to note:

(1) The object process is responsible for all semantics of the operation. The SCP interface will not attempt to filter out any invalid or illegal operations. For example, linking and unlinking directories and changing the ownership of an entity are not rejected for object files if the client is not the super user.

(2) The mode parameter in the open, mkdir and mknod operations passed in the RPC request has been modified according to the client's umask value.

(3) The size of data buffer parameters must be inferred from the size/type tag of the parameters (see section 4.1). This is true for reply parameters too, where size/type tag of the reply parameter is obtained from request parameter 0 (see section 4.5).

(4) The pathname suffix never begins with a slash and is always null terminated.

(5) The descriptor management functions of dup, fcntl operations F_GETFD and F_SETFD and the ioctl operations FIOCLEX and FIONCLEX do not result in object operations.

The same is true for the implied versions of these operations which occur in the fork and vfork operations.

(6) The close operation and its implied forms in exit and exec do not result in an RPC operation. Instead, the last close of a RPC conduit is communicated implicitly by the zero return of the read operations on the part of the object process (see section 4.9).

Table 1 shows the SCP operations with their parameters. The type (see section 4.1) follows the parameter in parentheses. The type codes are as follows:

1,2,4	A data structure of size 1, 2, or 4 bytes
i	An identification parameter
b	A data buffer of indefinite size
c	A channel parameter
s	A fixed size data structure (obvious by context)
S	A (n)ioctl data structure sized from command word
F	An array of descriptors

The SCP implementation is a separate layer of software which interfaces the RPC communication protocol to the system call service routines in the client and object processes. On the client side, this interface resides in the kernel (see figure 5) in order to provide a transparent system call interface for all existing software. On the object process (callee) side, the SCP service layer is placed in the user mode address space of the object process.

Operation	Request	Reply
open	1(1), I(i), suf(b), flag(2), mode(2)	chan(c)
exec	2(1), I(i), suf(b)	hdr(s), chan(c), len(4), flag(2) [, I(i)]
chmod	3(1), I(i), suf(b), mode(2)	
chown	4(1), I(i), suf(b), uid(2), gid(2)	
stat	5(1), I(i), suf(b)	statbuf(s)
lstat	6(1), I(i), suf(b)	statbuf(s)
utime	7(1), I(i), suf(b), time[2](s)	
truncate	8(1), I(i), suf(b), pos(4)	
mkdir	9(1), I(i), suf(b), mode(2)	
rmdir	10(1), I(i), suf(b)	
mount1	11(1), I(i), suf(b), dir(c), name(b), flag	
mount2	12(1), I(i), suf(b), dir(c), name(b), flag	
umount	13(1), I(i), suf(b)	
access	14(1), I(i), suf(b), flag(2)	
chdir	15(1), I(i), suf(b)	chan(c)
readlink	16(1), I(i), suf(b)	buf(b)
symlink	17(1), I(i), suf(b), buf(b)	
rename1	18(1), I(i), suf(b), dir(c), name(b)	
rename2	19(1), I(i), suf(b), dir(c), name(b)	
link1	20(1), I(i), suf(b), dir(c), name(b)	
link2	21(1), I(i), suf(b), dir(c), name(b)	
unlink	22(1), I(i), suf(b)	
mknod	23(1), I(i), suf(b), mode(2), dev(2)	
nioctl	25(1), I(i), suf(b), cmd(4) [, datain(S)]	[dataout(S)]
read	33(1), I(i)	buf(b)
write	34(1), I(i), buf(b)	count(2)
fchmod	35(1), I(i), mode(2)	
fchown	36(1), I(i), uid(2), gid(2)	
ftruncate	37(1), I(i), pos(4)	
flock	38(1), I(i), flag(2)	
fsync	39(1), I(i)	
ioctl	40(1), I(i), cmd(4) [, datain(S)]	[dataout(S)]
selscan	41(1), I(i), flag(2)	flag(2)
lseek	42(1), I(i), pos(4), flag(2)	pos(4)
fstat	43(1), I(i)	statbuf(s)
fcntl	44(1), I(i) cmd(4) [, datain(S)]	[dataout(S)]
connect	45(1), I(i), buf(b)	
accept	46(1), I(i)	buf(b)
send	47(1), I(i), buf(b), to(b), rts(F), flag(2)	count(2)
recv	48(1), I(i), flag(2)	buf(b), from(b), rts(F)
bind	49(1), I(i), buf(b)	
setsockopt	50(1), I(i),	level(4), opt(4), buf(b)
listen	51(1), I(i), count(2)	
getsockopt	52(1), I(i), level(4), opt(4), buf(b)	
shutdown	53(1), I(i), flag(2)	
getpeername	54(1), I(i)	buf(b)
getsockname	55(1), I(i)	buf(b)

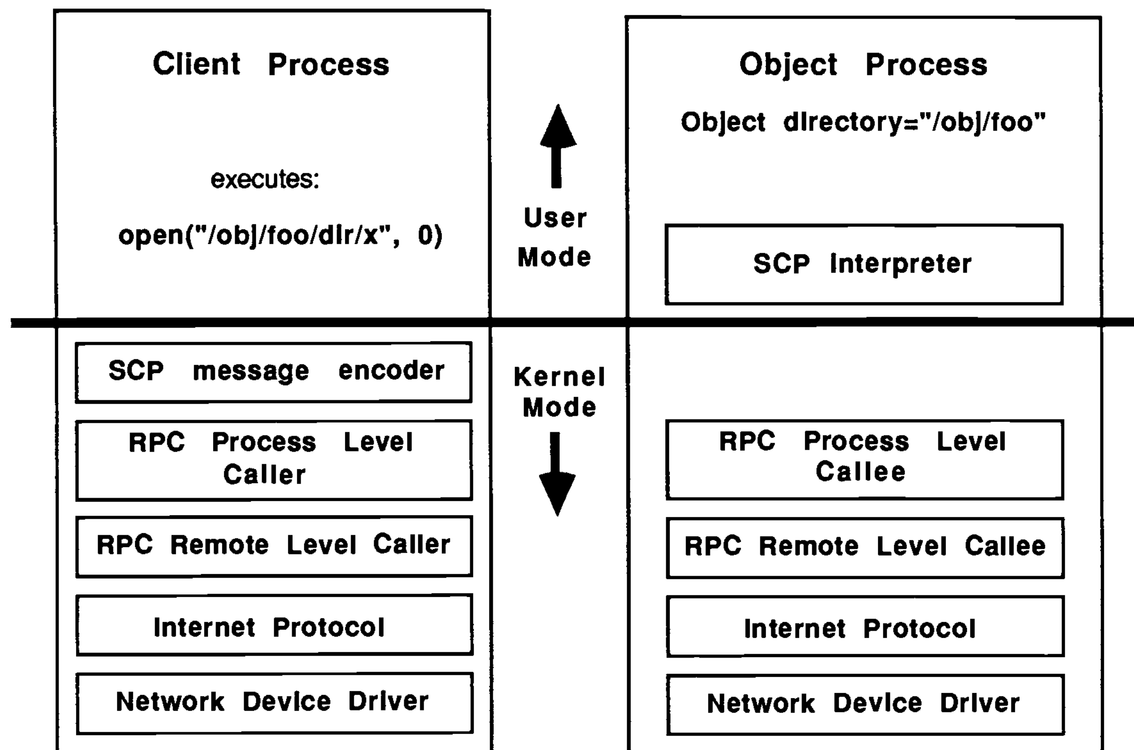

Figure 5: User and kernel states

3.11 Extensibility

The SCP provides the basis for a collection of common operations which can be used to manipulate object files. This common interface is one of the distinguishing features of object oriented software. Equally important, is the ability to define new interface operations as needed.

Historically, extra I/O operations (especially those which are specific to a certain type of I/O mechanism) have been encoded as ioctl operations in UNIX. The ioctl interface is used as the first mechanism of extension. In the ioctl analogy, a command word is formed which is an encoding of (1) the size of a data structure used to parameterize the operations, (2) flags indicating that the data structure is to be as an input or an output (or both) of the operation and (3) a operation specific 16 bit code. An object can define an arbitrary number of new ioctl operations.

The second observation is that object operation come in two types: those which invoke the object by means of a channel and those that invoke the object by means of a pathname. The pathname operations generally represent self-contained functions (similar to datagrams), where the channel operations can be considered part of a sequence of operations perhaps comprising a larger single entity such as a transaction (by analogy this is similar to a virtual circuit model). The ioctl interface is strictly a channel based extension. Pathname based extensions are formed using the nioctl interface. Nioctl is a new system call which is identical to ioctl (the same encoding of the command word is used and the same restrictions on the parameter data structure apply) except it takes a pathname argument in place of the descriptor.

The (n)ioctl based extensions are not completely general. They have the following restrictions: (1) The parameter data structure is required to reside in contiguous locations in memory (this restriction is due to the fact that the RPC mechanism does not know where the pointers to related data structures contained in the parameter are). Some of the existing ioctl parameterization schemes are handled by special case code in the kernel based on the command word). (2) There is an implementational size limit on the parameter data structure of 128 bytes (This is due to the way ioctls are implemented in 4.2 BSD UNIX). (3) Non-data parameter types cannot be handled (channel parameters in particular). An example of an operation which cannot be represented by an ordinary ioctl is one which takes 3 channels as arguments.

To overcome these restrictions, a way to access the raw RPC call interface is provided. The RPC call mechanism is itself a ioctl (symbolically RIOCCALL) where the parameter data structure is a vector of parameters to the RPC call interface. The facilities available at the level are described in the next section. It is general enough to allow over 800Kb of data and 20 open channels to be passed in a single operation.

Chapter 4

The remote procedure call mechanism

All object client communication occurs via the RPC protocol. This protocol provides the following major features:

1) Type checking between caller and callee

2) Channel parameters

3) Secure communication of process identification data

4) Caller exception processing integrated into the RPC mechanism

5) A uniform error recovery mechanism

6) The ability to bind object directories to RPC conduits

There have been several RPC implementations based on the installation of the RPC code in a user process. This method is not suitable for object files because it would be impossible to implement points 2, 3 and 6 from the UNIX user level. Further, such an implementation would require re-linking of all software intended to make use of the object file interface with an alternative library. The net effect would be to duplicate this code in the memory images of all processes, which is undesirable.

An RPC transaction, as described earlier, consists of a *request message* from the *caller* process to the *callee* process, followed by a *reply message* in the reverse direction. Both the request and reply messages are reliably delivered datagrams (see figure 6). The RPC mechanism buffers both requests and replies such that a multi-client (caller), multi-server (callee) queue is implemented. Once a request from a caller is accepted by a callee, the two processes are bound to each other such that the only process which can respond to the request (regardless of the actions of any other processes which access to the RPC conduit) is the process which received the read request.

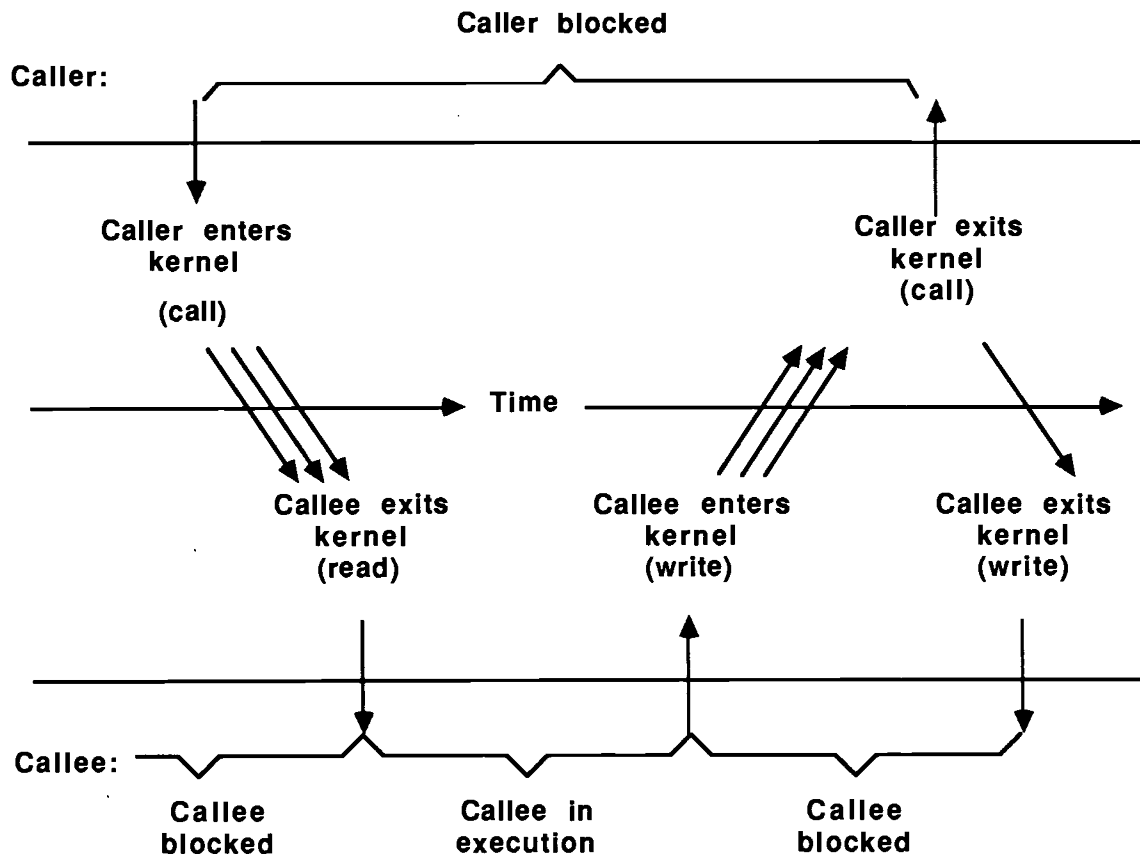

Figure 6: A RPC protocol

An important consequence of the caller process to callee process binding is that a callee cannot read a request and then create a child process (i.e. fork) to allow the child to complete the service of this request (i.e. to send the reply). It is possible for the child to monitor the RPC conduit while the parent assumes the task of servicing the RPC.

The multi-server, multi-client queue is implemented by connecting all RPCs to the RPC socket (the per-RPC-conduit data structure) in a double linked list. Each RPC is a sequence of buffers use to hold the request or reply message fragments during the different states of the RPC. The RPC socket also contains bidirectional references to the file structures used by the caller and callee channels plus the bound (object directory) inode. This is shown in figure 7. Each RPC header contains a pointer the callee's process table entry and each RPC in the invoked state is a member of a chain of RPCs from the callee's user structure.

