sun
microsystems

# Networking
# *on the* Sun Workstation

Sun Microsystems, Inc. • 2550 Garcia Avenue • Mountain View, CA 94043 • 415-960-1300

Part No: 800-1177-01
Revision A of 15 May, 1985

# Revision History

| Rev | Date | Comments |
|---|---|---|
| **A-$\alpha$** | 19 November 1984 | First release of this manual; mostly new material. |
| **A-$\beta$** | 1 February 1985 | Second release of this manual, with minor revisions. |
| **A** | 15 April 1985 | Third release of this manual, for customer shipment. |

# Sun's Network File System

# Contents

# Sun's Network File System

## 1. Introduction

This document gives an overview of Sun's network file system, which allows users to mount directories across the network, and then to treat remote files as if they were local. The first section is a bit elementary, so advanced users may want to skip straight to the examples of how it works. Beginning users may not be interested in the third section, which discusses network file system architecture.

The Network File System (NFS) is a facility for sharing files in a heterogeneous environment of machines, operating systems, and networks. Sharing is accomplished by mounting a remote filesystem, then reading or writing files in place. The NFS is open-ended, and Sun Microsystems encourages customers and other vendors to take advantage of the interface to extend the capabilities of other systems.

A distributed network of personal workstations can provide more aggregate computing power than a mainframe computer, with far less variation in response time over the course of the day. Thus, a network of personal computers is generally more cost-effective than a central mainframe computer, particularly when considering the value of people's time. However, for large programming projects and database applications, a mainframe has often been preferred, because all files can be stored on a single machine.

Those who work with unconnected personal computers know the inconveniences resulting from data fragmentation. Even in a network environment, sharing programs and data has sometimes been difficult. Files either had to be copied to each machine where they were needed, or users had to log in to the remote machine with the required files. Network logins were time-consuming, and having multiple copies of a file got confusing as incompatible changes were made to separate copies.

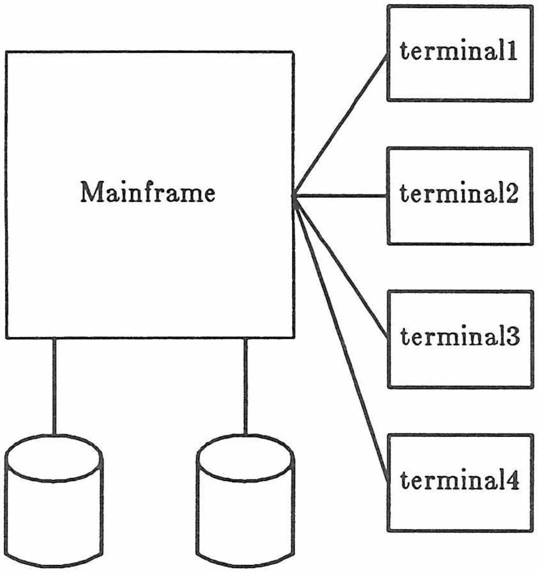
To solve this problem, Sun designed a distributed filesystem that permits client systems to gain access to shared files on a remote system. Client machines request resources provided by other machines, called servers. A server machine makes particular filesystems available, which client machines can mount as local filesystems. Thus, users can access remote files as if they were on the local machine.

The NFS was not designed by extending the UNIX† operating system onto the network. Instead, the NFS was designed to fit into Sun's network services architecture. Thus, NFS is not a distributed operating system, but rather, an interface to allow a variety of machines and operating systems to play the role of client or server. Sun has opened the NFS interface to customers and other vendors, in order to encourage the development of a rich set of applications working together on a single network.

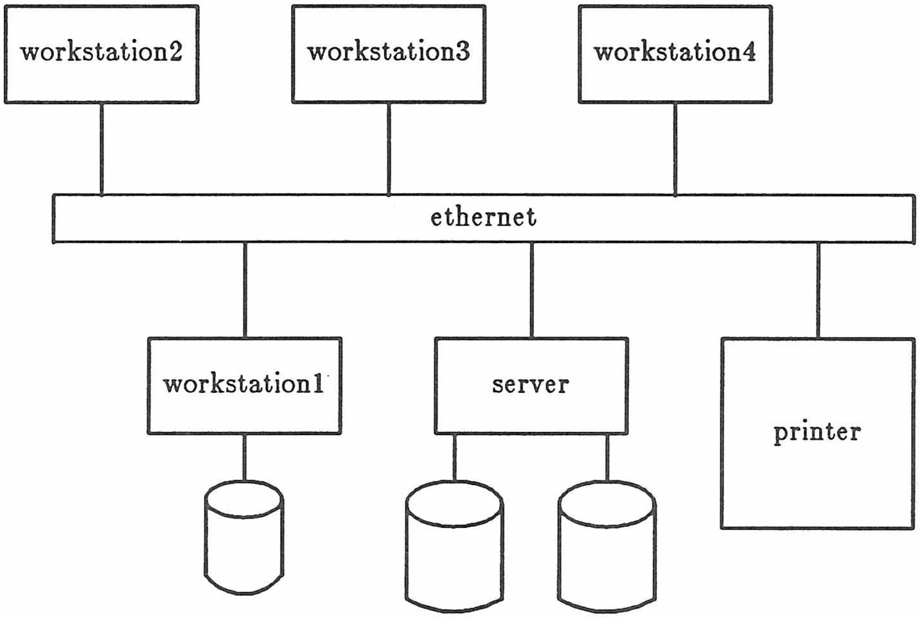
† UNIX is a trademark of Bell Laboratories.

## 1.1. Computing Environments

The current computing environment in many businesses and universities looks like this:



The major problem with this environment is competition for CPU cycles. The workstation environment solves that problem, but introduces more disk drives into the picture. A network of workstations looks like this:



Sun's goal with NFS was to make all disks available as needed. Individual workstations have access to all information residing anywhere on the network. Printers and supercomputers may also be available somewhere on the network.

## 1.2. Terms and Concepts

A machine that provides resources to the network is a ***server***, while a machine that employs these resources is a ***client***. A machine may be both a server and a client. A person logged in on a client machine is a ***user***, while a program or set of programs that run on a client is an ***application***. There is a distinction between the code implementing the operations of a filesystem, (called ***filesystem operations***), and the data making up the filesystem's structure and contents. (called ***filesystem data***).

A traditional UNIX filesystem is composed of directories and files, each of which has a corresponding ***inode*** (index node), containing administrative information about the file, such as location, size, ownership, permissions, and access times. ***Inodes*** are assigned unique numbers within a filesystem, but a file on one filesystem could have the same number as a file on another filesystem. This is a problem in a network environment, because remote filesystems need to be mounted dynamically, and numbering conflicts would cause havoc. To solve this problem, Sun has designed the virtual file system (VFS), based on the ***vnode***, a generalized implementation of ***inodes*** that are unique across filesystems.

The Remote Procedure Call (RPC) facility provides a mechanism whereby one process (the ***caller*** process) can have another process (the ***server*** process) execute a procedure call, as if the caller process had executed the procedure call in its own address space (as in the local model of a procedure call). Because the caller and the server are now two separate processes, they no longer have to live on the same physical machine.

The RPC mechanism is implemented as a library of procedures, plus a specification for portable data transmission, known as the eXternal Data Representation (XDR). Both RPC and XDR are portable, providing a kind of standard I/O library for interprocess communication. Thus programmers now have a standardized access to sockets without having to be concerned about the low-level details of the `accept`, `bind`, and `select` procedures.

The Yellow Pages (YP) is a network service to ease the job of administering networked machines. The YP is a centralized read-only database. For a client on the network file system, this means that an application's access to data served by the YP is independent of the relative locations of the client and the server. The YP database on the server provides password, group, network, and host information to client machines.

## 1.3. Comparison with Predecessors

The Network File System (NFS) is composed of a modified UNIX kernel, a set of library routines, and a collection of utility commands. The NFS presents a network client with a complete remote filesystem. Since NFS is largely transparent to the user, this document tells you about things you might not otherwise notice. Sun's NFS is an open system that can accommodate other machines on the net, even non-UNIX systems, without compromising security.

Sun users may be familiar with two previous networking schemes, `rcp` and ND. The first is a remote copy utility program that uses the networking facilities of 4.2 BSD to copy files from one machine to another. The second is a proprietary device driver for the Sun that makes raw disk available over the network. The NFS does not completely replace ND, so servers and clients will be running both ND and NFS.

Because machines need ND to boot, an NFS server still needs a ***/pub*** partition. However, unlike the old ND configuration, under NFS this partition contains only ***/pub/vmunix***, ***/pub/boot***,

*/pub/stand* and */pub/bin*. There is a separate file system mounted on */usr* containing everything else important. For example, */usr/bin* used to be a symbolic link to */pub/usr/bin*; now the server gets */usr/bin* off its own disk, while a client gets it by mounting the remote */usr* filesystem onto the local */usr* directory. This is true of */lib* as well. The other standard NFS remote mount is called */usr2*, where users' home directories reside.

An exception arises when a client mounts a server's */usr* filesystem on its directory. Some files in */usr* should be private, such as */usr/adm*, */usr/spool*, */usr/tmp*, among others. To get around the problem, these private files are symbolic links to */private/usr*. In an ND configuration, a few files in */usr/lib*, such as *crontab*, *aliases*, and *sendmail.cf* were private; these files are now symbolic links to */private/usr/lib*.

### *1.3.1. NFS and RCP*

The remote copy utility (`rcp`) allows data transfer only in units of files. The client of `rcp` supplies the path name of a file on a remote machine, and receives a stream of bytes in return. Access control is based on the client's login name and host name.

The major problem with `rcp` is that it is not transparent to the user, who winds up with a redundant copy of the desired file. The NFS, by contrast, is transparent — only one copy of the file is necessary. Another problem is that `rcp` does nothing but copy files. In a sense, there needs to be one remote command for every regular command: for example, `rdiff` to perform differential file comparisons across machines. By providing entire filesystems, NFS makes this unnecessary.

### *1.3.2. NFS and ND*

Sun's Network Disk (ND) is a device driver that makes a raw disk available using a simple protocol. The ND client builds its own filesystem, given the disk. Disk space on the server machine is partitioned, and diskless client machines mount one partition as their root filesystem, and another as their */usr* filesystem. Symbolic links can be made between this pseudo-filesystem and files on the server machine.

Under ND, access control of disk areas is based solely on the requester's Internet Protocol (IP) address. Since IP addresses are assumed to be unique, this does not permit file sharing by the ND server. The NFS, on the other hand, allows file sharing. The use of the IP address as the basis of access control has two other drawbacks: first, an erroneous or malicious piece of network software can easily corrupt a user's disk just by supplying an IP address; and second, it violates protocol layering concepts and makes it difficult to change a client's IP address or ND server. Since the server emulates only a disk and not a filesystem, there can be no cacheing on the server side. The NFS permits cacheing, with concomitant performance improvements.