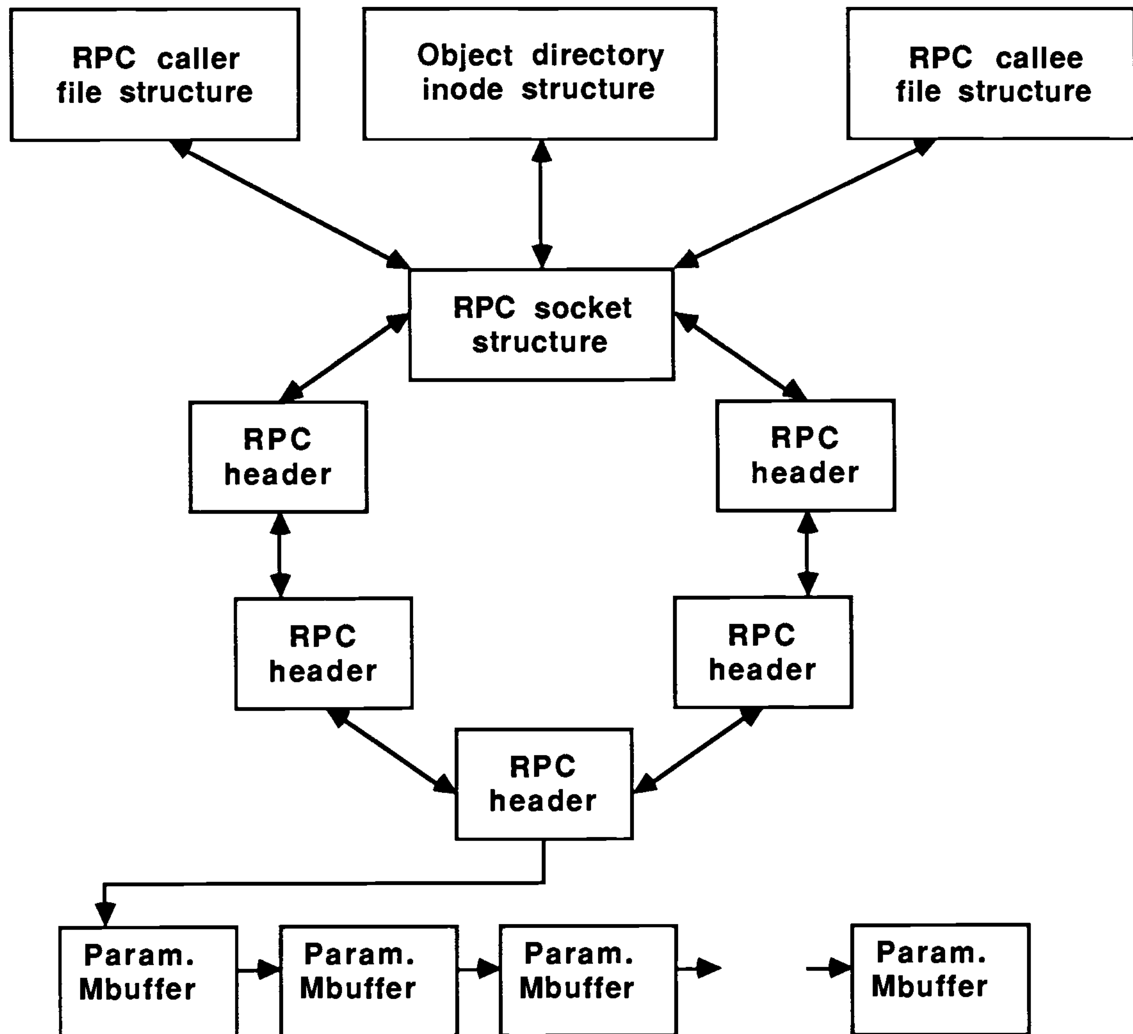

Figure 7: The local RPC data structures

The call, read (the action of the callee which receives the request) and write (the action of the callee which sends the reply) are all atomic. This means that the caller and callee cannot be in user mode in the position of having partially complete messages transferred. In other words, the call, read, and write system calls must transfer the entire message and receiving operations must be prepared to accept the maximum sized messages. The call operation cannot be interrupted between the sending of the request and receiving of the reply (see exception handling). Further the callee must produce a reply to any request which has been read before the next read can be executed.

A single RPC conduit consists of a caller channel connection and a callee channel connection. The only valid RPC operations on a caller channel are call (which is implemented as the RIOCCALL (n)ioctl operation) and close. The SCP interface will generate a SCP remote procedure call if any other operation is attempted. This means that the caller channel is used as is for all operations required to invoke the object process.

The callee can read request messages with either the read or readv operations and replies are produced by write or writev operations. The term read (write) will be use to refer to either the read (write) or readv (writev) operation. An error condition will occur if the read – then – write protocol is not followed by the callee.

There are times when the read – then – write protocol is not desired. In this case the callee can move the current RPC (an RPC whose request has been read but no reply has been written, such a RPC is in the *invoked* state) to the background. When the RPC is placed in the *background* the callee can executed a new read on the RPC conduit. When a reply is to be returned to a background RPC, it must first be moved to the *foreground.* A RPC can only be returned to the foreground when no existing RPC is in the invoked state. The RIOCBG and RIOCFW ioctl operations on the callee channel move the invoked RPC to and from the background. Note that several RPCs can be in the background on any given conduit so an index is returned by RIOCBG to identify the particular RPC. The number of RPCs which can be placed in the background is limited by the number of RPCs which can be associated with a single RPC conduit.

4.1 Parameter types and type checking

The request and reply messages each consist of a sequence of parameters. Each parameter has a type and a size which must be compatible between sender and receiver. By analogy with parameters to ordinary procedures, the sender's parameters will be called *actual* and the receiver's parameters are called *formal.* There are five types: data buffer, data structure, channel buffer, channel structure and identification parameters.

A *data buffer* parameter is a block of data with an indefinite size. Corresponding data buffer parameters clash if the receiver's buffer size is too small to hold the entire actual parameter. The current maximum size of a data buffer parameter is 52224 bytes.

A *data structure* parameter is a block of data with a fixed size. Data buffer and data structure actual parameters are interchangeable since the size of a particular actual parameter is always known when a message is assembled. A formal data structure parameter will not

clash with an actual data (structure or buffer) parameter if the parameter sizes are the same. A formal data buffer parameter will match actual data parameters whose sizes are less than or equal to the size of the formal parameter. The maximum size of a data structure parameter is 12800 bytes.

A *channel buffer* parameter is a sequence of zero or more channels of indefinite length. Open channel parameters are represented by their descriptors stored in consecutive bytes. The maximum number of channel parameters passed (in all channel parameters) is limited to the maximum number of open channels per process. In order to accept a message, all channel parameters must be able to be received by the receiving process without overflowing that process' open channel table. The two high order bits in each descriptor byte indicate whether the (1) the controlling terminal attribute and (2) a unique process group assignment are to be passed along with the channel itself. The controlling terminal and/or process group attributes can only be passed from caller to callee.

A *channel structure* parameter is a sequence of channels with a fixed size. The conditions for accepting channel parameters (both buffer and structure) are the same as the rules for data parameters. Note that all channel parameters are essentially passed by reference as opposed to data parameters which are passed by value.

An *identification* parameter is a data structure consisting of the real and effective user and group identifications plus the vector of groups to which the sender belongs. Normally this parameter is defined by the kernel on behalf of the sender. If the sender is the super user, this parameter can be explicitly defined. Identification parameters can only be accepted by a formal parameter whose type is also identification.

To escape from the inflexibility of strong typing, an untyped formal parameter is also provided. A formal untyped parameter is simply a buffer which will accept parameters regardless of type, size and number of parameters. The parameters are stored into the buffer in address order separated by the *size/type tags*. Actual untyped parameters are stored in the same format. The size/type tag is a two byte integer which is an encoding of the size and type of each parameter. Actual parameters can be specified in a mixture of the typed and untyped forms. Formal parameters can also used both representations, however all typed parameters must occur at the beginning of the formal parameter list followed by the the untyped parameters.

Normally the scatter – gather vectors of the readv / writev operations by the callee are used to specify the individual parameters with the type and other control information encoded into the high order bits of the length field. The parameters of a RIOCCALL (n)ioctl

operation are also specified by an array addresses and lengths where the request actual parameters are followed by the reply formal parameters. In the case of a receive operation, the length fields initially specify the maximum size of the formal parameter. When the receive operation is complete, these fields have been modified to contain the exact size of the particular actual parameter.

If the read and write system calls are used (there are no scatter – gather vectors), then the indicated buffers are assumed to contain untyped parameters.

When a read(v) or write(v) operation is used by the caller, it returns the number of parameters transferred in place of the number of bytes. If the FIONREAD operation is done by the callee, it will return the number of queued RPCs as opposed to the number of bytes available.

4.2 Structures of intersecting processes

Remote procedures can be nested. A nested remote procedure call occurs when the callee executes a call operation between the read and write operations. This is exactly analogous to procedures in a conventional programming language. Also note that for each dynamic level of remote procedure there exists a blocked process. To avoid excessive consumption of system resources, an arbitrary limit of 8 dynamic levels is imposed.

Recursion can also be implemented by directly or indirectly invoking oneself. Because the caller is always blocked during the servicing of a remote procedure call, recursion requires multiple servers. Deadlock will occur if the depth of recursion exceeds the number of server processes. This deadlock can be cleaned up by sending a fatal signal to the deepest server (see section 4.4). There is one way to avoid this type of deadlock: create a new server process before a potentially recursive call operation is executed. In the context of object files, a special operation is provided to test a pathname for self invocation.

Interprocess coroutines can also be implemented using the RPC mechanism. To set up coroutines, the participating process to be scheduled must execute RPC calls to a common callee (the coroutine scheduler). This can be done using distinct RPC channels or using the same RPC conduit where the inactive coroutines have been pushed into the background. When the coroutine scheduler process wants to resume a particular coroutine, it will just return from the appropriate RPC. If an RPC channel is shared among more than one coroutine, this may require pushing the current RPC into the background and popping the arising coroutine into the foreground.

4.3 Error notification

Error conditions and other abnormalities during the RPC process are communicated to both the caller and callee as system call error codes. Error conditions which occur during the transfer of the request message will cause the entire RPC to be aborted without informing the object process. Errors which occur during the communication of the reply message will return errors to both the caller process and the callee process. If the callee process' write reply operation completes without error, it can infer that the reply was successfully delivered to the caller.

4.4 Caller exception handling

The RPC call operation (regardless of whether it is done explicitly using the ioctl interface or implicitly by the SCP level) is always blocking and uninterruptible. If the caller receives a signal while a call is in progress, the associated exception handling is under the control of the RPC protocol.

First, signals which are masked or ignored are invisible to the RPC mechanism. The masked signal action will occur whenever it is unmasked which cannot occur during the call.

Second, signals which suspend the caller take place only after the RPC has been completed, however the SIGCHLD notification must occur immediately. In this scenario, the idea is to provide immediate notification to the parent (this means that shell job control will not be delayed by the RPC mechanism) that the suspending signal has been received without actually suspending the caller until the RPC is complete. If the caller were suspended in the midst of an RPC transaction, the callee would also be blocked since all transfers of data require an acknowledgement handshake between the two processes. If the callee became blocked, so would all other processes which are in the process of being serviced or are soliciting service from this callee.

The third case is that the signal is fatal to the caller. The caller always dies immediately (in the inter-machine case, an acknowledge must be received from the remote machine) after notifying the callee, regardless of the state of the RPC. If the callee has not yet read the complete request message, the request is discarded without informing the callee. If the callee has read the request, but not yet written the reply it will be informed by a SIGPIPE signal. The RPC protocol requires that the callee write a reply in this case even though the caller is dead (parameter 0 is all that is needed). If a callee write has been executed, the remaining fragments in the reply are discarded by the RPC protocol.

The fourth case deals with signals which are explicitly handled by the service routines in the caller. If the entire request has not been read by a callee, the entire request is discarded and the signal service occurs in the caller followed by re-executing the RPC (if SOUSIG is clear). The caller is unaware that a request was discarded in this event. If the caller has written the reply, signal service will occur after the RPC is complete. If the caller has read the request but has not yet produced a reply, the callee is informed by SIGURG that the caller has a pending non-fatal signal. The callee can abort the current RPC to give immediate signal service (see reply parameter 0) or ignore the signal and complete the RPC before signal service occurs.

The callee can explicitly test the signal state of the caller by using the RIOCSIG ioctl operation. If several RPC conduits serviced by the same callee, the callee would test each channel to determine which RPC was responsible for the signal. The RIOCSIG operation returns two bit masks (one for SIGURG, one for SIGPIPE) which indicate those RPCs by RIOCBG index which have pending signal conditions.

4.5 Request and reply parameter 0

The callee will receive an extra parameter as the first actual parameter. This parameter is provided by the RPC protocol to communicate certain control information to the caller. This information includes:

(1) The maximum number of open channel parameters which can be accepted by the caller. This number is the difference between the current number of channels the caller has open and the maximum per-process limit.

(2) The dynamic level of this RPC. The dynamic level is computed by the RPC protocol at the time a call is executed as the maximum of all the dynamic levels of all calls be serviced by the caller plus one. This computation is also used to reject the call operation if the nesting level becomes too deep.

(3) A remote flag which indicates that the caller is executing on a different host.

(4) A flag indicating that the caller can accept a controlling terminal assignment (see section 4.7).

(5) A flag indicating that the callee can accept a unique process group assignment (see section 4.7).

(6) A flag indicating that a byte swap within short is required (refer to section 5.8).

(7) A flag indicating that a short swap within long is required.

(8) The local file server target descriptor (described in section 5.4).

(9) A list of reply parameter tags. These two-byte integers indicate the size and type of the formal parameters expected in the reply and an error will occur if the callee attempts to return a reply which clashes with this list. The last element in the list is the number of bytes of untyped parameters which can be accepted. They always follow the typed parameters.

The callee must explicitly write one extra parameter in the reply. This parameter is absorbed by the RPC protocol and is not explicitly passed to the caller. Also note that a reply message consisting of only parameter 0 will never be rejected as a reply message type clash. Reply parameter 0 specifies:

(1) The system call error code (one byte).

(2) A reply synchronous signal number (such as SIGPIPE).

(3) A flag indicating that the RPC should be re-executed following signal service. This require that the AT&T compatible signal mode (SOUSIG) not be used.

(4) A flag indicating that the synchronous signal is to be sent to the process group associated with the caller RPC conduit as opposed to the caller process itself.

Note that neither request parameter 0 nor reply parameter 0 are explicitly stated in the table of SCP operations, however it is understood that they exist.

4.6 Size limits and fragmentation:

The size type tag is a short integer which must encode all type and size information. Each of the five types is encoded as an interval, which together partition the interval 0 to 65536. The largest of the type intervals is the data buffer type which occupies more that 52000 bytes. The reason for this is that SCP encodes the entire read / write buffer as a single RPC parameter. Limits on the size of an untyped parameter restrict it to 65535 bytes. There is also an implementational limit of 16 parameters per request or reply. This means that the maximum size of a complete request or reply parameter exceeds 800Kb. This size limit means that it is not practical for the kernel to attempt to buffer complete RPC messages.

Also if the RPC is to be communicated by a network, there is usually a smaller limit on the maximum size of a transmission unit. For ethernet[1] this limit is 1500 bytes.

To accommodate these restrictions, the complete RPC message may be broken into pieces which are more convenient for buffering and transmission. These pieces are called *fragments* (these fragments are not the same as IP fragments). At this time the receive window is exactly one fragment long. Except for large reads and writes, all SCP messages fall into a single fragment.

4.7 Asynchronous signals, process groups and controlling terminals

All of the semantics for dealing with UNIX terminal control are provided through the RPC parameter passing mechanism. The open system call will define (1) a controlling terminal and (2) a process group if the process executing the open does not have these attributes. If an object open operation is to provide these attributes, the callee must be informed that the caller does not have a controlling terminal or a process group. This is done via request parameter 0. If the callee wants to simulate a controlling terminal and define the a new process group which can receive the terminal related signals, the descriptor byte referring to the channel to be returned to the caller should contains the high bit(s) set. The bits independently specify (1) whether the controlling terminal attribute of the client is to be set to this channel and (2) whether the process group of the client is to be defined (as its process ID) and that the same process group to be assigned to the channel. The process group is used to specify the collection of processes which receive the I/O signal associated with the channel and other process-related control functions such as who can do I/O on the channel without being suspended. The controlling terminal definition and initial process group assignment are the only asymmetric functions in the open channel parameter passing.

If a channel parameter is specified to define the controlling terminal for a process, then it will receive all /dev/tty and kernel uprintf() I/O. If the channel returned to the caller as a controlling terminal is a RPC caller channel, then the operations done by uprintf()s and /dev/tty result in SCP RPC on this channel. Note that the channel returned as a control terminal is not restricted to be a either a local terminal (or pseudo terminal) or a caller RPC channel (a regular file could be used to log uprintf()s for example). The uprintf() mechanism is implemented with a queue of write messages (since uprintf() can be called at high interrupt priorities) to delay the RPC call until the processor priority is reduced. This queue is serviced by the local file server processes.

[1] Ethernet is a trademark of Xerox.

The RIOCASYNC ioctl can be use by the callee to send signal and select wakeups to the caller's process group via an RPC conduit even when no RPC is invoked. The signals are sent to the process group of associated with the caller connection to the RPC conduit. The signals are restricted to SIGHUP, SIGINT, SIGQUIT, SIGTSTP, SIGTTIN, SIGTTOU, SIGIO and SIGURG. Note that there are two process groups associated with a RPC conduit. The callee's process group is used for routing SIGURG, SIGPIPE and SIGIO to the callee process(es).

4.8 Inode binding

Inode binding is the mechanism by which a RPC caller channel is associated with an object directory inode. When a RPC conduit is bound to an object directory, the RPC conduit will be used to communicate the intersecting SCP (and nioctl RIOCCALLs) operations with respect to the bound directory. The lack of binding on a object directory is the internal indication that the object must be activated. This activation includes the creation of the object process as well as the creation of the RPC conduit which connects the object process' intersecting descriptor to the object directory. Inode binding can also be done explicitly using the RIOCBIND ioctl.

The inode is unbound when both the .i_nlink and .i_count reference counts are zero at the same time, or when the last close on the callee channel is executed, or by the RIOCBIND ioctl with a null pathname (note that the binding does not require an .i_count).

4.9 Closing RPC channels

When the last close on a (unbound) caller channel occurs, the corresponding callee read will return 0 (just like a pipe). It is impossible that any RPCs are buffered at this time since the caller must be blocked during the call operation (which requires an open caller channel). This close is non-blocking for a local RPC conduit and it requires only a single acknowledgement from the remote machine in the non-local case. If the RPC conduit is bound, the binding is considered to be a caller connection which is not closed until an unbinding event occurs.

When the last close on the callee channel occurs, buffered RPCs can exist in one of three states: (1) RPCs whose requests have not been read, (2) RPCs whose request has been read but no reply has been written, and (3) RPCs whose replies have been written but not yet returned to the caller. All RPCs in category (3) are allowed to complete normally. The other

categories get error returns which depend on their category. The last close of the callee channel will block until the caller processes of all buffered RPCs can be notified and the proper deallocation procedures performed.

SCP checks for category (1) errors on intersecting operations which result in a new object process being created, followed by re-execution of the SCP call.

4.10 The process level state machine

The sequence of actions done by the RPC mechanism is controlled by a finite state machine. The state transition diagrams for these machines are shown in figure 8 (for the caller) and figure 9 (for the callee). There is actually only a single state machine, where transitions are performed alternately by the caller and callee processes, where the essential synchronization is obtained by the waiting process sleeping until a particular state appears. The states are:

SRQ is an acronym for sender's request. The caller will wait for this state to appear before it will produce another request fragment.

RRQ (receiver's request) is the state in which the callee waits for subsequent fragments.

INVKD is the invoked state when no signals have been received.

NFS is an auxiliary invoked state use to hold the information that a non-fatal signal has been received.

SRP (sender's reply) is the state in which the callee waits for the caller to set to indicate that the reply fragment has been consumed.

RRP (receiver's reply) is the state in which the caller waits for the callee to set to indicate that the reply fragment has been produced.

ABT (abort) is the state that the caller set when the caller detects an error condition which is to terminate the RPC. If the callee observes this state, it should abort the RPC. If RPC has not entered the invoked state yet, the abortion is invisible to the callee process. If the callee is between the read and write operation, the SIGPIPE signal is used to notify the process level of the abortion. If the RPC is in the process of returning the reply message, the callee's write is ended with an error code. The callee is responsible for cleanup in this case.

ERR (error) is the state set by the callee to communicate an abnormal end to the RPC process to the caller. The error code communicated by this message is always returned as the result of the call operation. The caller is responsible for cleanup in this case.

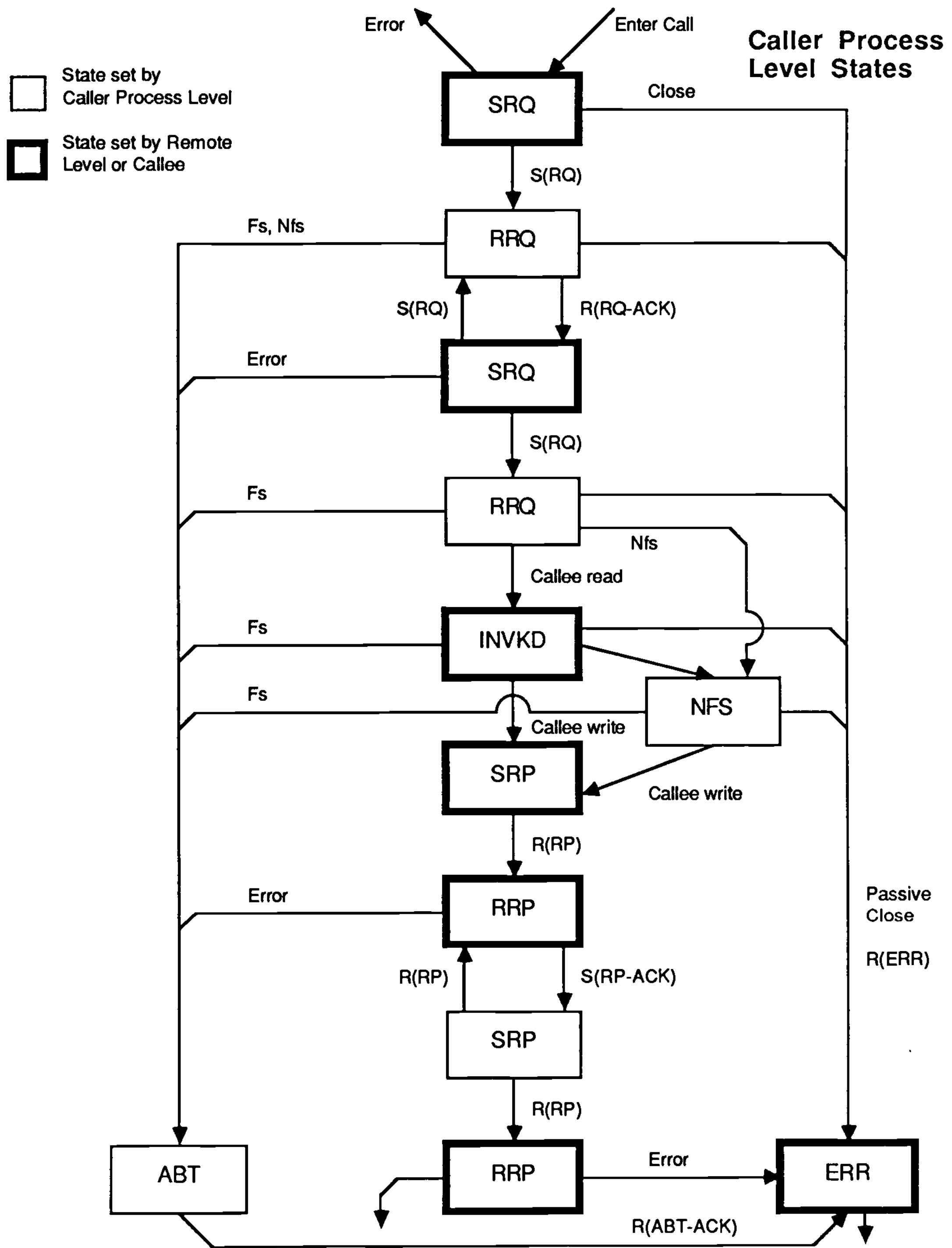

Figure 8: Caller's process level state machine

Figure 9: Callee's process level state machine

Chapter 5

Network RPC communication

The RPC implementation is divided into two parts: the process level and the remote level. The process level runs (as system call routines) on the kernel stacks of the various caller and callee processes. Also it will completely implement RPCs which are local to a single host.

The remote level deals with the intermachine communication issues. Loosely speaking, it simulates the remote process level on the local machine. Implementationally, it runs out of interrupt and timer routines which operate below the process level, in addition to explicit calls from the process level. Each RPC is considered to be a finite state machine which executes non-blocking transitions in response to events such as receive packet, timeout, and process level produced fragment.

The communication medium is assumed to be an unreliable datagram channel. The current RPC mechanism for network communication is implemented as a new internet protocol (IP) [NIC82]. In this scheme, the IP route is allocated only once when the host-to-host connection is established. Out-of-order packet delivery is avoided by use of sequence numbers on a per-remote-host basis. The only IP services being used are routing and checksum validation, since the packet size (RPC fragment size) is chosen such that the IP fragment assembly mechanism is not used. This means that only a small change would be required to use raw ethernet as the transport medium.

An alternative strategy would be to use a single virtual circuit connection between hosts to service all RPC traffic between those two machines. This mechanism has the advantage that a sliding window protocol is used to improve the performance of multi-fragment RPCs. Since the end-to-end connections are already reliable, all of the sequence numbering, re-transmissions, and timeouts could be avoided. The disadvantages of this method are that it is unable to take advantage of the natural boundaries of the RPC messages for packetizing and re-transmission plus at least one (perhaps two) extra levels of data copying is placed in the critical path on both the caller and callee machines. It is claimed in [Birr84] that the simple datagram based transport layer is more efficient for RPC communication.

5.1 Establishing a host-to-host connection

The RIOCREMOTE ioctl operation is used to establish a single symmetric pair of RPC channels between two hosts. To do this, processes on the two machines must concurrently execute RIOCREMOTEs which specify the host addresses of the opposite machine. This operation will block (signal interruptible) until the rendezvous between the RIOCREMOTEs is complete. The rendezvous will (1) guarantee that both machines have exactly the two primary RPC conduits established between them (i.e. all previous connections have been cleaned up), (2) initially synchronize packet sequence numbers, and (3) negotiate timeout and passive close intervals.

One problem with this method of establishing communication is that a process will be required to wait for a connection (ie have a RIOCREMOTE in execution) for each host. This requires an excessive number of processes. To avoid this situation, a connection server facility is used. This system call will return the host address and connection parameters whenever (1) a request to establish a connection messages is received (these messages are periodically sent by pending RIOCREMOTEs) or (2) an attempt to forward a channel to a host without a connection is received. Connections will be shutdown when the connection has remained idle for a certain interval and the number of host-to-host connections is above a certain threshold. This technique permits the network of RPC channels to be considered to be logically transitive without multi-hop messages (unless required by the underlying IP network) and without maintaining a complete connection graph.

The RIOCREMOTE operation takes 3 descriptors as arguments (in addition to the communication parameters and the address of the remote machine): (1) the primary caller (must be an RPC conduit), (2) the primary callee and (3) an RPC conduit to a local file server (LFS) process. The local primary caller is connected to the remote primary callee and vise-versa. The local file server is an optional parameter which is used only if non-RPC conduits are passed between the two machines or if the primary callee is not a RPC conduit. Its purpose is to execute the SCP operations required by a remote client process manipulating a target channel which is not a RPC caller channel. The interface and function of the LFS process(es) will be discussed in more detail later.

5.2 Distributed file system via RPCs

Since only one RIOCREMOTE connection between two machines is permitted, the top level (primary) RPC conduits are normally used to implement a facility which can be used to allocate additional RPC conduits between the two machines. The additional RPC conduits

are created by passing open channel parameters between the two machines. In this way an arbitrary number of RPC conduits can be supported. A distributed file system based on the remote mount concept is a convenient way of doing this. The term distributed file system in this context means only remote file access and manipulation. More sophisticated techniques such as storing of multiple copies of a file and transaction based processing (via a commit / abort protocol) are not part of this model, however the RPC / object file mechanism provides a means by which such a mechanism can be constructed.

The RPC conduits described above are used to (1) provide transparent remote file access to non-object files and (2) to execute object operations on an object which resides on a machine other than the client's host. In the first case, the LFS executes SCP operations to manipulate the local channels on behalf of remote clients. In the second case, the operations (both SCP and non-SCP RPCs) are serviced directly by the object process (on behalf of a remote client). To set up a remote mount connection, an object directory (the formal mount point) on each machine is bound to the the primary caller RPC conduit and the primary callee is the directory to be remote mounted (the actual mount point). Note that the primary callee here is not an RPC conduit. In this model, all of the additional RPC conduits are formed by the SCP operations which return open channels in response to pathname operations that cross the remote mount point. When a non-RPC-caller channel is passed between machines, the RPC mechanism automatically sets up the facilities by which all system calls on the remote machine are converted to SCP RPCs that are communicated to the machine on which the channel resides and serviced by one of the designated LFS processes on that machine.

The remote file access mechanism can be viewed as a special type of object file in which the object directory (the formal mount point) resides on one machine and the object process(es) (i.e. the LFS processes) resides another machine. The intersecting operations on this object are exactly the RPCs which are communicated between the primary caller and primary callee of the RIOCREMOTE. It is incidental that the RPCs differ in that they have to be communicated over a network and that the initial activation procedures involves an overt rendezvous between separate processes which must concurrently execute RIOCREMOTEs on the two machines.