# 2. Examples of How it Works

## 2.1. Mounting a Remote Filesystem

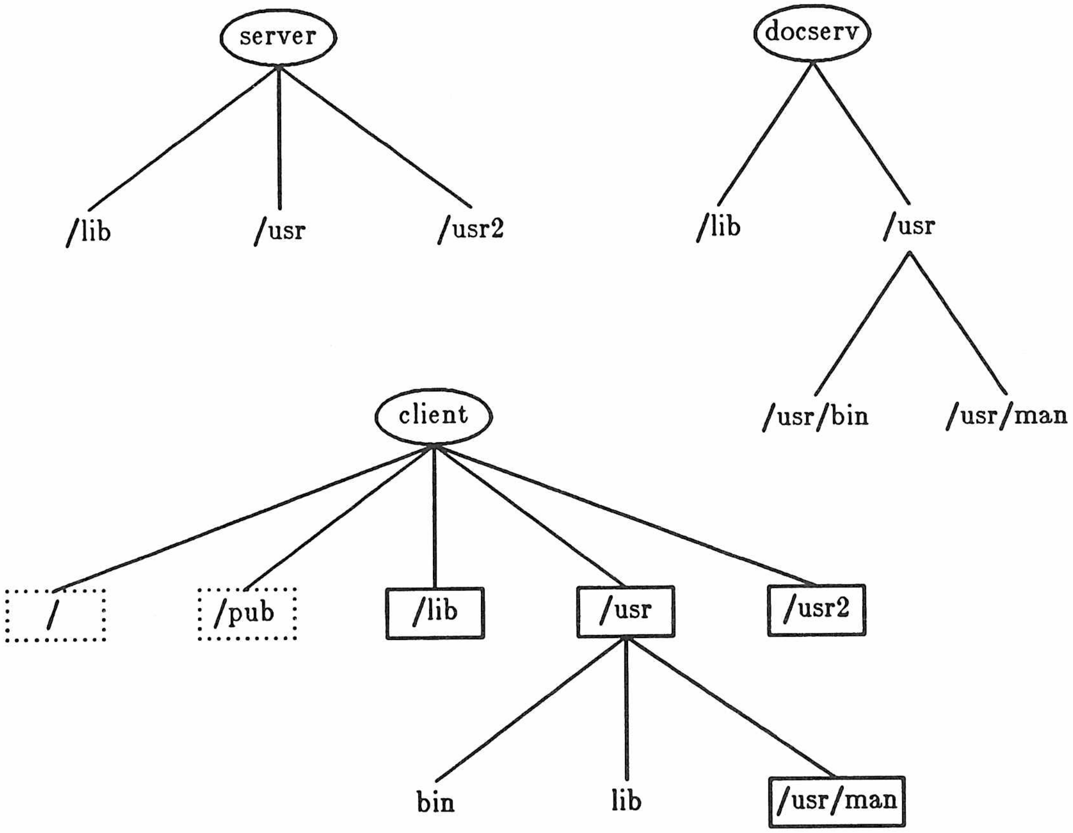
Suppose that you want to read some on-line manual pages. These pages are not available on the server machine, called **server**, but are available on a machine called **docserv**. You can mount the directory containing the manuals as follows:

```
client#  /etc/mount docserv:/usr/man /usr/man
```

Note that you have to be superuser in order to do this. Now you can use the **man** command whenever you want. Try running the df command after you've mounted the remote filesystem. Its output will look something like this:

```
Filesystem            kbytes    used    avail  capacity  Mounted on
/dev/nd0                4775    2765     1532     64%    /
/dev/ndp0               5695    3666     1459     72%    /pub
server:/lib             7295    4137     2428     63%    /lib
server:/usr            39315   31451     3932     89%    /usr
server:/usr2          326215  245993    47600     84%    /usr2
docserv:/usr/man      346111  216894    94605     70%    /usr/man
```

Here is a diagram of the three machines involved here. Ellipses represent machines, boxes represent remote filesystems, and dotted boxes represent ND partitions.



## 2.2. Exporting a Filesystem

Suppose that you and a colleague need to work together on a programming project. The source code is on your machine, in the directory */usr/proj*. It does not matter whether your workstation is a diskless node, or has local disk. Suppose that after creating the proper directory, your colleague tried to remote mount your directory. Unless you have explicitly exported the directory, your colleague's remote mount will fail with a "permission denied" message.

To export a directory, become superuser, and edit the file */etc/exports*. If your colleague is on a machine named `cohort`, then you need to put this one line in */etc/exports*:

```
/usr/proj        cohort
```

Without the keyword `cohort`, anybody on the network could remote mount your directory `/usr/proj`. The NFS mount request server ***mountd***(8c) will read the */etc/exports* file if necessary whenever it receives a request for a remote mount. Now your colleague can remote mount the source directory by issuing this command:

```
cohort#  /etc/mount client:/usr/proj /usr/proj
```

Since both you and your colleague will be able to change files on */usr/proj*, it would be best to use the *sccs*(1) source code control system for concurrency control.

## 2.3. Administering a Server Machine

System administrators must know how to set up the NFS server machine so that client workstations can mount all the necessary filesystems. You export filesystems (that is, make them available) by placing appropriate lines in the */etc/exports* file. Here is a sample */etc/exports* file for a typical server machine:

```
/
/usr
/usr2
/usr/src        staff
```

The pathnames specified in */etc/exports* must be real filesystems — that is, directory mount points for disk devices. The root filesystem must be exported so that */lib* is available to NFS clients. A netgroup, such as `staff`, may be specified after the filesystem, in which case remote mounts are limited to machines that are a member of this netgroup. At any one time, the system administrator can see which filesystems have been remote mounted, by executing the ***showmount***(8) command.

# 3. Architecture of NFS

## 3.1. Design Goals

### *3.1.1. Transparent Information Access*

Users are able to get directly to the files they want without knowing the network address of the data. To the user, all universes look alike: there seems to be no difference between reading or writing a file contained on a private disk, and reading or writing a file on a disk in the next building. Information on the network is truly distributed.

### *3.1.2. Different Machines and Operating Systems*

No single vendor can supply tools for all the work that needs to get done, so appropriate services must be integrated on a network. In keeping with its policy of supplying open systems, Sun is promoting the NFS as a standard for the exchange of data between different machines and operating systems.

### *3.1.3. Easily Extensible*

A distributed system must have an architecture that allows integration of new software technologies without disturbing the extant software environment. To allow this, the NFS provides network services, rather than a new network operating system. That is, the NFS does not depend on extending the underlying operating system onto the network, but instead offers a set of protocols for data exchange. These protocols can be easily extended.

### *3.1.4. Easy Network Administration*

The administration of large networks can be complicated and time-consuming. Sun wishes to make sure that a set of network filesystems is no more difficult to administer than a set of local filesystems on a timesharing system. UNIX has a convenient set of maintenance commands developed over the years. Some new utilities are provided for network administration, but most of the old utilities have been retained.

The Yellow Pages (YP) facility is the first example of a network service made possible with NFS. By storing password information and host addresses in a centralized database, the yellow pages ease the task of network administration. An overview of the YP facility is presented in the ***Network Services Guide***.

The most obvious use of the YP is for administration of ***/etc/passwd***. Since the NFS uses a UNIX protection scheme across the network, it is advantageous to have a common ***/etc/passwd*** database for all machines on the network. The YP allows a single point of administration, and gives all machines access to a recent version of the data, whether or not it is held locally. To install the YP version of ***/etc/passwd***, existing applications were not changed; they were simply relinked with library routines that know about the YP service. Conventions have been added to library routines that access ***/etc/passwd*** to allow each client to administer its own local subset of ***/etc/passwd***; the local subset modifies the client's view of the system version. Thus, a client is not forced to completely bypass the system administrator in order to accomplish a small amount of personalization.

The YP interface is implemented using RPC and XDR, so the service is available to non-UNIX operating systems and non-Sun machines. YP servers do not interpret data, so it is possible for new databases to take advantage of the YP service without modifying the servers.

### *3.1.5. Reliable*

Reliability of the UNIX-based filesystem derives primarily from the robustness of the 4.2BSD filesystem. In addition, the file server protocol is designed so that client workstations can continue to operate even when the server crashes and reboots. This property is shared with the current ND protocol, and has proven to be quite desirable. Sun achieves continuation after reboot without making assumptions about the fail-stop nature of the underlying server hardware.

The major advantage of a stateless server is robustness in the face of client, server, or network failures. Should a client fail, it is not necessary for a server (or human administrator) to take any action to continue normal operation. Should a server or the network fail, it is only necessary that clients continue to attempt to complete NFS operations until the server or network gets fixed. This robustness is especially important in a complex network of heterogeneous systems, many of which are not under the control of a disciplined operations staff, and which may be running untested systems often rebooted without warning.

### *3.1.6. High Performance*

The flexibility of the NFS allows configuration for a variety of cost and performance trade-offs. For example, configuring servers with large, high-performance disks, and clients with no disks, may yield better performance at lower cost than having many machines with small, inexpensive disks. Furthermore, it is possible to distribute the filesystem data across many servers and get the added benefit of multiprocessing without losing transparency. In the case of read-only files, copies can be kept on several servers to avoid bottlenecks.

Sun has also added several performance enhancements to the NFS, such as "fast paths" to eliminate the work done for high-runner operations, asynchronous service of multiple requests, cacheing of disk blocks, and asynchronous read-ahead and write-behind. The fact that cacheing and read-ahead occur on both client and server effectively increases the cache size and read-ahead distance. Cacheing and read-ahead do not add state to the server; nothing (except performance) is losi if cached information is thrown away. In the case of write-behind, both the client and server attempt to flush critical information to disk whenever necessary, to reduce the impact of an unanticipated failure; clients do not free write-behind blocks until the server verifies that the data is written.

Our performance goal was to achieve the same throughput as a previous release of the system that used the network only as a disk (and thus did not permit sharing). This goal has been achieved.

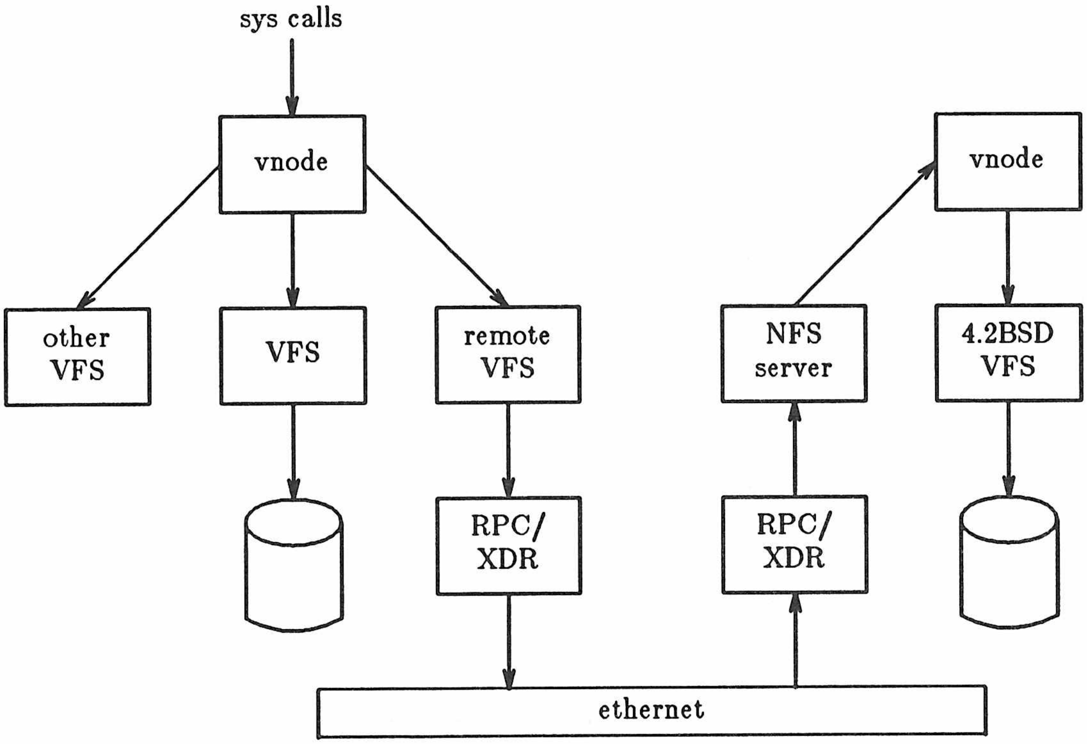
## 3.2. The NFS Implementation

In the Sun implementation of the NFS, there are three entities to be considered: the operating system interface, the virtual file system (VFS) interface, and the network file system (NFS) interface. The UNIX operating system interface has been preserved in the Sun implementation of the NFS, thereby insuring compatibility for existing applications.

***Vnodes*** are a re-implementation of ***inodes*** that cleanly separate filesystem operations from the semantics of their implementation. Above the VFS interface, the operating system deals in ***vnodes***; below this interface, the filesystem may or may not implement ***inodes***. The VFS interface can connect the operating system to a variety of filesystems (for example, 4.2 BSD or MS-DOS). A local VFS connects to filesystem data on a local device.

The remote VFS defines and implements the NFS interface, using the remote procedure call (RPC) mechanism. RPC allows communication with remote services in a manner similar to the procedure calling mechanism available in many programming languages. The RPC protocols are described using the external data representation (XDR) package. XDR permits a machine-independent representation and definition of high-level protocols on the network.

The figure below shows the flow of a request from a client (at the top left) to a collection of filesystems.



In the case of access through a local VFS, requests are directed to filesystem data on devices connected to the client machine. In the case of access through a remote VFS, the request is passed through the RPC and XDR layers onto the net. In the current implementation, Sun uses the UDP/IP protocols and the Ethernet. On the server side, requests are passed through the RPC and XDR layers to an NFS server; the server uses ***vnodes*** to access one of its local VFSs and service the request. This path is retraced to return results.

Sun's implementation of the NFS provides five types of transparency:

1. *Filesystem Type:* The ***vnode***, in conjunction with one or more local VFSs (and possibly remote VFSs) permits an operating system (hence client and application) to interface transparently to a variety of filesystem types.
2. *Filesystem Location:* Since there is no differentiation between a local and a remote VFS, the location of filesystem data is transparent.
3. *Operating System Type:* The RPC mechanism allows interconnection of a variety of operating systems on the network, and makes the operating system type of a remote server transparent.
4. *Machine Type:* The XDR definition facility allows a variety of machines to communicate on the network and makes the machine type of a remote server transparent.
5. *Network Type:* RPC and XDR can be implemented for a variety of network and internet protocols, thereby making the network type-transparent.