The paradigm of passing a channel to the remote process is the basis for the distributed file system. This technique is subtly different from the remote open mode used by most other distributed file systems (see chapter 8). In particular, it is normal to share (1) a common read/write offset within a file, (2) advisory locking access (or the lack of it) and (3) membership in a channel's process group among processes on several distinct machines. Further, unnamed channels such as pipes (Berkeley sockets in general) can be passed among machines

as channel parameters; these channels are not accessible from remote machines using the remote open model.

5.3 Formal and actual channels

A remote I/O connection on host A to a channel native to host B is represented by a *formal channel* on host A and the channel on host B is called the *actual channel*. A formal channel is created by the arrival of a RPC message containing a channel parameter which is not native to the local machine. Note that the formal channel only contains connection information and RPC buffering. The target actual channel contains all state information and data structures associated with the channel on the callee's machine (in the same configuration as a totally local version of this channel). The primary caller becomes the first formal channel and the primary callee becomes the first actual channel with respect to a host pair.

The correspondence between formal and actual parameters is maintained by the *channel ID* (CID) which is the index of the file table entry on host B. The connection between the particular process on host A which is executing an operation on the formal channel is described by the *RPC ID* (RID) which is the process ID of the caller on host A. In the procedure call analogy, the CID corresponds to a procedure entry point address and the RID has the same function as a procedure's return address. Each RPC packet which is transmitted between the two machines contains both a CID and an RID.

A formal file structure (an entry in the file table) holds all per-remote channel information in the context of the host-to-host connection. The per-remote channel information about an actual channel is stored in a new data structure called the *actual file link* structure. Note that a single local channel can be accessed remotely by several hosts each of which have separate actual file link structures to store the per remote connection information. Separate hash tables are used to expedite the lookup of the correct formal file structure or actual channel link by CID. There is a link from the formal file structure or actual channel link structure to the list (called the RID list) of all RPCs which have the same CID (but different RIDs). In this way (1) an incoming message can be quickly associated with the correct RPC by CID/RID lookup and (2) invalid remote accesses can be rejected.

All inter-machine RPCs originate from operations on formal channels by the caller. All RPCs from all formal channels associated with a particular remote machine are queued on the RPC socket associated with primary caller. The RPC socket is the per RPC conduit data structure which is a separate entity from a Berkeley socket [Leff82]. This socket is analogous to the single output queue to a remote host (the difference being that an RPC is an entity

which both sends and receives messages). The RPC socket contains the buffering high water marks, inode binding information, and pointers to the RPC process control block. When the initial request fragment arrives on the callee's machine, a duplicate RPC structure is created and it is queued to the RPC socket associated with the callee. This socket is either the RPC caller's socket (if the original channel passed between machines was a RPC caller channel) or the LFS socket.

Figure 10 shows the the major data structure organization in the remote level. The abbreviations used in the figure are:

AF is an actual file structure (file table entry). It can be any existing type of file structure other than a formal file structure. The data structure referenced by the AFs (inode structures, Berkeley sockets, or other file table entries) are not shown. Note that a single AF can be shared among many host-to-host connections.

AFL denotes the actual file link structure described above. The AFLs are chained from the actual file hash table header (a small mbuffer). Each AFL contains pointers to the RID list and to the associated AF.

F is used to represent the callee channel of an RPC conduit whose caller channel is accessible from the remote machine (see the upper right quadrant of figure 6). Note that the data structures for this conduit are the same as in the local RPC conduit case (compare with figure 3) except that the RPCs which have a remote origin are also linked into the RID list of the caller channel's AFL associated with the source machine.

PCB is the protocol control block for the host-to-host connection. The protocol control block contains all per-host-to-host connection information. This information includes a pointer to the internet route entry for the remote machine, timeout states for passive closes and keep alive messages (see section 5.5), the head of several event queues (not shown), references to the root of the split channel tree of LFS RPC sockets (only one is shown in the figure) and a pointer to the primary caller RPC socket, plus references to the hash table header for both the formal and actual channels. All PCBs in the system are linked together into a double linked list (not shown) so the correct PCB can be easily found if the address is known.

R is an RPC structure. Each RPC may also be the head of a chain of parameter buffers (not shown in figure 10, see figure 7). The RPC structure for a network RPC differs from local RPC structure by the addition of a new sub-structure to the header to hold the remote information. This remote information includes the RID list pointers, the

event list pointers (not shown), plus pointers (none of which are shown in figure 10) to the PCB, RPC socket, and AFL or FF (depending on whether the RPC is on the caller or callee machine). The remote sub-structure also contains the remote level's per-RPC state machine variables (described in section 5.7). Once the RPC has been accepted by the callee's machine, two RPC structure exist (one on each machine). The caller's RPC structure will be in the RID list of the originating formal file and the socket list of the primary caller. The callee's RPC structure will be in the RID list of the corresponding actual file link structure and the socket list of either the LFS's RPC socket or the target RPC socket (if the actual channel was an RPC caller channel).

FF is a formal file table entry as described above.

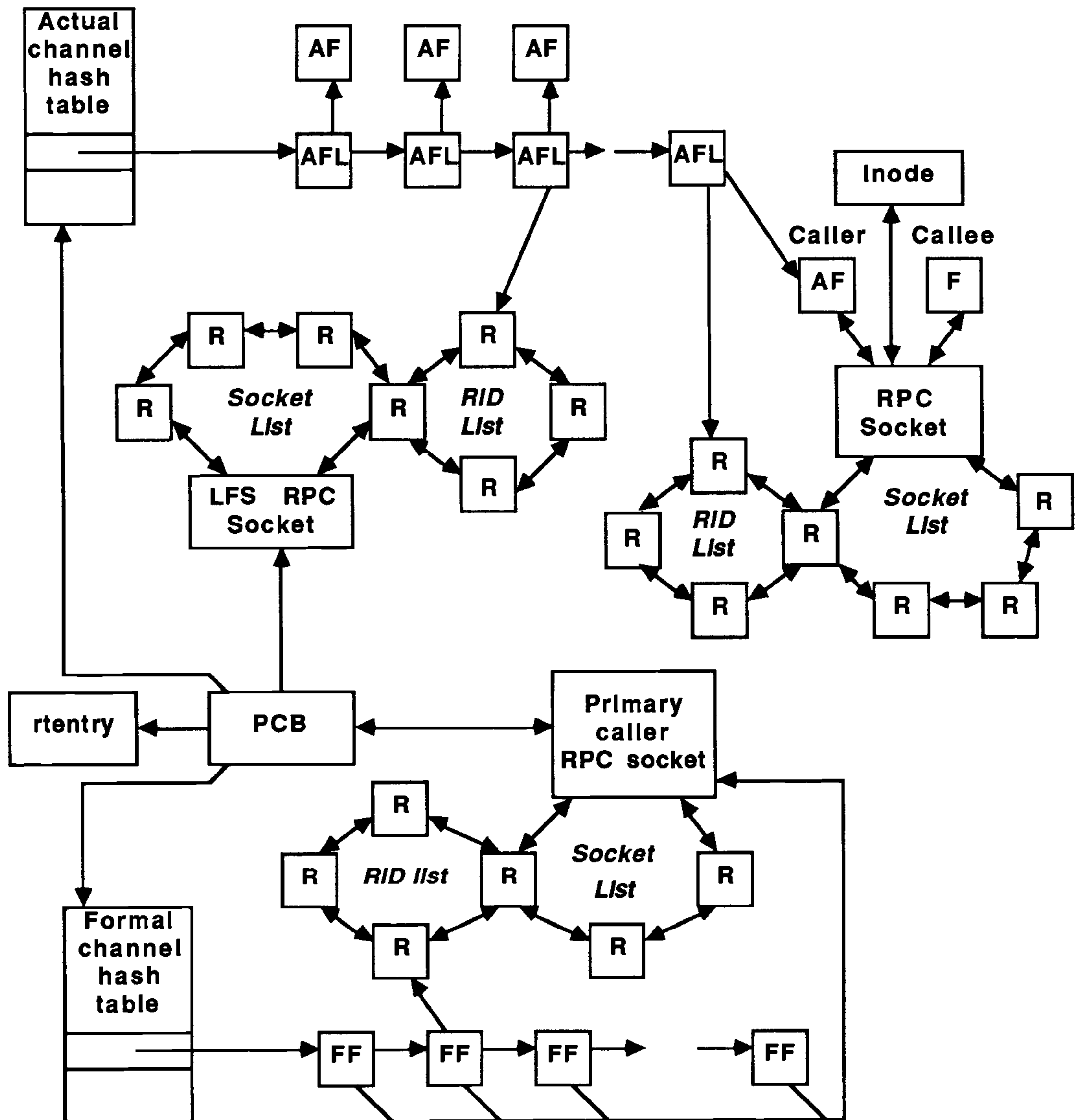

Figure 10: Remote level data structures

5.4 The local file server

Each actual channel can either be a RPC caller channel or a channel of some other type (both types of actual channel data structures are shown in figure 10 by the two AFLs with RID lists). If the actual channel is a RPC caller, incoming RPCs are simply queued on the RPC socket and are serviced by the callee process(es) associated with the socket.

If the actual channel is not an RPC caller, then an agent or server process is required to execute the I/O operations on behalf of the remote clients. The LFS processes are used for this purpose. When a RPC destined for a non RPC caller actual channel arrives, it is queued to the RPC socket of the LFS process, and it must be a SCP operation. When the LFS callee reads this request, it will receive in addition to all other parameters, an extra channel (called the LFS target channel and is passed to the callee via request parameter 0) which is the actual channel on which the operation is to be performed. This mechanism is used for four purposes: (1) It avoids having a separate RPC socket to queue RPCs for each actual channel. (2) It avoids partitioning the set of actual channels among LFS processes due to the limited number of open channels per LFS process. (3) It permits multiple LFS processes to be shared on a FIFO basis among all remote RPCs. This property comes from the fact that all data structures and algorithms which provide synchronization among all processes (including the LFS processes) accessing a given actual file reside in the UNIX kernel (as opposed to the data space of the LFS process). (4) It permits the target actual channels to be passed through the kernel from the process which originally exported this actual channel to the LFS server without any overt communication.

Note that as a consequence of the above scheme, LFS processes can be shared not only among RPCs from the same remote machine but also among many remote machines by simply using the same LFS channel in the RIOCREMOTEs for more than one remote machine. Also the particular set of LFS processes which can be used to provide service to a given SCP operation can depend on that operation by the use of a split channel for LFS service. All of the leaf channels in the split channel tree provided to the RIOCREMOTE operation as the LFS server must be RPC caller channels.

The SCP layer includes a generic LFS program implemented within kernel. That is, the instructions of this LFS program reside in the kernel code segment and they are executed in supervisor state (the processes which result from executing this code are often referred to as "light-weight processes" since they do not incur the overhead of user-kernel transfers of control, nor do they generally maintain the full UNIX user process context. Other examples of light-weight processes are the swapper and the page daemon). An arbitrary number of LFS processes can be created by reentrantly executing the LFS program. These processes use a

user mode virtual address space to hold those system call parameters which are referenced by address from the kernel's system call routines (for example, the data buffer of a read operation). In this way the user virtual address space is used to store the parameters passed between fragments of the request and reply messages. The LFS process has the restriction that it assumes an identical user and group ID space between the two hosts. The only UID transformation is the super user is mapped into an unprivileged user for remote operations. Note that the tasks of implementing local file service for remote processes can be shared among the kernelized LFS and user-provided LFS processes by means of the a split channel used to specify the LFS service channel.

These kernelized LFS processes also implement many of the housekeeping chores in the remote RPC domain. In particular, uprintf()'s are converted to blocking writes (RPC calls if the controlling terminal of a process is a RPC caller channel) by LFS processes and all actual channels which need to be closed by the remote level are closed from the LFS processes (since close is a blocking operation, it cannot be done directly by the remote level).

5.5 Keepalive messages and passive closes

The remote level must be constantly prepared to clean up any RPCs (in any state) if the network communications fail or the tables on the two hosts become unsynchronized (i.e. the two machines disagree on which formal and actual channels connect the two machines). To detect these conditions, periodic keepalive messages (KAMs) are transmitted between the two machines. Each KAM contains a list of all formal channel CIDs and all actual channel CIDs according to the sender. Each host is constantly attempting to timeout each actual channel and each formal channel. The arrival of the KAM verifying that the actual / formal channel does exists resets the timer. If a channel is timed out, a *passive close* is said to occur. For a formal channel, the passive close sets a flag in the formal file structure which inhibits all future I/O and the file structure is deallocated when the last process level reference is closed. For actual channels, the actual channel is placed on the close queue which is serviced by the kernelized LFS processes when the passive close represents the last close on this (local) channel.

When communication between the two hosts is completely lost, a *passive disconnect* occurs. The cleanup procedure for a passive disconnect is exactly the same as doing a passive close on every formal channel and every actual channel connecting the two hosts. The only difference is in the error code returned to the user processes.

The passive close timeout and KAM transmission periods are specified as parameters to the RIOCREMOTE operation. Further the initial connection procedure will redefine the KAM period to be the minimum of the KAM period of the two machines and the passive close time is redefined to be the maximum of the two passive close times.

5.6 The remote open channel transfer procedure

When a channel is transferred from one machine to another, its external representation consists of the host address (the internet host address in this case) of the machine on which the actual channel resides and the CID of the actual channel on that machine. This same representation is used for local RPCs within the kernel for channel parameters. The internet host address, which must be set before the RPC system can be used, is obtained by the sethostid system call.

The file structures, actual file link structures, and forwarding RPCs have extra reference counts to indicate the references from within queued RPCs even though there are no addresses stored here. The high order bits of the CID field in the buffer itself are used as flags to describe exactly which (if any) reference counts are held by the channel parameter while in transit. These bits are updated as the RPC goes through various states. For a local RPC, only reference counts on the file structures involved are used.

In the case of a remote RPC, reference counts held by channel parameters in transit are kept until the acknowledgement of the remote reception is received. This indicates that all structure allocations and reference counts have been obtained on the remote machine. The type of reference counts required by a channel parameter passed between machines depends on the location of the machine on which the file resides (FM) in relation to the message source machine (SM) and message destination machine (DM). There are three possibilities:

The first case is that the channel is a formal channel on SM and the actual channel resides on DM (i.e. FM is the same as DM). In this case the reference count on the formal channel structure is held by the sending machine (until ACKed). When the message arrives at the DM, this channel parameter reference acquires a reference count on the actual file link structure (in the remote level) before the ACK is sent. The actual file link reference is converted to a reference file structure associated with the channel in the process level before a descriptor can be allocated to the receiving process. There is nothing to allocate (on either the SM or DM) since the channel in transit is being returned to the machine on which it resides.