Simpler NFS implementations are possible at the expense of some advantages of the Sun version. In particular, a client (or server) may be added to the network by implementing one side of the NFS interface. An advantage of the Sun implementation is that the client and server sides are identical; thus, it is possible for any machine to be client, server or both. Users at client machines with disks can arrange to share over the NFS without having to appeal to a system administrator, or configure a different system on their workstation.

## 3.3. The NFS Interface

As mentioned in the preceding section, a major advantage of the NFS is the ability to mix filesystems. In keeping with this, Sun encourages other vendors to develop products to interface with Sun network services. RPC and XDR have been placed in the public domain, and serve as a standard for anyone wishing to develop applications for the network. Furthermore, the NFS interface itself is open and can be used by anyone wishing to implement an NFS client or server for the network.

The NFS interface defines traditional filesystem operations for reading directories, creating and destroying files, reading and writing files, and reading and setting file attributes. The interface is designed so that file operations address files with an uninterpreted identifier, starting byte address, and length in bytes.

Commands are provided for NFS servers to initiate service (***mountd***), and to serve a portion of their filesystem to the network (***/etc/exports***). Many commands are provided for constructing the YP database facility. A client builds its view of the filesystems available on the network with the ***mount*** command.

The NFS interface is defined so that a server can be ***stateless***. This means that a server does not have to remember from one transaction to the next anything about its clients, transactions completed or files operated on. For example, there is no ***open*** operation, as this would imply state in the server; of course, the UNIX interface uses an ***open*** operation, but the information in the UNIX operation is remembered by the client for use in later NFS operations.

An interesting problem occurs when a UNIX application ***unlinks*** an open file. This is done to achieve the effect of a temporary file that is automatically removed when the application terminates. If the file in question is served by the NFS, the ***unlink*** will remove the file, since the server does not remember that the file is open. Thus, subsequent operations on the file will fail.

In order to avoid state on the server, the client operating system detects the situation, renames the file rather than unlinking it, and ***unlink*s** the file when the application terminates. In certain failure cases, this leaves unwanted "temporary" files on the server; these files are removed as a part of periodic filesystem maintenance.

Another example of how the NFS provides a friendly interface to UNIX without introducing state is the ***mount*** command. A UNIX client of the NFS "builds" its view of the filesystem on its local devices using the ***mount*** command; thus, it is natural for the UNIX client to initiate its contact with the NFS and build its view of the filesystem on the network via an extended ***mount*** command. This ***mount*** command does not imply state in the server, since it only acquires information for the client to establish contact with a server. The ***mount*** command may be issued at any time, but is typically executed as a part of client initialization. The corresponding ***unmount*** command (which replaces the UNIX ***umount***) is only an informative message to the server, but it does change state in the client by modifying its view of the filesystem on the network.

The major advantage of a stateless server is robustness in the face of client, server or network failures. Should a client fail, it is not necessary for a server (or human administrator) to take any action to continue normal operation. Should a server or the network fail, it is only necessary that clients continue to attempt to complete NFS operations until the server or network is fixed. This robustness is especially important in a complex network of heterogeneous systems, many of which are not under the control of a disciplined operations staff and may be running untested systems and/or may be rebooted without warning.

An NFS server can be a client of another NFS server. However, a server will not act as an intermediary between a client and another server. Instead, a client may ask what remote mounts the server has and then attempt to make similar remote mounts. The decision to disallow intermediary servers is based on several factors. First, the existence of an intermediary will impact the performance characteristics of the system; the potential performance implications are so complex that it seems best to require direct communication between a client and server. Second, the existence of an intermediary complicates access control; it is much simpler to require a client and server to establish direct agreements for service. Finally, disallowing intermediaries prevents cycles in the service arrangements; Sun prefers this to detection or avoidance schemes.

The NFS currently implements UNIX file protection by making use of the authentication mechanisms built into RPC. This retains transparency for clients and applications that make use of UNIX file protection. Although the RPC definition allows other authentication schemes, their use may have adverse effects on transparency.

Although the NFS is UNIX-friendly, it does not support all UNIX filesystem operations. For example, the "special file" abstraction of devices is not supported for remote filesystems because it is felt that the interface to devices would greatly complicate the NFS interface; instead, devices are implemented in a local ***/dev*** VFS. Other incompatibilities are due to the fact that NFS servers are stateless. For example, file locking and guaranteed APPEND_MODE are not supported in the remote case.

Our decision to omit certain features from the NFS is motivated by a desire to preserve the stateless implementation of servers and to define a simple, general interface to be implemented and used by a wide variety of customers. The availability of open RPC and NFS interfaces means that customers and users who need stateful or complex features can implement them "beside" or "within" the NFS. Sun is considering implementation of a set of tools for use by applications that need file or record locking, replicated data, or other features implying state and/or distributed synchronization; however, these will not be made part of the base NFS definition.

# 4. Network Documentation Roadmap

The document *Network Services Guide* is intended for users who have a general interest in network services. It explains the yellow pages facility in some detail. Although it is not a manual for system administrators, the material is heavily slanted in that direction.

The document *Remote Procedure Call Programming Guide* is intended for programmers who wish to write network applications using remote procedure calls, thus avoiding low-level system primitives based on sockets. Readers must be familiar with the C programming language, and should have a working knowledge of network theory.

The document *External Data Representation Protocol Specification* is intended for programmers writing complicated applications using remote procedure calls, who need to pass complicated data across the network. It is also a reference guide for system programmers implementing Sun's Network File System on new machines.

The document *Remote Procedure Call Protocol Specification* is a reference guide for system programmers implementing Sun's Network File System on new machines. It is of little interest to programmers writing network applications.

The document *Network File System Protocol Specification* is a reference guide for system programmers implementing Sun's Network File System on new machines. It is of little interest to programmers writing network applications.

The document *Yellow Pages Protocol Specification* is a reference guide for system programmers implementing a Yellow Pages database facility on new machines. It is of little interest to programmers writing network applications.

The document *Inter-Process Communications Primer*, taken from Berkeley's 4.2 release, is for system programmers who need to use low-level networking primitives based on sockets. Since remote procedure calls are easier to use than sockets, this primer is of little interest to most network programmers.

The document *Network Implementation* describes the low-level networking primitives in the 4.2 UNIX kernel. It is of interest primarily to system programmers and aspiring UNIX gurus.

# Network Services Guide

# Contents

# Network Services Guide

## 1. Introduction

This document is intended for users who have a general interest in network services. Although this is not a manual for system administrators, the material is heavily slanted in that direction.

Sun provides several network services, such as Network Disk (ND), and the Network File System (NFS), discussed in the document ***Sun's Network File System***. The yellow pages are another network service offered for the first time on the 2.0 release. They permit password information and host addresses for an entire network to be held in a single database. This greatly eases the task of system and network administration. Sun will provide more network services in the future.

## 2. What Are The Yellow Pages?

The yellow pages (YP) constitute a distributed network lookup service:

- YP is a lookup service: it maintains a set of databases for querying. Programs can ask for the value associated with a particular key, or all the keys, in a database.
- YP is a network service: programs need not know the location of data, or how it is stored. Instead, they use a network protocol to communicate with a database server that knows those details.
- YP is distributed: databases are fully replicated on several machines, known as YP servers. Servers propagate updated databases among themselves, ensuring consistency. At steady state, it doesn't matter which server answers a request; the answer is the same everywhere.

### 2.1. The YP Map

The yellow pages serve information stored in YP ***maps***. Each map contains a set of keys and associated values. For example, the ***hosts*** map contains (as keys) all host names on a network, and (as values) the corresponding Internet addresses. Each YP map has a ***mapname***, used by programs to access data in the map. Programs must know the format of the data in the map. Currently, most maps are derived from ASCII files formerly found in ***/etc***: ***passwd***, ***group***, ***hosts***, ***networks***, and others. The format of data in the YP map is in most cases identical to the format of the ASCII file. Maps are implemented by ***dbm***(3) files located in subdirectories of ***/etc/yp*** on YP server machines.

### 2.2. The YP Domain

A YP ***domain*** is a named set of YP maps. You can determine your YP domain by executing the ***domainname***(1) command. Note that YP domains are different from both Internet domains and ***sendmail*** domains. A YP domain is simply a directory in ***/etc/yp*** containing a set of maps.

A domain name is required for retrieving data from a YP database. For instance, if your YP domain is ***sun*** and you want to find the Internet address of host ***dbserver***, you must ask YP for the value associated with the key ***dbserver*** in the map ***hosts.byname*** within the YP domain ***sun***. Each machine on the network belongs to a default domain, set in ***/etc/rc.local*** at boot time with the ***domainname***(8) command.

A YP server holds all the maps of a YP domain in a subdirectory of ***/etc/yp***, named after the domain. In the example above, maps for the ***sun*** domain would be held in ***/etc/yp/sun***. Every YP server must have the directory ***/etc/yp/yp_private***, which contains information about servers, domains, and maps. This information is used internally by the YP. For completeness, the YP server machine is its own client.

## 2.3. Servers And Clients

Servers provide resources, while clients consume them. A server or a client is not necessarily the same thing as a machine. To illustrate, let's consider three different services: ND (network disk), the YP, and the NFS (network file system).

ND ND is a method of providing virtual disk, used by diskless nodes. With ND, it makes sense to speak of server and client machines, since both provider and consumer are coterminous with machines. Furthermore, the server and client are always the same.

NFS The NFS allows client machines to mount remote filesystems and access files in place, provided a server machine has exported the filesystem. However, a server that exports filesystems may also mount remote filesystems exported by other machines, thus becoming a client. So a given machine may be both server and client, or client only, or server only. Furthermore, NFS servers and clients need not coincide with ND servers and clients.

YP The YP server, by contrast, is a process rather than a machine, running on a machine that may be neither ND server nor NFS server. All processes that make use of YP services are YP clients. Sometimes clients are served by YP servers on the same machine, but other times by YP servers running on another machine. To further muddy the waters, processes on master YP server machines (discussed below) don't use YP services at all, and aren't YP clients. But processes using YP services on slave YP servers are YP clients.

## 2.4. Masters and Slaves

YP servers are either master or slave. For any map, one YP server is designated the master, and all changes to the map should be made on that machine. The changes propagate from master to slaves. A newly built map is timestamped internally when ***makedbm*** creates it. If you build a YP map on a slave server, you will break the YP update algorithm (temporarily), and you will have to get all versions in synch manually. Moral: after you decide which server is the master, do all database updates and builds there, not on slaves.

It is possible for different maps to have different servers as master. Therefore, a given server may be a master with regard to one map, and a slave with regard to another. This can get confusing quickly. It is suggested that a single server be master for all the maps created by ***ypinit*** in a single domain. This document assumes the simple case, in which one server is the master for all maps in the database.

# 3. Overview of the Yellow Pages

In releases before 2.0, each machine on the network had its own copy of */etc/hosts*, a file containing the Internet address of each machine on the network. Every time a machine was added to the network, each */etc/hosts* file had to be updated.

The YP is a network service containing network-wide databases such as */etc/hosts*. There are servers spread throughout the network containing copies of the databases. When an arbitrary machine on the network wants to look up something in */etc/hosts*, it makes an RPC call to one of the servers to get the information. One server is the master — the only one whose database may be modified. The other servers are slaves, and they are periodically updated so that their information is in synch with that of the master.

The YP can serve up any number of databases. Normally that will include files that previously lived in */etc*, such as */etc/hosts* and */etc/networks*. However, users can add their own databases to the YP.

The YP itself simply serves up information, and has no idea what it means. Thus there are two parts of YP we need to consider: how it operates, and what files formerly in */etc* now live in the YP. This has serious ramifications for users.

## 3.1. The YP Network Service

### *3.1.1. Naming*

Imagine a company with two different networks, each of which has its own separate list of hosts and passwords. Within each network, user names, numerical user IDs, and host names are unique. However, there is duplication between the two networks. If these two networks are ever connected, chaos could result. The host name, returned by the ***hostname***(1) command and the ***gethostname***() system call, may no longer uniquely identify a machine. Thus a new command and system call, ***domainname***(1) and ***getdomainname(2)*** have been added. In the example above, each of the two networks could be given a different domain name. However, it is always simpler to use a single domain whenever possible.

The relevance of domains to YP is that data is stored in */etc/yp/* domainname. In particular, a machine can contain data for several different domains.

### *3.1.2. Data Storage*

The data is stored in *dbm*(3) format. Thus the database ***hosts.byname*** for the domain ***sun*** is stored as */etc/yp/sun/hosts.byname.pag* and */etc/yp/sun/hosts.byname.dir*. The command ***makedbm***(8) takes an ASCII file such as */etc/hosts* and converts it into a ***dbm*** file suitable for use by the YP. However, system administrators normally use the makefile in */etc/yp* to create new ***dbm*** files (read on for details). This makefile in turn calls ***makedbm***.

### *3.1.3. Servers*

To become a server, a machine must contain the YP databases, and must also be running the YP daemon ***ypserv***. The ***ypinit***(8) command invokes this daemon automatically. It also takes a flag saying whether you are creating a master or a slave. When updating the master copy of a database, you can force the change to be propagated to all the slaves with the ***yppush***(8) command. This pushes the information out to all the slaves. Conversely, from a slave, the ***yppull***(8) command gets the latest information from the master. The makefile in ***/etc/yp*** first executes ***makedbm*** to make a new database, and then calls ***yppush*** to propagate the change throughout the network.

### *3.1.4. Clients*

Remember that a client machine (which is not a server) does not contain any data itself, but rather makes an RPC call to a YP server each time it needs information from a YP database. The ***ypbind***(8) daemon caches the name of a server. When a client boots, ***ypbind*** broadcasts asking for the name of the YP server. Similarly, if the cached server crashes, ***ypbind*** broadcasts asking for the name of a new server. The ***ypwhich***(1) command gives the name of the server that ***ypbind*** currently points at.

Since client machines no longer have entire copies of files in the YP, a new command ***ypcat***(1) has been provided. The command `ypcat hosts` is equivalent to `cat /etc/hosts` in a pre 2.0 system; as you might guess, `ypcat passwd` is equivalent to `cat /etc/passwd`. To look for someone's password entry, searching through the password file no longer suffices; you have to issue the following command

```
        % ypcat passwd | grep userid
```

where you replace `userid` with the login name you're searching for.

## 3.2. Default YP Files

By default, Sun workstations have six files from */etc* in the YP: */etc/passwd*, */etc/groups*, */etc/networks*, */etc/hosts*, */etc/services*, and */etc/protocols*. In addition, there is a new file *netgroup*, which many sites ought to create and put in the YP database.

Library routines such as ,***getpwent***(3) ,***getgrent***(3) and ***gethostent***(3) have been rewritten to take advantage of the YP. Thus, C programs that call these library routines will have to be relinked in order to function correctly.

### *3.2.1. Hosts*

The hosts file is stored as two different files in the YP. The first, ***hosts.byname***, is indexed by hostname. The second, ***hosts.byaddr***, is indexed by Internet address. Remember that this actually expands into four files, with suffixes *.pag*, and *.dir*. When a user program calls the library routine ***gethostbyname***(3), a single RPC call to a server retrieves the entry from the ***hosts.byname*** file. Similarly, ***gethostbyaddr***(3) retrieves the entry from the ***hosts.byaddr*** file. Of course if the YP is not running (which is caused by commenting ***ypbind*** out of the ***/etc/rc*** file), then ***gethostbyname*** will read the ***/etc/hosts*** files, just as it always has.

Although the *ypcat* command is a general YP database print program, it knows about the standard files in the YP. Thus `ypcat hosts` is translated into `ypcat hosts.byaddr`, since there is no file called *hosts* in the YP.

Normally, the hosts file for the YP will be the same as the */etc/hosts* file on the machine serving as a YP master. In this case, the makefile in */etc/yp* will check to see if */etc/hosts* is newer than the *dbm* file. If it is, it will use a simple *sed* script to recreate *hosts.byname* and *hosts.byaddr*, run them through *makedbm*(8) and then call *yppush*(8). See *ypmake*(8) for details.

### *3.2.2. Passwd*

The *passwd* file is similar to the *hosts* file. It exists as two separate files, *passwd.byname* and *passwd.byuid*. The *ypcat* program prints it, and *ypmake* updates it. However, if *getpwent*(3) always went directly to the YP as does *gethostent*(3), then everyone would be forced to have an identical password file! Consequently, *getpwent* reads the local */etc/passwd* file, just as it always did. But now it interprets "+" entries in the password file to mean, interpolate entries from the YP database. If you wrote a simple program using *getpwent* to print out all the entries from your password file, it would print out a virtual password file: rather than printing out + signs, it would print out whatever entries the local password file included from the YP database. The difference between */etc/hosts* and */etc/passwd* is discussed in more detail in the section "How Security is Changed with the Yellow Pages," part of the *System Administrator's Manual*.

### *3.2.3. Others*

Of the other four files in */etc*, */etc/group* is treated like */etc/passwd*, in that *getgrent()* will only consult the YP if explicitly told to do so by the */etc/group* file. The files */etc/networks*, */etc/protocols*, */etc/services*, and */etc/networks* are treated like */etc/hosts*: for these files, the library routines go directly to the YP, without consulting the local files.

### *3.2.4. Changing your passwd*

To change data in the YP, you must log onto the master machine, and edit databases there; *ypwhich*(1) tells where the master server is. However, since changing a password is so commonly done, the *yppasswd*(1) command has been provided to change your YP password. It has the same user interface as the *passwd*(1) command. This command will only work if the *yppasswdd*(8c) server has been started up on the YP master server machine.

# Remote Procedure Call
# Programming Guide

# Contents

# Remote Procedure Call Programming Guide

## 1. Introduction

This document is intended for programmers who wish to write network applications using remote procedure calls (explained below), thus avoiding low-level system primitives based on sockets. The reader must be familiar with the C programming language, and should have a working knowledge of network theory.

Programs that communicate over a network need a paradigm for communication. A low-level mechanism might send a signal on the arrival of incoming packets, causing a network signal handler to execute. A high-level mechanism would be the Ada `rendezvous`. The method used at Sun is the Remote Procedure Call (RPC) paradigm, in which a client communicates with a server. In this process, the client first calls a procedure to send a data packet to the server. When the packet arrives, the server calls a dispatch routine, performs whatever service is requested, sends back the reply, and the procedure call returns to the client.

The RPC interface is divided into three layers. The highest layer is totally transparent to the programmer. To illustrate, at this level a program can contain a call to `rnusers()`, which returns the number of users on a remote machine. You don't have to be aware that RPC is being used, since you simply make the call in a program, just as you would call `malloc()`.

At the middle layer, the routines `registerrpc()` and `callrpc()` are used to make RPC calls: `registerrpc()` obtains a unique system-wide number, while `callrpc()` executes a remote procedure call. The `rnusers()` call is implemented using these two routines The middle-layer routines are designed for most common applications, and shield the user from knowing about sockets.

The lowest layer is used for more sophisticated applications, which may want to alter the defaults of the routines. At this layer, you can explicitly manipulate sockets used for transmitting RPC messages. This level should be avoided if possible.

Section 2 of this manual illustrates use of the highest two layers while Section 3 presents the low-level interface. Section 4 of the manual discusses miscellaneous topics. The final section summarizes all the entry points into the RPC system.

Although this document only discusses the interface to C, remote procedure calls can be made from any language. Even though this document discusses RPC when it is used to communicate between processes on different machines, it works just as well for communication between different processes on the same machine.

There is a diagram of the RPC paradigm on the next page.

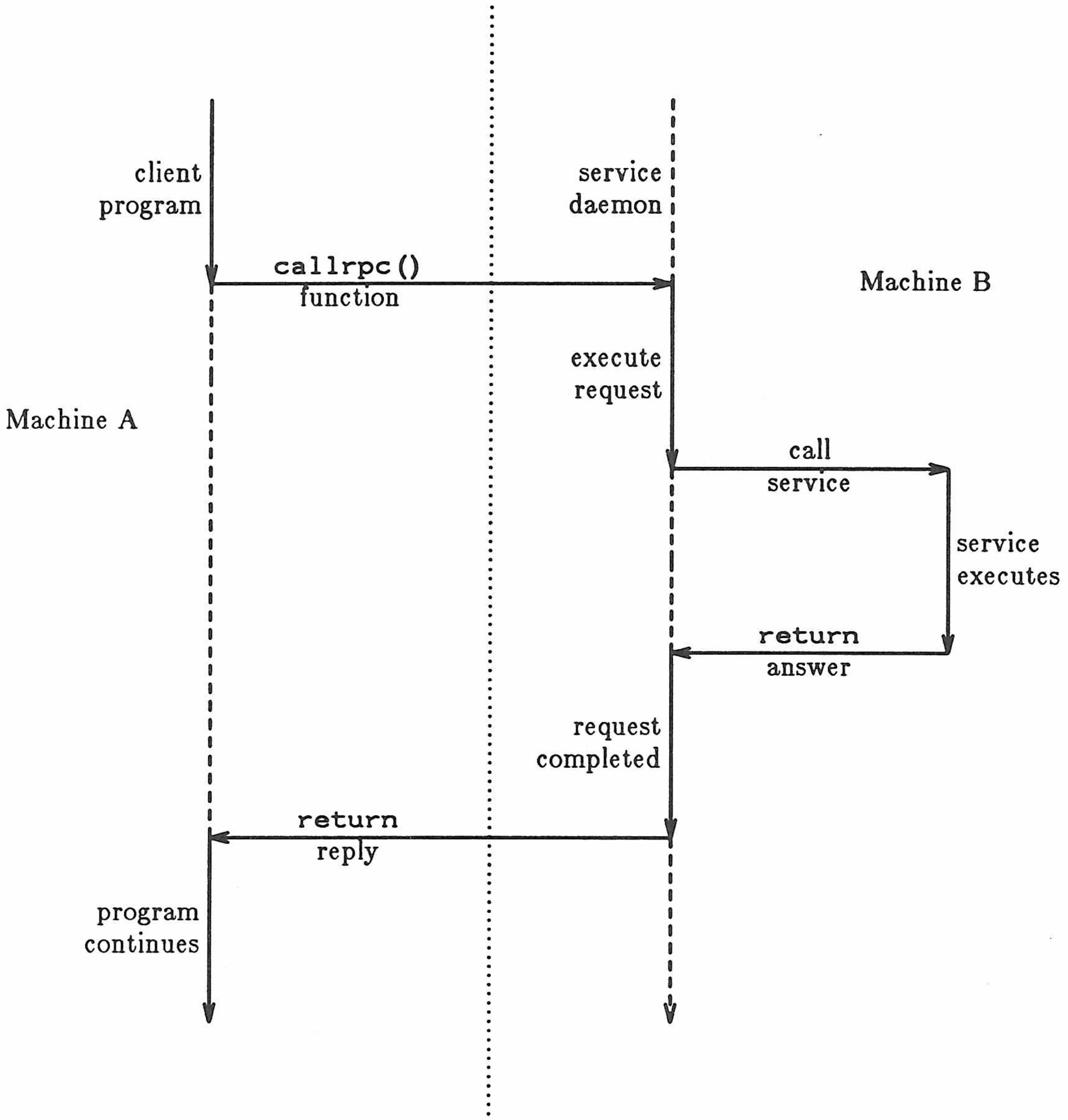


Figure 1: Network Communication with the Remote Procedure Call

# 2. Introductory Examples

## 2.1. Highest Layer

Imagine you're writing a program that needs to know how many users are logged into a remote machine. You can do this by calling the library routine **rnusers()**, as illustrated below:

```
#include <stdio.h>

main(argc, argv)
        int argc;
        char **argv;
{
        unsigned num;

        if (argc < 2) {
                fprintf(stderr, "usage: rnusers hostname\n");
                exit(1);
        }
        if ((num = rnusers(argv[1])) < 0) {
                fprintf(stderr, "error: rnusers\n");
                exit(-1);
        }
        printf("%d users on %s\n", num, argv[1]);
        exit(0);
}
```

RPC library routines such as **rnusers()** are included in the C library `libc.a`. Thus, the program above could be compiled with

```
% cc program.c
```

Some other library routines are **rstat()** to gather remote performance statistics, and **ypmatch()** to glean information from the yellow pages (YP). The YP library routines are documented on the manual page *ypclnt*(3N).