The second case is where the FM is the same as the SM. In this case the channel parameter is an actual channel on the sending machine. In this case the reference count on the file structure is converted to a reference count on the actual file link structure by the remote level of the SM. The actual file link structure may need to be allocated if the channel does not already have remote references from the DM. If a new remote channel connection is being created, the formal file will be allocated by the remote level on DM before this message is acknowledged. The reference to the formal file structure is held until it is converted to a descriptor for the receiving process.

The third case is where the SM, DM and FM are all distinct machines. In this case, the SM must make arrangements with the FM before the channel can be transferred to the DM. This process is called *forwarding*. The purpose of the forwarding operation is to allocate a new actual file link structure on the FM (if one does not already exist) which will permit the target channel to be directly manipulated from the DM (i.e. without involving the SM). The forwarding process uses a *short RPC* from the SM to the FM to ask the FM the set up the actual file link structure. A short RPC is a request/reply sequence where the reply and the request – received – acknowledge are the same message and the entire request is serviced in the remote level. Short RPCs are also used in signal propagation, and closing a remote file. Note that the forwarding operation may require a new RIOCREMOTE to be executed on the FM to establish communication directly with the DM (this would be done by the connection server). All forwarding RPCs for channels in the same message execute concurrently (and before the message can be sent to the DM). This algorithm leads to a race condition between the passive close timer on the FM and the eventual sending of KAMs containing the required CID from the DM to the FM. To prevent a passive close of the FM's actual file, special entries are added to the KAMs from SM to FM to indicate that the forwarding of this particular CID is in progress. These special entries in the SM's KAMs to FM will be discontinued when the message to DM is acknowledged. This technique also permits the RPC between the SM and DM to be aborted without any explicit message to the FM to cleanup the new actual file link. The newly created actual channel link on the FM will be deallocated by a passive close (after a considerable delay).

5.7 The remote level RPC state machine

The remote level is implemented as a collection of concurrent finite state machines, one for each network RPC in progress. The state transitions of these machines occur in response to events (i.e. input to the finite state machines) and actions can be performed at these times. Note that the state machines cannot block (i.e. use the sleep() function to wait for events)

since the transitions may be executed from interrupt routines. When the remote RPC needs to wait for an event (such as the arrival of an acknowledge message), the state machine is designed to execute a transition when the event occurs. The input events are totally asynchronous. This makes the code which implements the state machine a critical section which requires mutually exclusive access to the RPC data structures to execute a transition. To do this, the entire remote level operates at the software priority level of splnet. Higher priority inputs (such as the arrival of a new packet from IP) must reduce queue the input event and schedule a software interrupt at the splnet level to service the event. Lower priority events (such as being invoked from the process level) must raise the priority of the processor before the event can be serviced.

The state transition diagram for the caller is shown in figure 11 and a similar diagram for the callee is given in figure 12. The gross features of these diagrams show the transfer of each fragment of a request message by a loop the ring of states shown on the upper part of each figure. The transfer of each fragment of the reply message is denoted by a loop ring of states in the lower part of each diagram. The state machines (i.e. the RPCs) are dynamically created and destroyed. The entering arrows indicate creation and the exiting arrows denote destruction. The normal communication of an RPC is represented by the major fully connected portion of the diagrams. Various error and other abnormal events are indicated by the transitions labeled A through E, described on the side of each diagram. The WERRA and WABTA states denote the communication of a locally detected error condition to the opposite machine.

The individual transitions are denoted by labels on the various arcs in the diagram. The notation is *input-event : output-action*. If several inputs result in the same action, the will be separated by commas. If more than one action is to be performed (for any single input), the list of actions is separated by commas. The list of output actions may be empty. The abbreviations used are described below:

T — denotes a timeout event scheduled by queuing the RPC in the timeout event list. Timeout events are used for the scheduling of re-transmissions of the previous message if an acknowledge has not been received.

M — indicates a storage available event. The RPC state machine's progress can be suspended if it cannot allocate storage for an outgoing message or an actual file link structure. If this occurs the RPC will queue itself on a wait for storage list.

R() — indicates the reception of the message type indicated in parentheses. All received messages are individual IP datagrams.

S() denotes the sending of the message in parentheses as a single IP datagram. A timeout to re-transmit is automatically scheduled each time a non-acknowledge message is passed to IP. If the single buffer required in the send operation cannot be obtained, a storage event is scheduled. In many cases, it is known that the buffer exists because a buffer has been saved for this purpose.

Q() can be used as either an input or an output. When used as an input, the transition is made when the remote level is called from the process level in the indicated process level state. When used as an output, this indicates that the remote level is placing the process level state machine in the indicated state.

First is uses an input modifier to denote the first fragment of the request or reply message. It is preceded by a plus (or a minus) to indicate "and this is (or is not) the first fragment of the message" respectively.

Last is the same as first described above, except the predicate indicates the last fragment in a request or reply message.

W means wakeup (or unblock) the waiting process level.

X means exit or destroy this remote level state machine.

I means *internalize* (or convert from external to internal representation) the open channel passed in this fragment. This operation involves (1) allocating the formal file structure if one does not already exist and (2) acquiring the reference counts on the file structure or actual file link structure involved. Note that allocating an entry in the file[] table is not a blocking operation, it either succeeds or fails.

SIGPIPE action means send SIGPIPE to the callee process. The caller has been terminated by a signal or the connection has been lost due to a passive close while the callee in in the invoked state.

SIGURG action means send SIGURG to the callee process. This is due to a non-fatal signal received by the caller while the callee is in the invoke state.

Done is an input which is true if the request or reply fragment done message for the previous fragment has been received.

The messages sent between the callee and caller machines are abbreviated as follows:

RQ is a request fragment.

RQA is the request fragment acknowledge. All acknowledge messages are unreliably transferred. It is up to the sender of the original message being acknowledged to timeout and re-transmit to solicit an acknowledge.

RQD is the request fragment done message. It is the notification to the caller that the previous request fragment has been read by a callee and the next fragment can now be sent. This protocol permits the formation of the next request fragment to be overlapped with the reading of the previous fragment.

RQDA is the request fragment done acknowledge.

RP is a reply fragment.

RPA is the reply fragment receive acknowledge.

RPD is the reply fragment done message. It is exactly analogous the request fragment done message.

RPDA is the reply fragment done acknowledge.

NFS notifies the callee that the caller has received a non-fatal signal which is explicitly handled by the caller.

NFSA is the non-fatal signal message acknowledge.

ABT is the caller's notification to the callee that the RPC is being aborted. The specific error condition is included in the message. This message is also used for notification to the callee that the caller has been terminated by a fatal signal.

ABTA is the abort acknowledge.

ERR is the callee's notification to the caller that the RPC is being aborted.

ERRA is the error acknowledge.

If an RPC is deleted and the final acknowledge is lost, a re-transmission destined for a non-existent RPC will be generically acknowledged. In addition to the messages described above, there are request and reply-acknowledge for each of the short RPCs: forward, close and signal.

The states labels and the function of each state is described below:

States of the Caller

XRQ1 is an acronym for externalize request one. The term *externalize* refers to the process of converting an open channel representation to its external form which is used on the network. This state is actually a sequence of states, one for each open channel in this request fragment. Individual open channels are the only parameter which cannot be arbitrarily split by fragment boundaries. Each channel is required to lay totally within a fragment. The XRQ1 state will visit each channel within the fragment to (1) allocate the actual file link structure if one is required and (2) initiate the forwarding short RPC if one is required. Externalization requires separate state(s) because it can be blocked by out-of-storage situations for allocating the actual file link and the short RPC structure.

XRQ2 completes the externalization process by visiting each channel parameter for which a forwarding operation was executed to wait for the completion of that operation.

WRQD is the state used to wait for a request done message. This state is skipped if the request done message for the previous fragment (or in the case of the first fragment) has been received earlier.

WRQA is the wait for request fragment acknowledge state.

WERW is the state where the remote level waits for the process level to produce the next request fragment.

WNFSA is used only when a non-fatal signal notification is been passed to the callee. The signal must arrive between the sending of the last request fragment and the arrival of the first reply fragment. This state will consume RQAs and NFSAs and enter the reply code when the first reply fragment is received. Only one notification of a non-fatal signal arrival is passed to the callee regardless of how many non-fatal signals are actually received.

WRP is the state used to wait for the first reply message fragment and it is entered following last request acknowledge. The duration of the callee's process level invoked state (see figure 8) is spent here.

WERR is the state the caller remote level waits for the caller process level to consume the reply fragment.

WRPDA is the state where the caller waits for the request done acknowledge message. In the case of the last reply fragment, the RPD message is used to inform the callee of successful and complete reception of the reply which will permit it to return from the write operation.

WABTA is used to wait for the callee to acknowledge the caller abortion of this RPC.

States of the Callee

WEER is the state where the callee remote level waits for the callee process level to consume a request fragment. The remote level state machine is created in response to a new first request fragment which can be queued to the target RPC socket. This may be either before or after a callee has entered the kernel on a read operation. In the former case, the remote machine will wait in WEER for the read to be executed in addition to the actual processing of the first fragment. In the case of the last request fragment, the remote level will wait in WEER not only for the consumption of the previous request fragment but also for the duration of the callee process level invoked state (i.e. until a reply is written; see figure 9).

WRQDA is where the callee waits for the request processing done message to be acknowledged.

WRQ is the state used to await the arrival of subsequent request fragments.

XRP1 and XRP2 are the states used to externalize channel parameters in the reply. The algorithm is identical to the externalization of the request messages.

WRPD is used to wait for the reply fragment done message. This message can be received asynchronously in any previous state. If this is the case or if this is the first reply fragment, this wait is skipped. In the case of the last reply fragment done message, the remote level exits.

WRPA is used to wait for the reply fragment received acknowledge and implement the re-transmit logic. In the case of the last message, the transition is made directly to the WRPD state to wait for confirmation of the final completion of the RPC.

WEEW is used by the callee remote level to wait for the callee process level to produce the next reply fragment.

Caller Remote Level State Machine

WRQD
R(RQD): S(RQDA)
S(RQ)
-First, -Done
R(RQD):
S(RQDA)
Done,First: S(RQ)
XRQ2
T,M:
S(RQ)
WRQA
ABC
R(RQD)-Last:
S(RQDA),W,Q(SRQ)
XRQ1
R(RQD):
S(RQDA)
R(RQA)-Last: W
Q(SRQ)
WERW
Q(RRQ)
R(RQD): S(RQDA)
Q(RRQ)
Q(NFS)+Last:
S(NFS)
R(RQA)+Last: Q(INVKD)
T,M:
S(RPD)
WRPDA
R(RPDA)-Last
R(RPDA)+Last
ABC
R(RQA)
WNFSA
WRP
Q(SRP)
-Last:
S(RPD)
AC
Q(NFS)+
First: S(NFS)
R(RP): S(RPA),
Q(RRP),I,W
R(NFSA):
-T,-M
T,M:
S(NFS)
WERR
R(RP): S(RPA),
Q(RRP,W,I
B
Q(ABT): S(ABT)
A R(ERR): S(ERRA),W,X
T,M: S(ABT)
WABTA
R(ABTA): Q(ERR),W,X
B Q(ABT)
C PC: Q(ERR),W,X
R(RQA),R(RPDA),R(RP)

Figure 11: Caller's remote level state machine

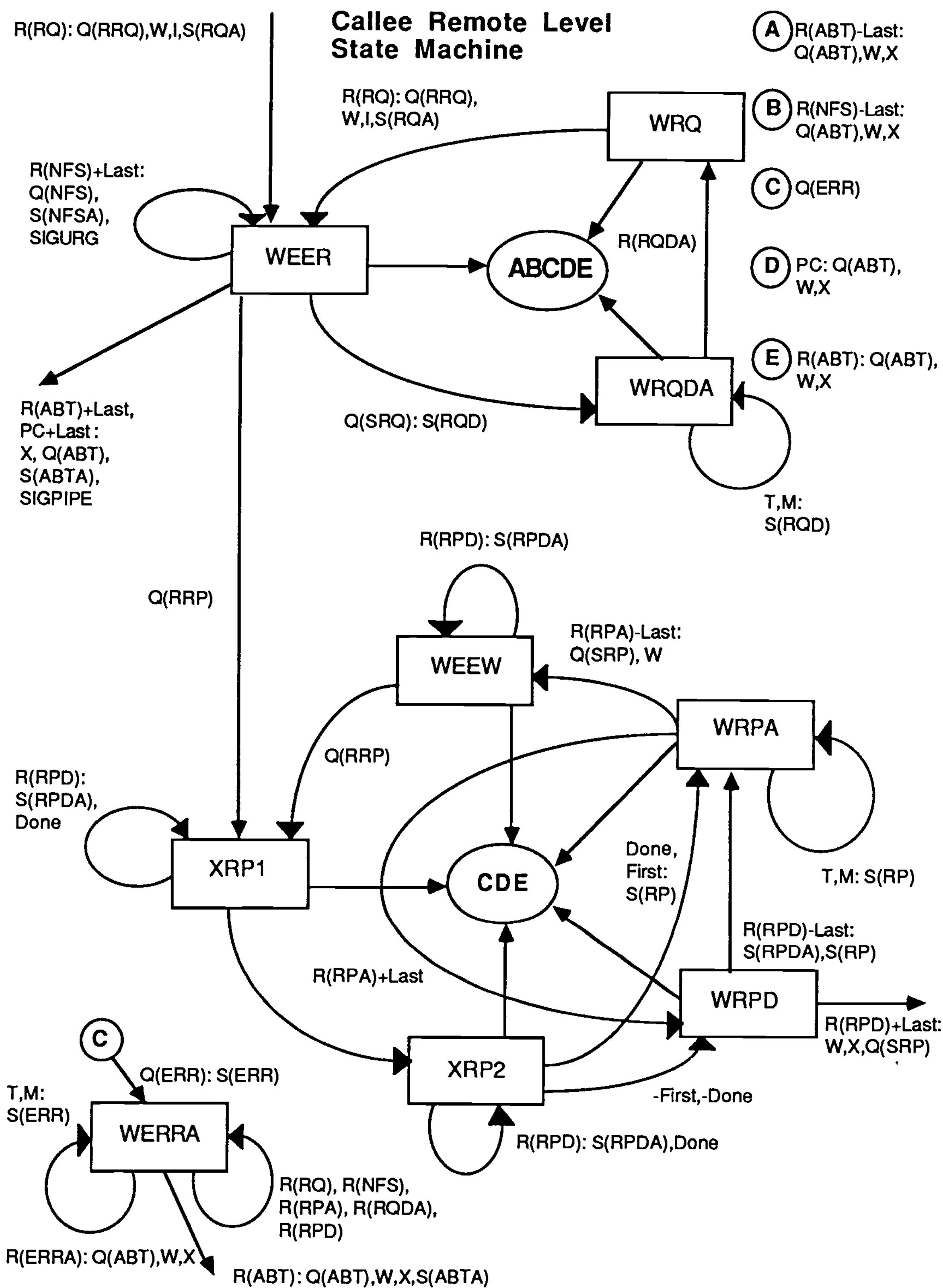

Figure 12: Callee's remote level state machine

5.8 Heterogeneous machines

Heterogeneous processors represent primitive data objects in different representations. The general problem is beyond the scope of this implementation of the RPC mechanism, however three concessions to differing architectures have been made. First, the lowest address byte of a multi-byte integer is always stored in an even address location in memory (it is also only placed in an even message relative address in the IP datagram). Second, 2-byte integers (shorts) may require swapping bytes. Third, the conversion of 4-byte integers (longs) may require swapping shorts (after each short has been converted according to the second rule). A more general approach is mentioned in section 9.1.