## 2.2. Intermediate Layer

The simplest interface, which explicitly makes RPC calls, uses the functions `callrpc()` and `registerrpc()`. Using this method, another way to get the number of remote users is:

```
#include <stdio.h>
#include <rpcsvc/rusers.h>

main(argc, argv)
        int argc;
        char **argv;
{
        unsigned long nusers;

        if (argc < 2) {
                fprintf(stderr, "usage: nusers hostname\n");
                exit(-1);
        }
        if (callrpc(argv[1], RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
            xdr_void, 0, xdr_u_long, &nusers) != 0) {
                fprintf(stderr, "error: callrpc\n");
                exit(1);
        }
        printf("number of users on %s is %d\n", argv[1], nusers);
        exit(0);
}
```

A program number, version number, and procedure number defines each RPC procedure. The program number defines a group of related remote procedures, each of which has a different procedure number. Each program also has a version number, so when a minor change is made to a remote service (adding a new procedure, for example), a new program number doesn't have to be assigned. When you want to call a procedure to find the number of remote users, you look up the appropriate program, version and procedure numbers in a manual, similar to when you look up the name of memory allocator when you want to allocate memory.

The simplest routine in the RPC library used to make remote procedure calls is `callrpc()`. It has eight parameters. The first is the name of the remote machine. The next three parameters are the program, version, and procedure numbers. The following two parameters define the argument of the RPC call, and the final two parameters are for the return value of the call. If it completes successfully, `callrpc()` returns zero, but nonzero otherwise. The exact meaning of the return codes is found in `<rpc/clnt.h>`, and is in fact an `enum clnt_stat` cast into an integer.

Since data types may be represented differently on different machines, `callrpc()` needs both the type of the RPC argument, as well as a pointer to the argument itself (and similarly for the result). For RUSERSPROC_NUM, the return value is an `unsigned long`, so `callrpc()` has `xdr_u_long` as its first return parameter, which says that the result is of type `unsigned long`, and `&nusers` as its second return parameter, which is a pointer to where the long result will be placed. Since RUSERSPROC_NUM takes no argument, the argument parameter of `callrpc()` is `xdr_void`.

After trying several times to deliver a message, if `callrpc()` gets no answer, it returns with an error code. The delivery mechanism is UDP, which stands for User Datagram Protocol. Methods for adjusting the number of retries or for using a different protocol require you to use the lower layer of the RPC library, discussed later in this document. The remote server procedure corresponding to the above might look like this:

```
char *
nuser(indata)
        char *indata;
{
        static int nusers;

        /*
         * code here to compute the number of users
         * and place result in variable nusers
         */
        return ((char *)&nusers);
}
```

It takes one argument, which is a pointer to the input of the remote procedure call (ignored in our example), and it returns a pointer to the result. In the current version of C, character pointers are the generic pointers, so both the input argument and the return value are cast to `char *`.

Normally, a server registers all of the RPC calls it plans to handle, and then goes into an infinite loop waiting to service requests. In this example, there is only a single procedure to register, so the main body of the server would look like this:

```
#include <stdio.h>
#include <rpcsvc/rusers.h>

char *nuser();

main()
{
        registerrpc(RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM, nuser,
            xdr_void, xdr_u_long);
        svc_run();              /* never returns */
        fprintf(stderr, "Error: svc_run returned!\n");
        exit(1);
}
```

The `registerrpc()` routine establishes what C procedure corresponds to each RPC procedure number. The first three parameters, RUSERPROG, RUSERSVERS, and RUSERSPROC_NUM are the program, version, and procedure numbers of the remote procedure to be registered; `nuser` is the name of the C procedure implementing it; and `xdr_void` and `xdr_u_long` are the types of the input to and output from the procedure.

Only the UDP transport mechanism can use `registerrpc()`; thus, it is always safe in conjunction with calls generated by `callrpc()`.

Warning: the UDP transport mechanism can only deal with arguments and results less than 8K bytes in length.

## 2.3. Assigning Program Numbers

Program numbers are assigned in groups of 0x20000000 (536870912) according to the following chart:

| | |
|---|---|
| 0 - 1fffffff | defined by sun |
| 20000000 - 3fffffff | defined by user |
| 40000000 - 5fffffff | transient |
| 60000000 - 7fffffff | reserved |
| 80000000 - 9fffffff | reserved |
| a0000000 - bfffffff | reserved |
| c0000000 - dfffffff | reserved |
| e0000000 - ffffffff | reserved |

Sun Microsystems administers the first group of numbers, which should be identical for all Sun customers. If a customer develops an application that might be of general interest, that application should be given an assigned number in the first range. The second group of numbers is reserved for specific customer applications. This range is intended primarily for debugging new programs. The third group is reserved for applications that generate program numbers dynamically. The final groups are reserved for future use, and should not be used.

The exact registration process for Sun defined numbers is yet to be established.

## 2.4. Passing Arbitrary Data Types

In the previous example, the RPC call passes a single `unsigned long`. RPC can handle arbitrary data structures, regardless of different machines' byte orders or structure layout conventions, by always converting them to a network standard called ***eXternal Data Representation*** (XDR) before sending them over the wire. The process of converting from a particular machine representation to XDR format is called ***serializing***, and the reverse process is called ***deserializing***. The type field parameters of `callrpc()` and `registerrpc()` can be a built-in procedure like `xdr_u_long()` in the previous example, or a user supplied one. XDR has these built-in type routines:

```
xdr_int()      xdr_u_int()      xdr_enum()
xdr_long()     xdr_u_long()     xdr_bool()
xdr_short()    xdr_u_short()    xdr_string()
```

As an example of a user-defined type routine, if you wanted to send the structure

```
struct simple {
        int a;
        short b;
} simple;
```

then you would call `callrpc` as

```
callrpc(hostname, PROGNUM, VERSNUM, PROCNUM, xdr_simple, &simple ...);
```

where `xdr_simple()` is written as:

```
#include <rpc/rpc.h>

xdr_simple(xdrsp, simplep)
        XDR *xdrsp;
        struct simple *simplep;
{
        if (!xdr_int(xdrsp, &simplep->a))
                return (0);
        if (!xdr_short(xdrsp, &simplep->b))
                return (0);
        return (1);
}
```

An XDR routine returns nonzero (true in the sense of C) if it completes successfully, and zero otherwise. A complete description of XDR is in the *XDR Protocol Specification*, so this section only gives a few examples of XDR implementation.

In addition to the built-in primitives, there are also the prefabricated building blocks:

```
xdr_array()         xdr_bytes()
xdr_reference()     xdr_union()
```

To send a variable array of integers, you might package them up as a structure like this

```
struct varintarr {
        int *data;
        int arrlnth;
} arr;
```

and make an RPC call such as

```
callrpc(hostname, PROGNUM, VERSNUM, PROCNUM, xdr_varintarr, &arr...);
```

with `xdr_varintarr()` defined as:

```
xdr_varintarr(xdrsp, varintarr)
        XDR *xdrsp;
        struct varintarr *arrp;
{
        xdr_array(xdrsp, &arrp->data, &arrp->arrlnth, MAXLEN,
                sizeof(int), xdr_int);
}
```

This routine takes as parameters the XDR handle, a pointer to the array, a pointer to the size of the array, the maximum allowable array size, the size of each array element, and an XDR routine for handling each array element.

If the size of the array is known in advance, then the following could also be used to send out an array of length SIZE:

```
int intarr[SIZE];

xdr_intarr(xdrsp, intarr)
        XDR *xdrsp;
        int intarr[];
{
        int i;

        for (i = 0; i < SIZE; i++) {
                if (!xdr_int(xdrsp, &intarr[i]))
                        return (0);
        }
        return (1);
}
```

XDR always converts quantities to 4-byte multiples when deserializing. Thus, if either of the examples above involved characters instead of integers, each character would occupy 32 bits. That is the reason for the XDR routine **xdr_bytes()**, which is like **xdr_array()** except that it packs characters. It has four parameters, the same as the first four parameters of **xdr_array()**. For null-terminated strings, there is also the **xdr_string()** routine, which is the same as **xdr_bytes()** without the length parameter. On serializing it gets the string length from **strlen()**, and on deserializing it creates a null-terminated string.

Here is a final example that calls the previously written **xdr_simple()** as well as the built-in functions **xdr_string()** and **xdr_reference()**, which chases pointers:

```
struct finalexample {
        char *string;
        struct simple *simplep;
} finalexample;

xdr_finalexample(xdrsp, finalp)
        XDR *xdrsp;
        struct finalexample *finalp;
{
        int i;

        if (!xdr_string(xdrsp, &finalp->string, MAXSTRLEN))
                return (0);
        if (!xdr_reference(xdrsp, &finalp->simplep,
            sizeof(struct simple), xdr_simple);
                return (0);
        return (1);
}
```

# 3. Lower Layers of RPC

In the examples given so far, RPC takes care of many details automatically for you. In this section, we'll show you how you can change the defaults by using lower layers of the RPC library. It is assumed that you are familiar with sockets and the system calls for dealing with them. If not, consult *The IPC Tutorial*.

## 3.1. More on the Server Side

There are a number of assumptions built into `registerrpc()`. One is that you are using the UDP datagram protocol. Another is that you don't want to do anything unusual while deserializing, since the deserialization process happens automatically before the user's server routine is called. The server for the `nusers` program shown below is written using a lower layer of the RPC package, which does not make these assumptions.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>

int nuser();

main()
{
        SVCXPRT *transp;

        transp = svcudp_create(RPC_ANYSOCK);
        if (transp == NULL){
                fprintf(stderr, "couldn't create an RPC server\n");
                exit(1);
        }
        pmap_unset(RUSERSPROG, RUSERSVERS);
        if (!svc_register(transp, RUSERSPROG, RUSERSVERS, nuser,
            IPPROTO_UDP)) {
                fprintf(stderr, "couldn't register RUSER service\n");
                exit(1);
        }
        svc_run();  /* never returns */
        fprintf(stderr, "should never reach this point\n");
}
```

```
nuser(rqstp, tranp)
        struct svc_req *rqstp;
        SVCXPRT *transp;
{
        unsigned long nusers;

        switch (rqstp->rq_proc) {
        case NULLPROC:
                if (!svc_sendreply(transp, xdr_void, 0)) {
                        fprintf(stderr, "couldn't reply to RPC call\n");
                        exit(1);
                }
                return;
        case RUSERSPROC_NUM:
                /*
                 * code here to compute the number of users
                 * and put in variable nusers
                 */
                if (!svc_sendreply(transp, xdr_u_long, &nusers) {
                        fprintf(stderr, "couldn't reply to RPC call\n");
                        exit(1);
                }
                return;
        default:
                svcerr_noproc(transp);
                return;
        }
}
```

First, the server gets a transport handle, which is used for sending out RPC messages. registerrpc() uses svcudp_create() to get a UDP handle. If you require a reliable protocol, call svctcp_create() instead. If the argument to svcudp_create() is RPC_ANYSOCK, the RPC library creates a socket on which to send out RPC calls. Otherwise, svcudp_create() expects its argument to be a valid socket number. If you specify your own socket, it can be bound or unbound. If it is bound to a port by the user, the port numbers of svcudp_create() and clntudp_create() (the low-level client routine) must match.

When the user specifies RPC_ANYSOCK for a socket or gives an unbound socket, the system determines port numbers in the following way: when a server starts up, it advertises to a port mapper demon on its local machine, which picks a port number for the RPC procedure if the socket specified to svcudp_create() isn't already bound. When the clntudp_create() call is made with an unbound socket, the system queries the port mapper on the machine to which the call is being made, and gets the appropriate port number. If the port mapper is not running or has no port corresponding to the RPC call, the RPC call fails. Users can make RPC calls to the port mapper themselves. The appropriate procedure numbers are in the include file <rpc/pmap_prot.h>.

After creating an SVCXPRT, the next step is to call pmap_unset() so that if the nusers server crashed earlier, any previous trace of it is erased before restarting. More precisely, pmap_unset() erases the entry for RUSERS from the port mapper's tables.

Finally, we associate the program number for nusers with the procedure nuser(). The final argument to svc_register() is normally the protocol being used, which, in this case, is IPPROTO_UDP. Notice that unlike registerrpc(), there are no XDR routines involved in the registration process. Also, registration is done on the program, rather than procedure, level.

The user routine **nuser()** must call and dispatch the appropriate XDR routines based on the procedure number. Note that two things are handled by **nuser()** that **registerrpc()** handles automatically. The first is that procedure NULLPROC (currently zero) returns with no arguments. This can be used as a simple test for detecting if a remote program is running. Second, there is a check for invalid procedure numbers. If one is detected, **svcerr_noproc()** is called to handle the error.

The user service routine serializes the results and returns them to the RPC caller via **svc_sendreply()**. Its first parameter is the SVCXPRT handle, the second is the XDR routine, and the third is a pointer to the data to be returned. Not illustrated above is how a server handles an RPC program that passes data. As an example, we can add a procedure RUSERSPROC_BOOL, which has an argument **nusers**, and returns TRUE or FALSE depending on whether there are nusers logged on. It would look like this:

```
case RUSERSPROC_BOOL: {
        int bool;
        unsigned nuserquery;

        if (!svc_getargs(transp, xdr_u_int, &nuserquery) {
                svcerr_decode(transp);
                return;
        }
        /*
         * code to set nusers = number of users
         */
        if (nuserquery == nusers)
                bool = TRUE;
        else
                bool = FALSE;
        if (!svc_sendreply(transp, xdr_bool, &bool){
                 fprintf(stderr, "couldn't reply to RPC call\n");
                 exit(1);
        }
        return;
}
```

The relevant routine is **svc_getargs()**, which takes an SVCXPRT handle, the XDR routine, and a pointer to where the input is to be placed as arguments.

## 3.2. Memory Allocation with XDR

XDR routines not only do input and output, they also do memory allocation. This is why the second parameter of **xdr_array()** is a pointer to an array, rather than the array itself. If it is NULL, then **xdr_array()** allocates space for the array and returns a pointer to it, putting the size of the array in the third argument. As an example, consider the following XDR routine **xdr_chararr1()**, which deals with a fixed array of bytes with length SIZE:

```
xdr_chararr1(xdrsp, chararr)
        XDR *xdrsp;
        char chararr[];
{
        char *p;
        int len;

        p = chararr;
        len = SIZE;
        return (xdr_bytes(xdrsp, &p, &len, SIZE));
}
```

It might be called from a server like this,

```
char chararr[SIZE];

svc_getargs(transp, xdr_chararr1, chararr);
```

where chararr has already allocated space. If you want XDR to do the allocation, you would have to rewrite this routine in the following way:

```
xdr_chararr2(xdrsp, chararrp)
        XDR *xdrsp;
        char **chararrp;
{
        int len;

        len = SIZE;
        return (xdr_bytes(xdrsp, charrarrp, &len, SIZE));
}
```

Then the RPC call might look like this:

```
char *arrptr;

arrptr = NULL;
svc_getargs(transp, xdr_chararr2, &arrptr);
/*
 * use the result here
 */
svc_freeargs(xdrsp, xdr_chararr2, &arrptr);
```

After using the character array, it can be freed with svc_freeargs(). In the routine xdr_finalexample() given earlier, if finalp->string was NULL in the call

```
svc_getargs(transp, xdr_finalexample, &finalp);
```

then

```
svc_freeargs(xdrsp, xdr_finalexample, &finalp);
```

frees the array allocated to hold finalp->string; otherwise, it frees nothing. The same is true for finalp->simplep.

To summarize, each XDR routine is responsible for serializing, deserializing, and allocating memory. When an XDR routine is called from callrpc(), the serializing part is used. When called from svc_getargs(), the deserializer is used. And when called from svc_freeargs(), the memory deallocator is used. When building simple examples like those in this section, a user doesn't have to worry about the three modes. The XDR reference manual has examples of more sophisticated XDR routines that determine which of the three modes they are in to function correctly.

## 3.3. The Calling Side

When you use `callrpc`, you have no control over the RPC delivery mechanism or the socket used to transport the data. To illustrate the layer of RPC that lets you adjust these parameters, consider the following code to call the **nusers** service:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>

main(argc, argv)
        int argc;
        char **argv;
{
        struct hostent *hp;
        struct timeval pertry_timeout, total_timeout;
        struct sockaddr_in server_addr;
        int addrlen, sock = RPC_ANYSOCK;
        register CLIENT *client;
        enum clnt_stat clnt_stat;
        unsigned long nusers;

        if (argc < 2) {
                fprintf(stderr, "usage: nusers hostname\n");
                exit(-1);
        }
        if ((hp = gethostbyname(argv[1])) == NULL) {
                fprintf(stderr, "cannot get addr for '%s'\n", argv[1]);
                exit(-1);
        }
        pertry_timeout.tv_sec = 3;
        pertry_timeout.tv_usec = 0;
        addrlen = sizeof(struct sockaddr_in);
        bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr, hp->h_length);
        server_addr.sin_family = AF_INET;
        server_addr.sin_port =  0;
        if ((client = clntudp_create(&server_addr, RUSERSPROG,
            RUSERSVERS, pertry_timeout, &sock)) == NULL) {
                perror("clntudp_create");
                exit(-1);
        }
        total_timeout.tv_sec = 20;
        total_timeout.tv_usec = 0;
        clnt_stat = clnt_call(client, RUSERSPROC_NUM, xdr_void, 0,
            xdr_u_long, &nusers, total_timeout);
        if (clnt_stat != RPC_SUCCESS) {
                clnt_perror(client, "rpc");
                exit(-1);
        }
        clnt_destroy(client);
}
```

The low-level version of `callrpc()` is `clnt_call()`, which takes a CLIENT pointer rather than a host name. The parameters to **clnt_call()** are a CLIENT pointer, the procedure number, the XDR routine for serializing the argument, a pointer to the argument, the XDR routine for deserializing the return value, a pointer to where the return value will be placed, and the time in seconds to wait for a reply.

The CLIENT pointer is encoded with the transport mechanism. **callrpc()** uses UDP, thus it calls **clntudp_create()** to get a CLIENT pointer. To get TCP (Transport Control Protocol), you would use **clnttcp_create()**.

The parameters to **clntudp_create()** are the server address, the length of the server address, the program number, the version number, a timeout value (between tries), and a pointer to a socket. The final argument to **clnt_call()** is the total time to wait for a response. Thus, the number of tries is the **clnt_call()** timeout divided by the **clntudp_create()** timeout.

There is one thing to note when using the **clnt_destroy()** call. It deallocates any space associated with the CLIENT handle, but it does not close the socket associated with it, which was passed as an argument to **clntudp_create()**. The reason is that if there are multiple client handles using the same socket, then it is possible to close one handle without destroying the socket that other handles are using.

To make a stream connection, the call to **clntudp_create()** is replaced with a call to **clnttcp_create()**.

```
clnttcp_create(&server_addr, prognum, versnum, &socket, inputsize,
          outputsize);
```

There is no timeout argument; instead, the receive and send buffer sizes must be specified. When the **clnttcp_create()** call is made, a TCP connection is established. All RPC calls using that CLIENT handle would use this connection. The server side of an RPC call using TCP has **svcudp_create()** replaced by **svctcp_create()**.

# 4. Other RPC Features

This section discusses some other aspects of RPC that are occasionally useful.

## 4.1. Select on the Server Side

Suppose a process is processing RPC requests while performing some other activity. If the other activity involves periodically updating a data structure, the process can set an alarm signal before calling svc_run(). But if the other activity involves waiting on a a file descriptor, the svc_run() call won't work. The code for svc_run() is as follows:

```
void
svc_run()
{
        int readfds;

        for (;;) {
                readfds = svc_fds;
                switch (select(32, &readfds, NULL, NULL, NULL)) {

                case -1:
                        if (errno == EINTR)
                                continue;
                        perror("rstat: select");
                        return;
                case 0:
                        break;
                default:
                        svc_getreq(readfds);
                }
        }
}
```

You can bypass svc_run() and call svc_getreq() yourself. All you need to know are the file descriptors of the socket(s) associated with the programs you are waiting on. Thus you can have your own select() that waits on both the RPC socket, and your own descriptors.

## 4.2. Broadcast RPC

The pmap and RPC protocols implement broadcast RPC. Here are the main differences between broadcast RPC and normal RPC calls:

1) Normal RPC expects one answer, whereas broadcast RPC expects many answers (one or more answer from each responding machine).

2) Broadcast RPC can only be supported by packet-oriented (connectionless) transport protocols like UPD/IP.

3) The implementation of broadcast RPC treats all unsuccessful responses as garbage by filtering them out. Thus, if there is a version mismatch between the broadcaster and a remote service, the user of broadcast RPC never knows.

4) All broadcast messages are sent to the portmap port. Thus, only services that register themselves with their portmapper are accessible via the broadcast RPC mechanism.

### *4.2.1. Broadcast RPC Synopsis*

```
#include <rpc/pmap_clnt.h>

enum clnt_stat  clnt_stat;

clnt_stat =
clnt_broadcast(prog, vers, proc, xargs, argsp, xresults, resultsp, eachresult)
u_long          prog;           /* program number */
u_long          vers;           /* version number */
u_long          proc;           /* procedure number */
xdrproc_t       xargs;          /* xdr routine for args */
caddr_t         argsp;          /* pointer to args */
xdrproc_t       xresults;       /* xdr routine for results */
caddr_t         resultsp;       /* pointer to results */
bool_t (*eachresult)();         /* call with each result obtained */
```

The procedure `eachresult()` is called each time a valid result is obtained. It returns a boolean that indicates whether or not the client wants more responses.

```
bool_t                  done;

done =
eachresult(resultsp, raddr)
caddr_t                 resultsp;
struct sockaddr_in      *raddr;   /* address of machine that sent response */
```

If done is TRUE, then broadcasting stops and `clnt_broadcast()` returns successfully. Otherwise, the routine waits for another response. The request is rebroadcast after a few seconds of waiting. If no responses come back, the routine returns with RPC_TIMEDOUT. To interpret `clnt_stat` errors, feed the error code to `clnt_perrno()`.

## 4.3. Batching

The RPC architecture is designed so that clients send a call message, and wait for servers to reply that the call succeeded. This implies that clients do not compute while servers are processing a call. This is inefficient if the client does not want or need an acknowledgement for every message sent. It is possible for clients to continue computing while waiting for a response, using RPC batch facilities.

RPC messages can be placed in a "pipeline" of calls to a desired server; this is called batching. Batching assumes that: 1) each RPC call in the pipeline requires no response from the server, and the server does not send a response message; and 2) the pipeline of calls is transported on a reliable byte stream transport such as TCP/IP. Since the server does not respond to every call, the client can generate new calls in parallel with the server executing previous calls. Furthermore, the TCP/IP implementation can buffer up many call messages, and send them to the server in one `write` system call. This overlapped execution greatly decreases the interprocess communication overhead of the client and server processes, and the total elapsed time of a series of calls.

Since the batched calls are buffered, the client should eventually do a legitimate call in order to flush the pipeline.

A contrived example of batching follows. Assume a string rendering service (like a window system) has two similar calls: one renders a string and returns void results, while the other renders a string and remains silent. The service (using the TCP/IP transport) may look like:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/windows.h>

void windowdispatch();

main()
{
        SVCXPRT *transp;

        transp = svctcp_create(RPC_ANYSOCK, 0, 0);
        if (transp == NULL){
                fprintf(stderr, "couldn't create an RPC server\n");
                exit(1);
        }
        pmap_unset(WINDOWPROG, WINDOWVERS);
        if (!svc_register(transp, WINDOWPROG, WINDOWVERS, windowdispatch,
            IPPROTO_TCP)) {
                fprintf(stderr, "couldn't register WINDOW service\n");
                exit(1);
        }
        svc_run();  /* never returns */
        fprintf(stderr, "should never reach this point\n");
}

void
windowdispatch(rqstp, transp)
        struct svc_req *rqstp;
        SVCXPRT *transp;
{
        char *s = NULL;

        switch (rqstp->rq_proc) {
        case NULLPROC:
                if (!svc_sendreply(transp, xdr_void, 0)) {
                        fprintf(stderr, "couldn't reply to RPC call\n");
                        exit(1);
                }
                return;
        case RENDERSTRING:
                if (!svc_getargs(transp, xdr_wrapstring, &s)) {
                        fprintf(stderr, "couldn't decode arguments\n");
                        svcerr_decode(transp); /* tell caller he screwed up */
                        break;
                }
                /*
                 * call here to to render the string s
                 */
                if (!svc_sendreply(transp, xdr_void, NULL)) {
                        fprintf(stderr, "couldn't reply to RPC call\n");
                        exit(1);
                }
                break;
        case RENDERSTRING_BATCHED:
                if (!svc_getargs(transp, xdr_wrapstring, &s)) {
                        fprintf(stderr, "couldn't decode arguments\n");
                        /*
                         * we are silent in the face of protocol errors
                         */
                        break;
                }
                /*
                 * call here to to render the string s,
                 * but sends no reply!
                 */
                break;
        default:
                svcerr_noproc(transp);
                return;
        }
        /*
         * now free string allocated while decoding arguments
         */
        svc_freeargs(transp, xdr_wrapstring, &s);
}
```

Of course the service could have one procedure that takes the string and a boolean to indicate whether or not the procedure should respond.

In order for a client to take advantage of batching, the client must perform RPC calls on a TCP-based transport and the actual calls must have the following attributes: 1) the result's XDR routine must be zero (NULL), and 2) the RPC call's timeout must be zero.

Here is an example of a client that uses batching to render a bunch of strings; the batching is flushed when the client gets a null string:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/windows.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>

main(argc, argv)
        int argc;
        char **argv;
{
        struct hostent *hp;
        struct timeval pertry_timeout, total_timeout;
        struct sockaddr_in server_addr;
        int addrlen, sock = RPC_ANYSOCK;
        register CLIENT *client;
        enum clnt_stat clnt_stat;
        char buf[1000];
        char *s = buf;

        /*
         * initial as in example 3.3
         */
        if ((client = clnttcp_create(&server_addr, WINDOWPROG,
            WINDOWVERS, &sock, 0, 0)) == NULL) {
                perror("clnttcp_create");
                exit(-1);
        }
        total_timeout.tv_sec = 0;
        total_timeout.tv_usec = 0;
        while (scanf("%s", s) != EOF) {
                clnt_stat = clnt_call(client, RENDERSTRING_BATCHED,
                    xdr_wrapstring, &s, NULL, NULL, total_timeout);
                if (clnt_stat != RPC_SUCCESS) {
                        clnt_perror(client, "batched rpc");
                        exit(-1);
                }
        }
        /*
         * now flush the pipeline
         */
        total_timeout.tv_sec = 20;
        clnt_stat = clnt_call(client, NULLPROC,
            xdr_void, NULL, xdr_void, NULL, total_timeout);
        if (clnt_stat != RPC_SUCCESS) {
                clnt_perror(client, "rpc");
                exit(-1);
        }

        clnt_destroy(client);
}
```

Since the server sends no message, the clients cannot be notified of any of the failures that may occur. Therefore, clients are on their own when it comes to handling errors.