All information manipulated by the RPC protocol itself is converted to the internet network byte order [NIC82] for transmission by IP. This conversion is never seen outside of the RPC protocol routines which totally encapsulate this conversion. These quantities are always represented in the local byte order for the caller and callee. The quantities converted by RPC are:

(1) All message header information, excluding host addresses which are already in network byte order.

(2) All size / type tags

(3) All CIDs used in the external representation of channel parameters.

(4) Request parameter 0

(5) Reply parameter 0

(6) Identification parameters

All data parameters are transferred as a binary sequence of bytes. The RPC mechanism is unaware of the internal structure of this information so no conversion can be done. To aid in the conversion of 2 and 4 byte integers, request parameter 0 contains two flags which indicate the a byte swap and/or a word swap is required to convert between the caller's host byte order and the callee's host byte order. Note that this transformation is self-inverting so it can be used for both the request and reply data parameters.

5.9 Data copying and storage allocation

The RPC protocol makes use of the m-buffer pool for all storage requirements. The buffers are used for (1) RPC sockets, (2) RPC protocol control blocks, (3) actual file link structures, (4) RPC headers, (5) formal and actual channel hash table headers, and (6) parameter storage (i.e. fragment buffering). Overuse of the available storage is prevented by using a high water mark for the number of queued RPCs in each RPC socket. The maximum size of a RPC is bounded by the maximum fragment size. Attempting to exceed the high water mark will cause the process level to become blocked. New RPCs received from the network will be discarded if the socket is unable to queue these new RPCs (the sender will re-transmit until successful reception occurs, and re-transmit period increases exponentially to a constant ceiling). A network RPC fragment will only be discarded when it is the first fragment of a new RPCs request message. To make it less likely that this will happen more than once, a single RPC slot in a RPC socket can be reserved for remote RPCs only (with timeout). As a performance issue, this could be increased to more than one slot if too many RPCs are discarded in this way.

The parameter data is never copied between buffers. This is done by adding a reference count in the m-buffer structure. Typically copying can occur when a message is transmitted since one copy needs to be saved for possible re-transmits. Since the RPC parameter buffers are chained together in order, a message passed to IP contains a unique header buffer which points to the same copy of the parameter buffers used to save this message fragment for possible re-transmission. Figure 13 shows the data structure analogous to figure 7 where a message associated with a RPC is queued for network output.

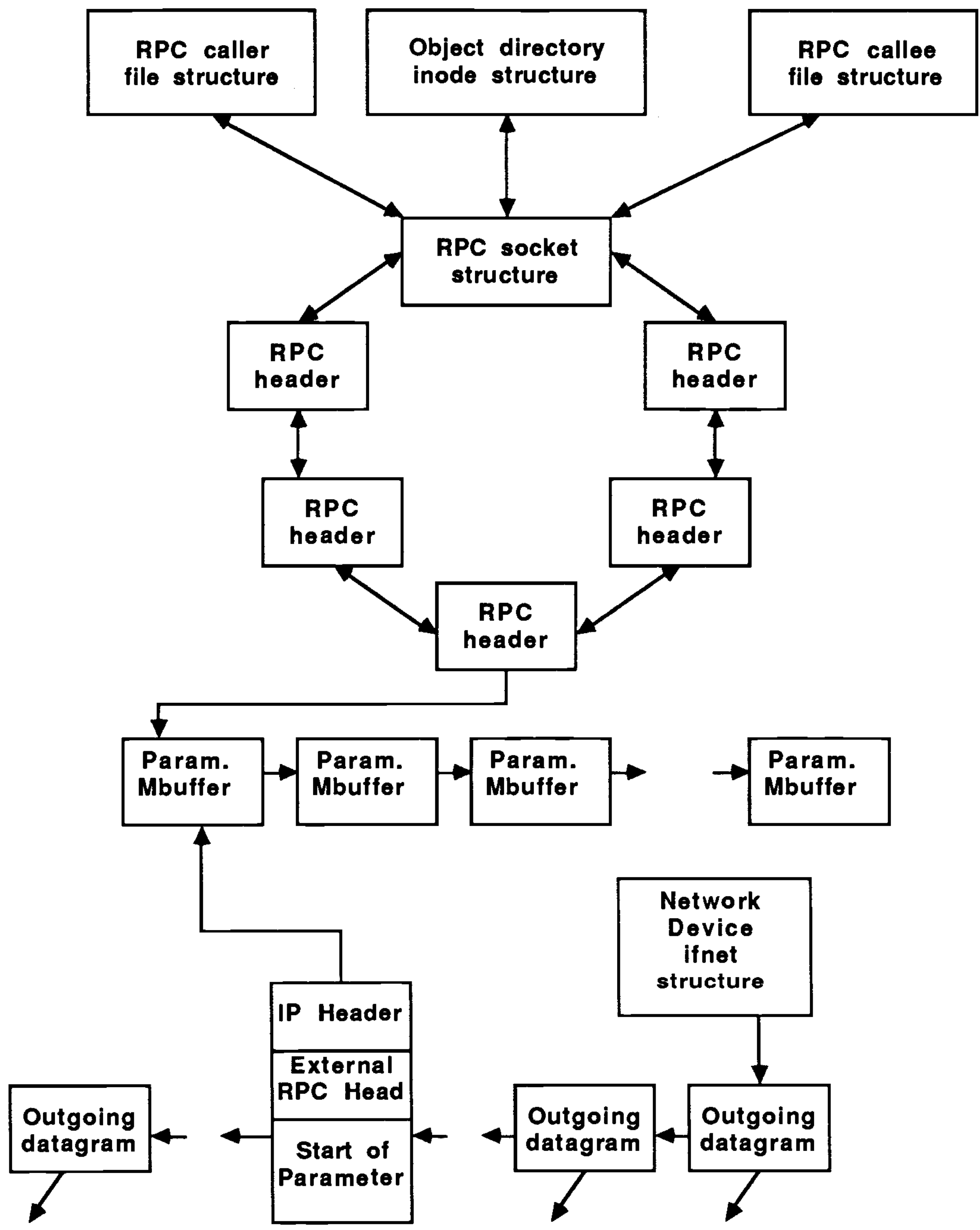

Figure 13: RPC message queued for network output

As a consequence of this strategy the remote level will never require more than one small buffer to be allocated at one time by any single RPC. All larger allocations are done at the process level where blocking allocation functions are used. Further, an event list is used (unlike the existing UNIX networking software) to retain the state of a RPC blocked due to an exhausted buffer supply. The continuation of processing of these RPCs occurs as soon as storage becomes available. Large buffer allocations are also required by the network driver, however if a large amount of hardware buffering is available, loss of data due to running out of buffer storage is unlikely. These strategies are designed to make the RPC mechanism robust against and suffer minimal performance degradation due to tight storage situations.

5.10 Distributed process groups

The representation of process group information in a multi-machine environment is more complex. A process group identifier is a datum which is stored both in the process structures and in the I/O data structures and is used to specify which process get I/O related signals. All the processes with the same process group identifier as the I/O object get signals from the I/O object. Traditionally process IDs have been used for this purpose. Because the process ID is visible outside the kernel, this cannot be easily changed. The approach taken in the distributed RPC system is to make the process group identifier the pair of local process ID and host address (the same algorithm was used in the LOCUS operating system [LOCU84] and V project [Cher83] for identification of processes in a network environment).

This decision is not totally compatible (The full internet address as opposed to a short machine index is used here. This requires a physically larger process group entity.) with existing software however it preserves some important properties: (1) The choice of a new process ID can be made locally and are not altered by the kernel during the group's lifetime (new process IDs cannot intersect any existing process group *in the whole distributed system).* (2) Local processes using local I/O objects can still use process ID based software and the process ID based process group control system calls. (3) The extension to distributed process groups requires that user process deal with a 6-byte data structure instead of a 2-byte entity (all algorithms are unchanged). New functions to get and set the process group this distributed process group. (4) By analogy, the distributed process ID is the host address plus local process ID pair which is exactly what is needed for process migration.

This decision to make a larger process group data structure violates the transparency goal. The alternative is to use a short (16-bit) machine identifier which has a one-to-one correspondence with the host address. The problems here is that some of the details in the automatic maintenance of these parallel host address have not been resolved. It seems that

broadcast polling of all other sites is required here and the current RPC mechanism only deals in unicast messages.

A closely related issue is how the I/O related signals are returned to either all processes in a process group or to a single process (according to the way the process group was defined). When a signal producing event occurs associated with an actual channel, short RPCs are executed to all remote machines which have access to this channel. This RPC communicates the particular signal and the target process or process group. The recipient machines will search their process tables for the target process(es). This algorithm is not quite the same as the single host case because only those hosts with the channel open can receive the signal notification.

The controlling terminal is a real open channel (a first class citizen), so it these processes will always be signal recipients. This change was also required to allow non-terminal devices to be a controlling terminal.

Chapter 6

Example applications of object files

This section is intended to describe various ways object files can be used to implement software which either unnecessarily complex under UNIX without object files or will require substantial modifications to the kernel. Please note examples below are restricted to being thumbnail sketches only.

6.1 A simple spooling system

The basic idea here is to use an object file to fold the front end of the queuing system (lpr) and the back end of the queuing system (lpd) into a single process. This will avoid the major problems that arise from coordination of the front end and the back end which traditionally have been solved by means of lock files and signals. Further the idea of namespace extension can be used to replace the display queue program (lpq) and the remove a queue entry program (lprm). The standard utility programs for displaying the contents of a directory (ls) and deleting a file (rm) can be use in place of these utilities.

The front end function would be invoked by redirecting output to /obj/lpr

pr foo.c > /obj/lpr

Various parameters can be communicated to the queuing by encoding the parameters into the suffix passed to the object process:

pr foo.c > /obj/lpr/"label=prfoo copies=3 print=lgp banner=foo.c"

Another facility which is lacking in most UNIX queuing systems is the ability to *spool in time*. The idea here is to suspend the producing process(es) until the time to print directly on the output device arrives. This avoids the necessity of creating temporary files which may be too large to hold on a workstation's local disk.

A slightly more subtle problem which is avoided is the need for the queuing system to maintain split identities. The split identity problem occurs when a program wants to (1) be able to manipulate a restricted collection of files (e.g. spooling directory, /usr/spool/lpd) and (2) be able to access the user's files with exactly the set of permissions attributed to the user. This avoids the need to be setuid root or to do some of clever setreuid()ing.

Another noteworthy point to observe is that the object file provides a natural encapsulation of the files, both data and executable, within the object directory. Compare this to the hither and yon placement of files in the existing queuing system with files in /usr/ucb, /etc, /usr/spool and /usr/lib.

6.2 Object mail system

The idea is to implement a mail message storage system under the control of an object file. As in MH, the individual messages will be entities in the namespace of the object file. This is a structured namespace which would implement folders (directory entities) similar to the native file system. Again, the name suffix is used to parameterize the list messages being displayed:

ls /obj/mail/new/"arrived=today from=doug,john@lastvax showsubject"

The at-sign is passed to the object process since it occurs in the suffix of an intersecting operation. Setup options, such as the message list display format in the above example, and state information, such as processing history, can be stored in the private memory of the object file on a per user basis. The processing history gives the concept of current message as well as processing context where the verbose options can be specified only once and inherited by future references. This context is not restricted to the collection of mail related programs, but is available to any software which accesses the mail data base (i.e. the object). This information is preserved across login sessions and is shared among concurrent login sessions of the same user. This sort of parameterization will make programs such as the MH [Rose85] scan redundant. The compose function can be replaced directly by the editor, as shown in the following example of how to create a new message:

vi /obj/mail/junk/"draft to=scott@foovax cc=jhr@pyramid-1 showheader"

To send this message

mv /obj/mail/junk/*to=scott@foovax* /obj/mail/"send logcopy"

The use of wildcard matching can be done by the shell (as shown above) if there is a standard format for the name of an entity. Alternatively, specialized pattern matching operators can be placed in the pathname suffix to implement object specific wildcards (probably requires the suffix to be quoted with respect to the shell). The former approach has the advantage of being able to interface to the shell for a centralized universal pattern matcher. The latter has advantage that it can be used from within a client program through the primitive system call interface. Also note that both methods can be used simultaneously.

The mail message storage object can do more than simply convert MH style processing options into the filename. There are also two functional improvements that can be made. First, large numbers of small messages can be stored efficiently without having to round up to the next disk block size multiple. The smallest disk block is often 1024 bytes in UNIX system while the average mail message is an order of magnitude smaller. The second optimization is to be able to share a single copy of a message among many addressees.

Other features which could be added include support for bulletin boards, and electronic discussion groups.

6.3 A resource allocator

In this example a subset of the special files are designated to be serially reusable resources. This could include self-service tape and cartridge disk drives, utility communication interfaces, and modems. The modems are to be used for dialin devices when they are not allocated to some other program. By default the allocation is to take place invisibly whenever an intersecting operation occurs (the open being the most important special case) and released when the last close occurs.

The special files to be allocated will be installed in a new directory called /res. This directory is an object directory so that the special files are now part of the private memory of the /res object file. Duplicate object files which have no accessibility can be kept in /dev in order to satisfy programs like ps and those which use ttyname(). The allocation problem is simple. Whenever a open operation is detected, the resource manager (i.e. the object process) will consult its tables to determine whether the device is in use. If the device is available, the resource manager will open the device and return a split channel consisting of the special file combined with a caller RPC channel such that all operations are directed to the special file. The RPC conduit is used only to detect the last close of the special file. The resource allocator or one of its subprocess will be doing a select on the callee side of the RPC conduit which will return as readable when the caller is closed. When the close of the special file is detected, the resource allocator will update its tables to show that the special file is available.

A slight complication occurs with the dialin/dialout modems. When used for dialin, /etc/init will attempt to open the special file associated with the modem which will block until carrier from the remote modem is detected. The object process will detect an open by /etc/init by noting that the client does not have a controlling terminal and it has a UID of 0. In this case, a subprocess of the object process will execute the open on the named special file which will block. The special file will not be marked as in use until the open returns. Again,

the split file technique is used to determine when the close (i.e. the logout) occurs. If a non-/etc/init process wants the special file before carrier is detected, the subprocess attempting to open is killed and the special file is awarded to the non-/etc/init process. Note that there is no need to adjust /etc/ttys or kill the copy of /etc/init attempting to do the open. The attempt to login will continue when the non-/etc/init process releases the special file.