The above example was completed to render all of the (2000) lines in the file */etc/termcap*. The rendering service did nothing but to throw the lines away. The example was run in the following four configurations: 1) machine to itself, regular RPC; 2) machine to itself, batched RPC; 3) machine to another, regular RPC; and 4) machine to another, batched RPC. The results are as follows: 1) 50 seconds; 2) 16 seconds; 3) 52 seconds; 4) 10 seconds. Running fscanf() on */etc/termcap* only requires six seconds. These timings show the advantage of protocols that allow for overlapped execution, though these protocols are often hard to design.

## 4.4. Authentication

In the examples presented so far, the caller never identified itself to the server, and the server never required an ID from the caller. Clearly, some network services, such as a network filesystem, require stronger security than what has been presented so far.

In reality, every RPC call is authenticated by the RPC package on the server, and similarly, the RPC client package generates and sends authentication parameters. Just as different transports (TCP/IP or UDP/IP) can be used when creating RPC clients and servers, different forms of authentication can be associated with RPC clients; the default authentication type used as a default is type *none*.

The authentication subsystem of the RPC package is open ended. That is, numerous types of authentication are easy to support. However, this section deals only with *unix* type authentication, which besides *none* is the only supported type.

### 4.4.1. The Client Side

When a caller creates a new RPC client handle as in:

```
clnt = clntudp_create(address, prognum, versnum, wait, sockp)
```

the appropriate transport instance defaults the associate authentication handle to be

```
clnt->cl_auth = authnone_create();
```

The RPC client can choose to use *unix* style authentication by setting clnt->cl_auth after creating the RPC client handle:

```
clnt->cl_auth = authunix_create_default();
```

This causes each RPC call associated with clnt to carry with it the following authentication credentials structure:

```
/*
 * Unix style credentials.
 */
struct authunix_parms {
        u_long  aup_time;        /* credentials creation time */
        char    *aup_machname;   /* host name of where the client is calling */
        int     aup_uid;         /* client's UNIX effective uid */
        int     aup_gid;         /* client's current UNIX group id */
        u_int   aup_len;         /* the element length of aup_gids array */
        int     *aup_gids;       /* array of 4.2 groups to which user belongs */
};
```

These fields are set by authunix_create_default() by invoking the appropriate system calls.

Since the RPC user created this new style of authentication, he is responsible for destroying it with:

```
        auth_destroy(clnt->cl_auth);
```

### *4.4.2. The Server Side*

Service implementors have a harder time dealing with authentication issues since the RPC package passes the service dispatch routine a request that has an arbitrary authentication style associated with it. Consider the fields of a request handle passed to a service dispatch routine:

```
        /*
         * An RPC Service request
         */
        struct svc_req {
                u_long          rq_prog;        /* service program number */
                u_long          rq_vers;        /* service protocol version number*/
                u_long          rq_proc;        /* the desired procedure number*/
                struct opaque_auth rq_cred;     /* raw credentials from the ``wire'' */
                caddr_t         rq_clntcred;    /* read only, cooked credentials */
        };
```

The `rq_cred` is mostly opaque, except for one field of interest: the style of authentication credentials:

```
        /*
         * Authentication info.  Mostly opaque to the programmer.
         */
        struct opaque_auth {
                enum_t  oa_flavor;      /* style of credentials */
                caddr_t oa_base;        /* address of more auth stuff */
                u_int   oa_length;      /* not to exceed MAX_AUTH_BYTES */
        };
```

The RPC package guarantees the following to the service dispatch routine:

1) That the request's `rq_cred` is well formed. Thus the service implementor may inspect the request's `rq_cred.oa_flavor` to determine which style of authentication the caller used. The service implementor may also wish to inspect the other fields of `rq_cred` if the style is not one of the styles supported by the RPC package.

2) That the request's `rq_clntcred` field is either NULL or points to a well formed structure that corresponds to a supported style of authentication credentials. Remember that only ***unix*** style is currently supported, so (currently) `rq_clntcred` could be cast to a pointer to an `authunix_parms` structure. If `rq_clntcred` is NULL, the service implementor may wish to inspect the other (opaque) fileds of `rq_cred` in case the service knows about a new type of authentication that the RPC package does not know about.

Our remote users service example can be extended so that it computes results for all users except UID 16:

```
nuser(rqstp, tranp)
        struct svc_req *rqstp;
        SVCXPRT *transp;
{
        struct authunix_parms *unix_cred;
        int uid;
        unsigned long nusers;

        /*
         * we don't care about authentication for the null procedure
         */
        if (rqstp->rq_proc == NULLPROC) {
                if (!svc_sendreply(transp, xdr_void, 0)) {
                        fprintf(stderr, "couldn't reply to RPC call\n");
                        exit(1);
                 }
                 return;
        }
        /*
         * now get the uid
         */
        switch (rqstp->rq_cred.oa_flavor) {
        case AUTH_UNIX:
                unix_cred = (struct authunix_parms *) rqstp->rq_clntcred;
                uid = unix_cred->aup_uid;
                break;
        case AUTH_NULL:
        default:
                svcerr_weakauth(transp);
                return;
        }
        switch (rqstp->rq_proc) {
        case RUSERSPROC_NUM:
                /*
                 * make sure the caller is allow to call this procedure.
                 */
                if (uid == 16) {
                        svcerr_systemerr(transp);
                        return;
                }
                /*
                 * code here to compute the number of users
                 * and put in variable nusers
                 */
                if (!svc_sendreply(transp, xdr_u_long, &nusers) {
                        fprintf(stderr, "couldn't reply to RPC call\n");
                        exit(1);
                }
                return;
        default:
                svcerr_noproc(transp);
                return;
        }
}
```

A few things should be noted here. First, it is customary not to check the authentication parameters associated with the NULLPROC (procedure number zero). Second, if the authentication parameter's type is not suitable for your service, you should call `svcerr_weakauth()`. And finally, the service protocol itself should return status for access denied; in the case of our example, the protocol does not have such a status, so we call the service primitive `svcerr_systemerr()` instead.

The last point underscores the relation between the RPC authentication package and the services; RPC deals only with authentication and not with individual services' access control. The services themselves must implement their own access control policies and reflect these policies as return statuses in their protocols.

## 4.5. Using Inetd

An RPC server can be started from `inetd`. The only difference from the usual code is that `svcudp_create()` should be called as

```
transp = svcudp_create(0);
```

since `inet` passes a socket as file descriptor 0. Also, `svc_register()` should be called as

```
svc_register(PROGNUM, VERSNUM, service, transp, 0);
```

with the final flag as 0, since the program would already be registered by `inetd`. Remember that if you want to exit from the server process and return control to `inet`, you need to explicitly exit, since `svc_run()` never returns.

The format of entries in /etc/servers for RPC services is

```
rpc udp server program version
```

where ***server*** is the C code implementing the server, and ***program*** and ***version*** are the program and version numbers of the service. The key word `udp` can be replaced by `tcp` for TCP-based RPC services.

If the same program handles multiple versions, then the version number can be a range, as in this example:

```
rpc udp /usr/etc/rstatd 100001 1-2
```

# 5. More Examples

## 5.1. Versions

By convention, the first version number of program FOO is FOOVERS_ORIG and the most recent version is FOOVERS. Suppose there is a new version of the `user` program that returns an `unsigned short` rather than a `long`. If we name this version RUSERSVERS_SHORT, then a server that wants to support both versions would do a double register.

```
if (!svc_register(transp, RUSERSPROG, RUSERSVERS_ORIG, nuser,
    IPPROTO_TCP)) {
        fprintf(stderr, "couldn't register RUSER service\n");
        exit(1);
}
if (!svc_register(transp, RUSERSPROG, RUSERSVERS_SHORT, nuser,
    IPPROTO_TCP)) {
        fprintf(stderr, "couldn't register RUSER service\n");
        exit(1);
}
```

Both versions can be handled by the same C procedure:

```
nuser(rqstp, tranp)
        struct svc_req *rqstp;
        SVCXPRT *transp;
{
        unsigned long nusers;
        unsigned short nusers2

        switch (rqstp->rq_proc) {
        case NULLPROC:
                if (!svc_sendreply(transp, xdr_void, 0)) {
                        fprintf(stderr, "couldn't reply to RPC call\n");
                        exit(1);
                }
                return;
        case RUSERSPROC_NUM:
                /*
                 * code here to compute the number of users
                 * and put in variable nusers
                 */
                nusers2 = nusers;
                if (rqstp->rq_vers == RUSERSVERS_ORIG)
                        if (!svc_sendreply(transp, xdr_u_long, &nusers) {
                                fprintf(stderr, "couldn't reply to RPC call\n");
                                exit(1);
                        }
                else
                        if (!svc_sendreply(transp, xdr_u_short, &nusers2) {
                                fprintf(stderr, "couldn't reply to RPC call\n");
                                exit(1);
                return;
        default:
                svcerr_noproc(transp);
                return;
        }
}
```