To retain ownership between opens, a "keep" command could be used. The simplest form would open the target special file and sleep. A more elaborate method would require a secure way to detect the logout event of the invoking user.

The table which indicates which devices are allocated would be kept in the memory of the object process to avoid the necessity of any cleanup or initialization commands to be run at boot time. This is a common problem with server maintained state information which is kept on disk. Again, the status of any particular device is obtained via ls.

The are many things which can be added to such a resource allocator. Perhaps one of the most useful is to provide (1) automatic free modem selection and (2) modem independence by use of a name such as

/res/dialout=9-1-213-890-8312

where a connection from the open will be return only when contact with the remote modem has been established.

6.4 A foreign network interface

In this example, assume that you have a network interface which has V7 style special file driver (the driver provides only buffering and interface control functions). This network does not talk in any protocol now implemented in the kernelized protocol drivers. Further there are nameable, stream oriented logical channels which one wishes to access from UNIX process level software for which the open-read-write-close paradigm is applicable.

As examples, consider the telenet packet switching network, a proprietary network standard used by a particular manufacturer (e.g. IBM's system network architecture (SNA) or Apple Computer's AppleBus), or a TELEX machine.

Without going into the details, an object file is an obvious way to encapsulate the interface software (e.g. protocol driver, et cetera) while providing named virtual circuit connections to the UNIX processes. Again, any parameterization required can be installed in the name suffix strings.

6.5 A foreign file system

In this example, an on-disk file system which is not the native UNIX file system is to be transparently manipulated as if it were part of the native UNIX file system. The important property of this object file is that all UNIX software is to be completely unaware that it is not manipulating UNIX files. This means that the result of the stat/fstat/lstat system calls and the result of reading a directory must be constructed from corresponding information in the foreign file system.

6.6 An ISAM database

In this example, the indexed sequential access method (ISAM) to used to access files in a file system with a flat directory structure. This file system is stored in an disk partition which is not used by UNIX. Some of the primitive operations are: (1) create an ISAM file (this operation specifies record size and location plus size of the key field), (2) delete an ISAM file, (3) open an ISAM file, (4) insert record, (5) delete record, (6) lookup record by key, (7) sequential read, (8) sequential write, and (9) seek.

To implement the ISAM data base, one would want to be UNIX compatible for all sequential operations and define new RPC operations to implement the operations of insert record, delete record, plus lookup by key. Create can be done either with a new RPC or via alphanumeric parameters encoded in the filename. The object process will implement the entire file system including (1) layout of the files on the physical disk, (2) block allocation strategies, (3) keeping track of available storage on disk, (4) buffer management within the object process' address space, and (4) RPC operation service.

In reference to object files, the noteworthy aspects of this example are that previously non-existing operations (e.g. lookup by key) have been added without kernel modifications or replicated routines in all clients.

6.7 The checkin – checkout paradigm

The checkin - checkout paradigm is used by storage control software where a copy of an entity is extracted from its database and is provided on a file for manipulation. When the queried entity is accessed for modification, the database system records a locked state for the entity (the readers and writers problem). The entity will be returned to its database on file after the modification is complete. The act of removing a file copy is called *checkout* and the retuning of the modified copy is called *checkin* (the terms checkin and checkout were taken

from the Revision Control System [Tich82] which makes use of this scheme). An object file interface can be used to make the checkin and checkout operations invisible by replacing the checking and checkout operation by copy operations. In many cases creation of copy (or temporary) file can be avoided entirely. One of the other problems with the checkin – checkout mechanism is that some of the attributes of the files are not set meaningfully. As an example, the date of a file in some contexts should be the date of creation of the temporary file (for incremental backups) and in other contexts, it should be the date of creation of the underlying entity (for make). The object file solves this problem.

6.8 A window manager

An object file interface can be used by a window manager to dynamically create and delete the fictitious terminal special files associated with each window. Further, many of the parameterization functions (such as controlling the size and location of the window) can be done with the suffix string without having to (1) embed the window manager specific ioctls in otherwise general purpose software or (2) use a collection of auxiliary control programs to encapsulate these functions. The creation of a new window can be done automatically by a new open operation. This would lead to an easy, window manager independent implementation of pop-up boxes for dialog, errors, et cetera. This mechanism assumes that part of the window management facilities are implemented in a UNIX process (as opposed to a totally kernelized implementation or a display processor based implementation).

6.9 Transaction processing

An object file is a convenient place to encapsulate transaction processing. In a transaction based model, operations on a data base are performed in a manner such that they are not permanent until a *commit* operation is done. These operations can be explicitly canceled by an *abort* operation or by a hardware failure in order to insure data consistency. This transaction mechanism is not available under most UNIXes for operations on ordinary files (see [LOCU84] for a description of an operating system based implementation for all files) however it can be provided selectively by object files.

Chapter 7

Implementation status

At this time the implementation of object files as described in this outline is only partially complete. The local RPC mechanism and the client side SCP code (the code which generates SCP RPCs) exists in an experimental UNIX kernel and has been superficially tested. A completely untested implementation of the remote level and the kernelized LFS also exists.

Only one object file has been implemented so far. This object file implements a BSD 2.9 UNIX file system using the methodology described in section 6.5. All of the SCP operations except mount, umount, ioctl, and fcntl, and the socket operations have been implemented. There are a few noteworthy points:

First, the implementation of symbolic links will use the root of the full local file system (not the root of the simulated 2.9 file system). Together with ".." from the root of the 2.9 file system, this means that pathnames can enter and exit this object.

Second, an implementation of FIFOs (named pipes) has been added. This implementation is similar to the pre-4.2 disk based pipes except that a directory reference is considered both a potential reader and a potential writer. The system 3 (and system 5) requirement that both a reader and a writer must be present in order to do I/O is dropped since the full state of the FIFO is represented on disk. This mechanism was used to test object files as a mechanism for implementing a coroutine scheduler between UNIX processes.

Third, the object process' internal architecture is very similar to a UNIX kernel in miniature. The UNIX data structures for the buf, inode, file, and proc tables all exist and serve exactly the same function as their kernel counterparts. Most of the routines which manipulate these data structures also exist. Further, there is a user block (which is actually implemented as part of the proc table since there is no distinction between swappable and non-swappable process context) which contains the per-process kernel stack. The process scheduling routines (sleep, wakeup and switch) are used to control context switching and the blocking of processes. The need for processes occurs because waiting on a FIFO read/write operation or a flock system call is a long term event which would cause deadlock unless the object file could service other requests in the interim.

Chapter 8

Related work

The idea for the object files is taken from the GOOSE paper operating system [Nebu82, Bail81] where the User Defined File Type (UDFT) implements a similar function. The RPC mechanism replaces the stream oriented implementation in GOOSE. Further, the mechanism was entirely local to a single processor. A similar mechanism for associating software layers with directory entries is described in [Lind81].

Another source of related work is in the area of distributed file systems as implemented in UNIX [Brow82, Brun85, Cole85, Sand85, Tich84]. An object file can be considered a generalization of the distributed file system where the remote server processes are provided by the object file creator to implement the methods particular to his object. The other generalization is that the servers do not need to be on a remote machine.

All of the distributed file systems make use of an effective RPC protocol, however it is usually implemented in terms of lower level communication primitives. An explicit RPC discipline is implemented for the Newcastle Connection [Brow82, Panz82], however it is (1) implemented as user mode library routines and (2) it presents an unreliable interface (the RPC can fail simply due to the server being busy for an extended period of time). It is not clear how exception processing is handled in this system. Also, there is no mention the problems of identity parameters or channel parameters.

The implementation of RPCs in Mesa is described in [Birr84]. The Grapevine nameserver [Birr82] is used to export (or register) service names and to import (or bind) clients to these names. The export, import and call operation are all distinct operations. This RPC mechanism includes an interface to the Mesa (among others) language via "stub routines". The term RPC as used in this paper refers only request-reply communication paradigm where the caller is blocked for the duration of the RPC transaction, but excluding the complier language interface. The transport layer is provide by the PUP [Bogg79] datagram service with the RPCRuntime layer implementing RPC specific reliable delivery mechanism on top of PUP. Further, a much more elaborate exception handling facility is used to support the stackable exception handlers provided in Mesa. This stack extends between the caller and callee (across the two machines). Further, the invocation of the exception handlers on the caller's machine while the callee is in execution occurs in the sense of nested RPC calls without disturbing the context of the outer level RPCs. Authentication is handled using

Grapevine to provide an accurate and secure identification of the caller to the callee and vise-versa. In addition, encryption services are available. An outline of how similar facilities can be provided with object files is given in section 9.1.

The V kernel [Cher83, Cher84] is based on a RPC-like communication paradigm for all process-to-process (as well as many process-to-kernel) interactions. V kernel processes are much cheaper than their UNIX counterparts. The V kernel currently does not support paging and the concept of I/O is provided by explicit agreement with a I/O server. This means that channel parameters, as a distinct parameter type, are not required; channel parameters are required in the RPC mechanism described in this paper because it is built on top of the existing UNIX kernel's I/O facilities. The binding between the caller and callee is done by the caller knowing the V kernel PID (machine address, local process ID pair) which is required in the *send* (analogous to a call) operation. Nameservers can be used to provide the name to PID translations. The V kernel provides the following extensions to the simple RPC model: (1) *CopyTo* and *CopyFrom* primitives which can be used by the callee in the invoked state to access the caller's address space (if allowed by the caller) and (2) a mechanism for forwarding RPCs (see section 9.9) to other callees. The multicast mechanisms sends are also available. It is not clear how authentication is handled in the V kernel, however it seems to be left to higher levels of software in the interest of efficiency.

Sun Microsystems' Network File System [Sand85, SUN86a, SUN86b, SUN86c, SUN86d] (NFS) contains an remote procedure call mechanism built on top of UDP (TCP can also be used, but not for NFS) [NIC82]. This mechanism relies on the use of idempotent (repeated executions of the same RPC will not change the state) RPCs since the kernel does not provide reliable delivery through UDP. The resulting stateless servers provide easy recovery from crashes, however it violates the basic semantics of object files (and UNIX files too). The connectionless nature of the RPC mechanism means that the server must provide the client with an opaque handle which represents each target file. This handle must be specified in each remote operation. These design considerations mean that NFS cannot correctly deal with (a) unlinking, changing the ownership of, or changing the mode of an open file and (b) it cannot support file locking (due to state requirements in the server). Pathname operations are forwarded communicated to the remote machine in a one component at a time in order to detect inter-machine hops (this information can be kept only by the client). This mechanism is also used to accurately traverse the .. back through a remote mount point paths. The remote mount facility described in this paper cannot deal with the .. through mount point problem (the point of view taken is that a remote mount point is considered to be a symbolic link to the remote file system). The lower level RPC service provided by SUN supports the transfer of the equivalent of an identification parameter (only from caller to callee)

augmented by the hostname string, however the real UID/GID pair is not transferred. In addition, SUN provides batching (RPCs which do not require a reply delivered reliably by TCP without blocking the caller) and broadcast (zero or more replies returned unreliably) modes. The binding and rendezvous facilities are primitive: the user-visible "name" consists of a triple of program number, version number, and procedure number which are mapped to and from internet port numbers by a portserver program. This protocol does not support the transfer of open channels between processes.

The overall architecture of object files is the exact inverse of the structure which appears in both the V system and distributed Mesa as implemented at Xerox PARC. In both of the latter cases, the RPC mechanism was the lowest level operation, with the file system built on top of the RPC layer using the file server model. In object files, the RPC mechanism is built on top of the local operating systems (which include the file systems) so special parameter types (identification and channel parameters) are added to deal which quantities managed by these operating systems. The Sun approach treats the local file system and the NFS a separate software forks with in the kernel under the *vnode* interface. The Sun local file system is traditional UNIX in contradistinction to the NFS branch which is similar to Xerox/V file server approach.

The concept of passing open channels between processes as part of a general IPC mechanism is implemented (partially, at least) in the 4.2 BSD AF_UNIX domain datagrams where the open channels are passed as "access rights". This communication domain is restricted to being local to a single processor.

In some cases, the implementors of the distributed file system provided an interface to append different file system types. This mechanism can be used to provide an object file like interface to the new file system. This was used to implement "/proc" directory under 8th Edition UNIX [Kill84] and to add an IBM PC file system driver to a SUN Workstation [Sand85]. The distinctions between this approach and object files are: (1) The interface described above is strictly within the UNIX Kernel and the code which is analogous to the object process is kernel resident. (2) The connection mechanism is not automatic, but requires an explicit connection by a generalized mount system call. (3) The interface operations are a set of internal interface routines as opposed to raw system calls.

The addition of object files to UNIX is not an attempt to make UNIX into an object oriented operating system [Niel84, Wulf74], however some of the same issues are being addressed. In particular, the ability to replace, expand and customize parts of the operating system is common. The differences are that modification via object files is restricted to the I/O operations and the algorithms involved exist in UNIX processes. In object oriented

operating systems, the objects correspond to individual modules within the kernel itself.

Chapter 9

Future work

The object file implementation is an experiment in software interfaces. The results of that experiment are not yet complete, however several area of additional work are indicated. These area are described below.

9.1 A library of routines useful to object programs

Such a library would contain routines for validating parameters, management of request and reply message storage, byte order conversion routines, coroutine support for C programs, the outermost interpreter loop which interfaced by a table of routines to service the SCP operations, et cetera.

Direct use of the object file mechanism to provide RPC service can be done at this level. There are three interfaces to this RPC mechanism: (1) export (executed by the callee), (2) import (executed by the caller) and (3) call. The export interface is parameterized by a unique pathname, the routine to be called and an optional specification for the arguments expected in the request and to be returned in the reply (if no specification is given, the arguments are passed in the RPC untyped format; a maximum size is required here). The choice of the unique pathname should encode server/service name, procedure code (if the callee can be invoked by more than one remote procedure) and version number. The latter two entities can optionally be encoded as RPC request parameters. The export mechanism will create an object directory node if it does not exist and RIOCBIND a RPC channel to this node. This channel will be placed in asynchronous read mode so that a SIGASYNC will be delivered when a RPC request arrives. The signal service routine will execute a select poll to determine which RPC channel has been called, followed by a read to get the request message, followed by the call to the target procedure.

The import procedure has two forms which correspond to accessing the callee by name or accessing the callee by an established channel. The former case is analogous to a reliably delivered datagram which should be used if only a small number of calls are to be made. The import procedure, in this case, only initializes internal data structures without executing any system calls. In the latter case, a connection is established using a SCP open operation to create the RPC conduit; this action will not invoke any of the target procedures in the callee, however it is serviced with the library routines of the callee. For both of the cases, the

external parameters of the import routine are (1) the pathname used to rendezvous with the callee, (2) a flag specifying the case described above, and (3) an optional description of the parameters passed (a single untyped parameter is used if explicit types are not specified). The return value of the import routine is a pointer to a the function to call to execute the remote procedure. This procedure (a dynamically created thunk routine) will execute the RIOCCALL operation as either a nioctl or an ioctl depending on the case described above.

The conversion of data between the internal representation of heterogeneous machines is not handled by the kernel (and cannot be handled by the kernel the information currently available; further, the kernel would be an inappropriate place for such conversion). This means that a library would be the reasonable place to implement the conversion routines. The client's transparency requirements suggest that end-to-end conversion routines be located in the object process (or the callee in general), however determination of the exact type of the internal representation of the client's machine is both awkward and will result in excessively large, slow, and complex conversion routines. The current implementation, which is restricted to dealing only conversion of binary integers in 2's complement representation in 2 and 4 bytes sizes, does end-to-end conversion, but this method does not generalize easily. The alternative is to convert all data to a network standard data representation; the conversion is done on both the caller and callee machines. The transparency requirements will force SCP to do the client conversion in the kernel. An example specification is given in [SUN86b].

9.2 Support for shared text object exec operations

Shared text support requires that the object process recognize two or more object exec operations as referring to the same target entity. The object process in this case would have to return the same indirect channel parameter in all cases where the same reentrant program was exec'ed (note that direct object exec operations where the returned channel is that of an executable regular file can take advantage of both shared text and demand paging according to the file's magic number). The kernel would support the shared text object exec's by using a pointer to the (unique) caller file structure in place of the inode reference from the text structure.

9.3 An identity server

At this time identification parameters consist of a structure containing the sender's user and group identifications. When a RPC is sent to remote machine, the user and group ID

numbers should be transformed into the corresponding ID number on that machine using a algorithm similar to that implemented in RFS [Brun85]. In the RPC scenario, the RPC protocol would contact the server to invisibly accomplish the ID translation. In addition some method for establishing what translation is appropriate must be done at the time the connection is established. The RPC mechanism would cache the translations (separate caches for UIDs and GIDs) provided by the server on a per remote host basis in order to keep the server out of the critical path.

The server would also be directly callable by the object process to handle the (f)chown, and (l,f)stat system calls (and other direct use of UID/GIDs). The proposed mechanism is yet another ioctl where the translation required would be determined by the source machine of the currently invoked RPC. Mechanisms for both client-to-server and server-to-client transformations must be provided.

The main unresolved issue here is how (and when) are the per-host name to UID and name to GID maps passed between machines.

9.4 Improvements to the RPC communication protocol

The RPC communication protocol used between machines is, at best, simplistic. One obvious improvement would be to use a sliding window algorithm to transfer multi-fragment request and replies [Tane81]. The amount of data contained in KAM messages should be reduced. The forwarding of split channels can also be optimized.

There are several schemes for reducing the number of messages transmitted between the callee and caller machines. First, delay the request acknowledge message transmission until either (1) a re-transmit is received, (2) the first reply fragment is produced, or (3) a timeout occurs. Second, remove the use of request/reply fragment consumed (done) messages and let the retransmission mechanism in the subsequent message handle this flow control problem. Third, request/reply fragments can be transmitted without individual acknowledges using a group acknowledgements for the maximum number of fragments which can be buffered within the kernel.

Also, the option of using a virtual circuit connection as the inter-machine communication mechanism must be investigated in more detail.

9.5 Increase the generality of object files

At this time object files cannot be used for "core" files, to save data produced by the acct() system call, or to store quota information. Also the object program cannot be itself an object file. Most of these restrictions are due to the fact that they are implementationally awkward (and of questionable value).

9.6 The RPC mechanism and process migration

One of the difficulties in process migration is the necessity of moving the total environment of the process along with the process itself. The RPC mechanism provides the means for (1) open channels, (2) current directories, (3) controlling terminal attributes, and (4) process groups to be moved with the target process. The key feature is passing channel parameters transparently between the migrating process' source and destination machines.

The general case of process migration may be considered expensive in terms of communication cost, however it is relatively cheap at the time of an exec operation. In this case, the move between processors is done when there is no user address space to be transferred which permits it to occur among heterogeneous processors to the extent that common operating system services are available. A combined fork / exec system call as typified by the *run* system call implemented in LOCUS [LOCU84] provides a convenient packaging for doing remote execution.

9.7 Inter-object hard links

The problem of simulating a inward directed edge into an object (in particular to an entity within the object) has not been solved. What must be done is that when such an edge is traversed in a pathname, the object containing the referenced entity must be invoked in such a way that it can determine that the reference occurred via the edge. Further, the representation of this edge must occur on disk, outside of the influence of either the source or destination object if it is to be static (in fact the inter-object hard link must be representable as a primitive directory hard link).

Related issues are: (1) What should be done if the destination object is not there due to its machine being down (temporarily)? (2) Who checks (guarantees) the consistency of the directed graph as a whole (i.e. what is analogous to fsck)? (3) Can the consistency check be done without involving the object processes using the on-disk structure only? (4) What about object hard linking to directories inside of objects?

If an underlying UNIX hard link to a dummy file within the private memory of the destination object were used, there must be some way of associating the dummy file with the destination object represented by the object directory itself such that traversing this directed edge will invoke (and possibly activate) the destination object. If this were done, inter-object hard links can be simulated by hard links from within the source object's private memory to the dummy files within the destination object's private memory. Adding the information to uniquely describe which dummy file was referenced is not a problem, however establishing the association between the dummy file and the object directory is difficult.

9.8 Objects with multiple intersection points

The idea here is to provide a single object with multiple nodes in the file system which can be the site at which intersecting operations originate. The association between the object and its intersection points must be retained while the object is passive. The first problem is that the activating intersecting operations can occur via any intersection point. The second problem is how to specify which intersection point was used in a particular intersecting operation. Three hypothetical approaches to this problem are given below.

First, duplicate the object for all intersection points when the first instance of the object is activated, bind all remaining object directories to new RPC channels at the object process' execution (initialization) time. The problems with this approach are the race conditions in the initial binding and that all intersection points have to have separate RPC conduits (which limits the number of intersection points to the number of open channels per process).

Second, use multiple object directories which are distinguished from regular object directories by the class program being a sub-object directory. All mount points of a single object would be linked (hard or symbolic) to the same sub-object directory which would provide the UID/GID/current directory to the object process. Only one level of sub-object directory would be allowed. The particular intersecting object directory is communicated by passing a channel to the object directory as the LFS target channel in request parameter 0. The problems with this approach is that (object) directories cannot be multiply linked (by non-super-users) which prevents this (and the first solution) from being used to represent in-links.

Third, use symbolic links to the (unique) object directory. The symbolic links would have to be distinguished by a unique inode type or by an escape sequence within the contents of the symbolic link. As in the second case, the particular intersection point used by an operation is indicated by passing a channel to that intersection point (the symbolic link itself) to the object process via request parameter 0. The advantage of this mechanism is that it can

(probably) be used to represent hard in-links, however in all other respects it is grotesque.

9.9 RPC Forwarding

RPC forwarding is the operation by which a callee determines that the RPC should be totally serviced by another callee. The purpose for examining forwarding operation is to eliminating RPC nesting in these instances. In the existing scheme, this will require a nested RPC call by the original callee to the subsequent callee. This will block the original callee for the duration RPC service thereby degrading the response to service requests of the original callee and making deadlock a possibility. The pass-through operations in object files are an example of a situation where forwarding is appropriate. Another example would be the case where a pathname in the distributed file system passes through more than one remote host.

A way to avoid nesting RPC callees was mentioned in [Cher84] and implemented in the V kernel. The idea presented is to have original callee to send a message to the caller asking the caller to redirect the RPC at a new (or subsequent) callee. In the V system, this is a simple operation where the original callee only need specify the V system PID (i.e. the combined host address and process ID on that host). The comparable function for the object file / RPC context would require delivery of a new RPC-caller channel and/or a substitute pathname. An important degenerate case occurs when a SCP operation is forwarded back to a primitive object on the local site. In this case a RPC is not required.

The problem with this type of redirection is that it conflicts with the implicit goal of being able to invoke insecure (or untrusted) objects without the granting the object process the right to execute under the client's identity. In this case, a more privileged process can invoke a service from a less privileged object file without transferring any of its privilege to the callee. If an arbitrary redirection facility exists, the less privileged object process could redirect an unlink operation from a more privileged client to an arbitrary local file. A (unfortunate) consequence of the existing policy is that pass-through operations are performed under the identity of the object process instead of the identity of the client (this can be avoided only by root privileged object processes).

It is clear that forwarding would be a very useful concept to integrate into the RPC mechanism, but more work is required to do it in general (for non-root processes).

9.10 Improvements to the exception service model

The exception service model used for RPC corresponds to the exception service model used in the UNIX kernel for system calls. This exception service model is different from the exception service model provided by a procedure call with a process. A system call is either interruptible or atomic with respect to signals (exceptions). If a system call is interruptible execution is rolled back to the the beginning of the system call were exception service can take place. The reason for this strategy is to permit longjmp()s (stack unwinding) without having to clean up any state information in the kernel. Further, no information about which exceptions are pending is communicated by the non-fatal signal notification is available to the callee and only the first exception (signal) results in notification to the callee.

A more general scheme would be to treat an exception as an asynchronous (possibly remote) procedure call to the top of a stack of exception handers for each exception. This scheme is implemented in [Birr84]. This requires integrating the concept of stack unwinding (which is compiler language dependent) into the RPC mechanism. More fundamentally, the change permits a caller to have one outstanding RPC call for each active exception invocation plus one for the top level code. Many of the existing algorithms would be broken by this change.

Bibliography

[Bail81] Bailey, K. A., et al, "User Defined Files", Operating Systems Review, ACM 1981, Vol 15, No 4, pgs 75-84.

[Bogg79] Boggs, D. R., et al., "PUP: An Internetwork Architecture", Xerox Palo Alto Research Center report CSL-79-10, 3333 Coyote Hill Road, Palo Alto, CA 94304 (Oct 1979).

[Brow82] Brownbridge, D. R., et al., "The Newcastle Connection or UNIXes of the World Unite!", Software -- Practice and Experience, Vol 12., pgs 1147-1162 (1982).

[Birr82] Birrel, A. D., et al., "Grapevine: An exercise in distributed computing", Communications of the ACM, Vol 25, No 4 (April 1982), pgs 260-274.

[Birr84] Birrel, A. D., and Nelson, B. J., "Implementing Remote Procedure Calls", ACM Transactions on Computer Systems, Vol 2., No 1. (Feb 1984), pgs 39-59.

[Brun85] Brunhoff, Todd., "Installing and Operating RFS", Computer Environments Group, Applied Research Group, Tektronix, Beaverton, OR (1985).

[Cher83] Cheriton, D. R., and Zwaenepoel, W. "The Distributed V Kernel and its performance for Diskless Workstations", Proceedings of the 9th SOSP, (Nov 1983).

[Cher84] Cheriton, D. R., "The V Kernel: A Software Base for Distributed Systems", IEEE Software 1, 2 (April 1984), pgs 19-42.

[Cole85] Cole, Clement T., et al., "An implementation of an Extended File System for UNIX", USENIX Summer 1985 Conference Proceedings.

[Gold85] Goldberg, Adele and Robson, David, "Smalltalk 80 The Language and its Implementation", Addison-Wesley Publishing, Reading MA, May 1983

[Kill84] Killian, Thomas J., "Processes as Files", USENIX Summer 1984 Conference Proceedings.

[Leff82] Leffler, S. J., et al, "4.2 BSD Interprocess Communication Primer", Computer Systems Research Group, EECS-UCB (June 1982).

[Lind81] Lindsey, D. C., "On Binding Layers of Software", Operating Systems Review, ACM 1981, Vol 15, No 2, pgs 22-37.

[LOCU84] LOCUS Computing Corporation, "The LOCUS Distributed System Architecture, Edition 3.1", Santa Monica, CA (June 1984)

[Nebu82] Nebula Software Group, "The GOOSE Bible, Rough Draft 6", 1982 (Unpublished).

[NIC82] Network Information Center, "Internet Protocol Transition Workbook", SRI International, Menlo Park, CA, 94025 (March 1982).

[Niel84] Nielsen, E. R., et al., "An Expandable Object-Based UNIX Kernel", USENIX Summer 1984 Conference Proceedings.

[Panz82] Panzieri, F. and Shrivastava, S. K., "Reliable Remote Procedure Calls for Distributed UNIX: An Implementation Study", Proceedings of the 2nd Symposium on Reliability in Distributed Software and Database Systems (July 1982).

[Pope81] Popek, G., et al. "LOCUS: A Network Transparent, High Reliability Distributed System." Proceedings of the 8th Symposium on Operating System Principles, ACM (Dec 1981), pgs 169-177.

[Rich74] Richie, D. M. and Thompson, K., "The UNIX Time-Sharing System", Communications of the ACM 17(7), pgs 365-375, 1974.

[Rose85] Rose, M. T., and Romine, J. L., "MH5: How to process 200 messages a day and still get some real work done", USENIX Summer 1985 Conference Proceedings.

[Sand85] Sandberg, R., et al, "Design and Implementation of the Sun Network File System", USENIX Summer 1985 Conference Proceedings.

[SUN86a] Sun Microsystems, "Remote Procedure Call Programming Guide", Revision B (Feb 1986), 2550 Garcia Ave., Mountain View, CA, 94043.

[SUN86b] Sun Microsystems, "External Data Representation Protocol Specification", Revision B (Feb 1986), 2550 Garcia Ave., Mountain View, CA, 94043.

[SUN86c] Sun Microsystems, "Remote Procedure Call Protocol Specification", Revision B (Feb 1986), 2550 Garcia Ave., Mountain View, CA 94043.

[SUN86d] Sun Microsystems, "Network File System Protocol Specification", Revision B (Feb 1986), 2550 Garcia Ave., Mountain View, CA 94043.

[Tane81] Tanenbaum, A. S., "Computer Networks", Prentice Hall Inc., Englewood Cliffs N.J. 1981.

[Tich82] Tichy, W. F., "Design, implementation and evaluation of a Revision Control System", Proceedings of the 6th International Conference on Software Engineering, IPF, ACM, IEEE, NBS (Sept 1982) pgs 58-67.

[Tich84] Tichy, W. F., et al, "Toward a Distributed File System", USENIX Summer 1984 Conference Proceedings.

[Ucb83] "UNIX Programmer's Manual", 4.2 Berkeley Software Distribution (1983), Computer Science Division, Department of Electrical Engineering and Computer Science, Berkeley, CA, 94720.

[Wulf74] Wulf, W., et al., "Hydra: The kernel of a multiprocessor operating system", Communications of the ACM 17(6), pgs 337-345 (June 1974).