## 5.2. TCP

Here is an example that is essentially `rcp`. The initiator of the RPC `snd()` call takes its standard input and sends it to the server `rcv()`, which prints it on standard output. The RPC call uses TCP. This also illustrates an XDR procedure that behaves differently on serialization than on deserialization.

```
/*
 * The xdr routine:
 *
 * on decode, read from wire, write onto fp
 * on encode, read from fp, write onto wire
 */
#include <stdio.h>
#include <rpc/rpc.h>

xdr_rcp(xdrs, fp)
        XDR *xdrs;
        FILE *fp;
{
        unsigned long size;
        char buf[MAXCHUNK], *p;

        if (xdrs->x_op == XDR_FREE)/* nothing to free */
                return 1;
        while (1) {
                if (xdrs->x_op == XDR_ENCODE) {
                        if ((size = fread (buf, sizeof(char), MAXCHUNK, fp))
                            == 0 && ferror(fp)) {
                                fprintf(stderr, "couldn't fread\n");
                                exit(1);
                        }
                }
                p = buf;
                if (!xdr_bytes(xdrs, &p, &size, MAXCHUNK))
                        return 0;
                if (size == 0)
                        return 1;
                if (xdrs->x_op == XDR_DECODE) {
                        if (fwrite(buf, sizeof(char), size, fp) != size) {
                                fprintf(stderr, "couldn't fwrite\n");
                                exit(1);
                        }
                }
        }
}
```

```
/*
 * The sender routines
 */
#include <stdio.h>
#include <netdb.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
#include <sys/time.h>

main(argc, argv)
        int argc;
        char **argv;
{
        int err;

        if (argc < 2) {
                fprintf(stderr, "usage: %s server-name\n", argv[0]);
                exit(-1);
        }
        if ((err = callrpctcp(argv[1], RCPPROG, RCPPROC_FP, RCPVERS,
            xdr_rcp, stdin, xdr_void, 0) != 0)) {
                clnt_perrno(err);
                fprintf(stderr, " couldn't make RPC call\n");
                exit(1);
        }
}

callrpctcp(host, prognum, procnum, versnum, inproc, in, outproc, out)
        char *host, *in, *out;
        xdrproc_t inproc, outproc;
{
        struct sockaddr_in server_addr;
        int socket = RPC_ANYSOCK;
        enum clnt_stat clnt_stat;
        struct hostent *hp;
        register CLIENT *client;
        struct timeval total_timeout;

        if ((hp = gethostbyname(host)) == NULL) {
                fprintf(stderr, "cannot get addr for '%s'\n", host);
                exit(-1);
        }
        bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr, hp->h_length);
        server_addr.sin_family = AF_INET;
        server_addr.sin_port =  0;
        if ((client = clnttcp_create(&server_addr, prognum,
            versnum, &socket, BUFSIZ, BUFSIZ)) == NULL) {
                perror("rpctcp_create");
                exit(-1);
        }
        total_timeout.tv_sec = 20;
        total_timeout.tv_usec = 0;
        clnt_stat = clnt_call(client, procnum, inproc, in, outproc, out, total_timeout);
        clnt_destroy(client)
        return (int)clnt_stat;
}
```

```
/*
 * The receiving routines
 */
#include <stdio.h>
#include <rpc/rpc.h>

main()
{
        register SVCXPRT *transp;

        if ((transp = svctcp_create(RPC_ANYSOCK, 1024, 1024)) == NULL) {
                fprintf("svctcp_create: error\n");
                exit(1);
        }
        pmap_unset(RCPPROG, RCPVERS);
        if (!svc_register(transp, RCPPROG, RCPVERS, rcp_service, IPPROTO_TCP)) {
                fprintf(stderr, "svc_register: error\n");
                exit(1);
        }
        svc_run();  /* never returns */
        fprintf(stderr, "svc_run should never return\n");
}

rcp_service(rqstp, transp)
        register struct svc_req *rqstp;
        register SVCXPRT *transp;
{
        switch (rqstp->rq_proc) {
        case NULLPROC:
                if (svc_sendreply(transp, xdr_void, 0) == 0) {
                        fprintf(stderr, "err: rcp_service");
                        exit(1);
                }
                return;
        case RCPPROC_FP:
                if (!svc_getargs(transp, xdr_rcp, stdout)) {
                        svcerr_decode(transp);
                        return;
                }
                if (!svc_sendreply(transp, xdr_void, 0)) {
                        fprintf(stderr, "can't reply\n");
                        return;
                }
                exit(0);
        default:
                svcerr_noproc(transp);
                return;
        }
}
```

## 5.3. Callback Procedures

Occasionally, it is useful to have a server become a client, and make an RPC call back the process which is its client. An example is remote debugging, where the client is a window system program, and the server is a debugger running on the remote machine. Most of the time, the user clicks a mouse button at the debugging window, which converts this to a debugger command, and then makes an RPC call to the server (where the debugger is actually running), telling it to execute that command. However, when the debugger hits a breakpoint, the roles are reversed, and the debugger wants to make an rpc call to the window program, so that it can inform the user that a breakpoint has been reached.

In order to do an RPC callback, you need a program number to make the RPC call on. Since this will be a dynamically generated program number, it should be in the transient range, 0x40000000 - 0x5fffffff. The routine `gettransient()` returns a valid program number in the transient range, and registers it with the portmapper. It only talks to the portmapper running on the same machine as the `gettransient()` routine itself. The call to `pmap_set()` is a test and set operation, in that it indivisibly tests whether a program number has already been registered, and if it has not, then reserves it. On return, the `sockp` argument will contain a socket that can be used as the argument to an `svcudp_create()` or `svctcp_create()` call.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/socket.h>

gettransient(proto, vers, sockp)
        int *sockp;
{
        static int prognum = 0x40000000;
        int s, len, socktype;
        struct sockaddr_in addr;

        switch(proto) {
                case IPPROTO_UDP:
                        socktype = SOCK_DGRAM;
                        break;
                case IPPROTO_TCP:
                        socktype = SOCK_STREAM;
                        break;
                default:
                        fprintf(stderr, "unknown protocol type\n");
                        return 0;
        }
        if (*sockp == RPC_ANYSOCK) {
                if ((s = socket(AF_INET, socktype, 0)) < 0) {
                        perror("socket");
                        return (0);
                }
                *sockp = s;
        }
        else
                s = *sockp;
        addr.sin_addr.s_addr = 0;
        addr.sin_family = AF_INET;
        addr.sin_port = 0;
        len = sizeof(addr);
        /*
         * may be already bound, so don't check for err
         */
        bind(s, &addr, len);
        if (getsockname(s, &addr, &len)< 0) {
                perror("getsockname");
                return (0);
        }
        while (pmap_set(prognum++, vers, proto, addr.sin_port) == 0)
                continue;
        return (prognum-1);
}
```

The following pair of programs illustrate how to use the `gettransient()` routine. The client makes an RPC call to the server, passing it a transient program number. Then the client waits around to receive a callback from the server at that program number. The server registers the program EXAMPELPROG, so that it can receive the RPC call informing it of the callback program number. Then at some random time (on receiving an ALRM signal in this example), it sends a callback RPC call, using the program number it received earlier.

```
/*
 * client
 */
#include <stdio.h>
#include <rpc/rpc.h>

int callback();
char hostname[256];

main(argc, argv)
        char **argv;
{
        int x, ans, s;
        SVCXPRT *xprt;

        gethostname(hostname, sizeof(hostname));
        s = RPC_ANYSOCK;
        x = gettransient(IPPROTO_UDP, 1, &s);
        fprintf(stderr, "client gets prognum %d\n", x);

        if ((xprt = svcudp_create(s)) == NULL) {
            fprintf(stderr, "rpc_server: svcudp_create\n");
            exit(1);
        }
        (void)svc_register(xprt, x, 1, callback, 0);

        ans = callrpc(hostname, EXAMPLEPROG, EXAMPLEPROC_CALLBACK,
            EXAMPLEVERS, xdr_int, &x, xdr_void, 0);
        if (ans != 0) {
                fprintf(stderr, "call: ");
                clnt_perrno(ans);
                fprintf(stderr, "\n");
        }
        svc_run();
        fprintf(stderr, "Error: svc_run shouldn't have returned\n");
}
```

```
callback(rqstp, transp)
        register struct svc_req *rqstp;
        register SVCXPRT *transp;
{
        switch (rqstp->rq_proc) {
                case 0:
                        if (svc_sendreply(transp, xdr_void, 0)  == FALSE) {
                                fprintf(stderr, "err: rusersd\n");
                                exit(1);
                            }
                        exit(0);
                case 1:
                        if (!svc_getargs(transp, xdr_void, 0)) {
                                svcerr_decode(transp);
                                exit(1);
                        }
                        fprintf(stderr, "client got callback\n");
                        if (svc_sendreply(transp, xdr_void, 0)  == FALSE) {
                                fprintf(stderr, "err: rusersd");
                                exit(1);
                        }
        }
}
```

```
/*
 * server
 */
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/signal.h>

char *getnewprog();
char hostname[256];
int docallback();
int pnum;                   /*program number for callback routine */

main(argc, argv)
        char **argv;
{
        gethostname(hostname, sizeof(hostname));
        registerrpc(EXAMPLEPROG, EXAMPLEPROC_CALLBACK, EXAMPLEVERS,
            getnewprog, xdr_int, xdr_void);
        fprintf(stderr, "server going into svc_run\n");
        alarm(10);
        signal(SIGALRM, docallback);
        svc_run();
        fprintf(stderr, "Error: svc_run shouldn't have returned\n");
}

char *
getnewprog(pnump)
        char *pnump;
{
        pnum = *(int *)pnump;
        return NULL;
}

docallback()
{
        int ans;

        ans = callrpc(hostname, pnum, 1, 1, xdr_void, 0, xdr_void, 0);
        if (ans != 0) {
                fprintf(stderr, "server: ");
                clnt_perrno(ans);
                fprintf(stderr, "\n");
        }
}
```

# Appendix A. Synopsis of RPC Routines

### auth_destroy()

```
void
auth_destroy(auth)
        AUTH *auth;
```

A macro that destroys the authentication information associated with auth. Destruction usually involves deallocation of private data structures. The use of auth is undefined after calling auth_destroy().

### authnone_create()

```
AUTH *
authnone_create()
```

Creates and returns an RPC authentication handle that passes no usable authentication information with each remote procedure call.

### authunix_create()

```
AUTH *
authunix_create(host, uid, gid, len, aup_gids)
        char *host;
        int uid, gid, len, *aup_gids;
```

Creates and returns an RPC authentication handle that contains UNIX† authentication information. The parameter host is the name of the machine on which the information was created; uid is the user's user ID; gid is the user's current group ID; len and aup_gids refer to a counted array of groups to which the user belongs. It is easy to impersonate a user.

### authunix_create_default()

```
AUTH *
authunix_create_default()
```

Calls authunix_create() with the appropriate parameters.

### callrpc()

```
callrpc(host, prognum, versnum, procnum, inproc, in, outproc, out)
        char *host;
        u_long prognum, versnum, procnum;
        char *in, *out;
        xdrproc_t inproc, outproc;
```

Calls the remote procedure associated with prognum, versnum, and procnum on the machine, host. The parameter in is the address of the procedure's argument(s), and out is the address of where to place the result(s); inproc is used to encode the procedure's parameters, and outproc is used to decode the procedure's results. This routine returns zero if it succeeds, or

† UNIX is a trademark of Bell Laboratories.

the value of enum clnt_stat cast to an integer if it fails. The routine clnt_perrno() is handy for translating failure statuses into messages. Warning: calling remote procedures with this routine uses UDP/IP as a transport; see clntudp_create() for restrictions.

### clnt_broadcast()

```
enum clnt_stat
clnt_broadcast(prognum, versnum, procnum, inproc, in, outproc, out, eachresult)
        u_long prognum, versnum, procnum;
        char *in, *out;
        xdrproc_t inproc, outproc;
        resultproc_t eachresult;
```

Like callrpc(), except the call message is broadcast to all locally connected broadcast nets. Each time it receives a response, this routine calls eachresult, whose form is

```
        eachresult(out, addr)
                char *out;
                struct sockaddr_in *addr;
```

where out is the same as out passed to clnt_broadcast(), except that the remote procedure's output is decoded there; addr points to the address of the machine that sent the results. If eachresult() returns zero, clnt_broadcast() waits for more replies; otherwise it returns with appropriate status.

### clnt_call()

```
enum clnt_stat
clnt_call(clnt, procnum, inproc, in, outproc, out, tout)
        CLIENT *clnt; long procnum;
        xdrproc_t inproc, outproc;
        char *in, *out;
        struct timeval tout;
```

A macro that calls the remote procedure procnum associated with the client handle, clnt, which is obtained with an RPC client creation routine such as clntudp_create. The parameter in is the address of the procedure's argument(s), and out is the address of where to place the result(s); inproc is used to encode the procedure's parameters, and outproc is used to decode the procedure's results; tout is the time allowed for results to come back.

### clnt_destroy()

```
clnt_destroy(clnt)
        CLIENT *clnt;
```

A macro that destroys the client's RPC handle. Destruction usually involves deallocation of private data structures, including clnt itself. Use of clnt is undefined after calling clnt_destroy(). Warning: client destruction routines do not close sockets associated with clnt; this is the responsibility of the user.

### clnt_freeres()

```
clnt_freeres(clnt, outproc, out)
        CLIENT *clnt;
        xdrproc_t outproc;
        char *out;
```

A macro that frees any data allocated by the RPC/XDR system when it decoded the results of an RPC call. The parameter out is the address of the results, and outproc is the XDR routine describing the results in simple primitives. This routine returns one if the results were successfully freed, and zero otherwise.

### clnt_geterr()

```
void
clnt_geterr(clnt, errp)
        CLIENT *clnt;
        struct rpc_err *errp;
```

A macro that copies the error structure out of the client handle to the structure at address errp.

### clnt_pcreateerror()

```
void
clnt_pcreateerror(s)
        char *s;
```

Prints a message to standard error indicating why a client RPC handle could not be created. The message is prepended with string s and a colon.

### clnt_perrno()

```
void
clnt_perrno(stat)
        enum clnt_stat;
```

Prints a message to standard error corresponding to the condition indicated by stat.

### clnt_perror()

```
clnt_perror(clnt, s)
        CLIENT *clnt;
        char *s;
```

Prints a message to standard error indicating why an RPC call failed; clnt is the handle used to do the call. The message is prepended with string s and a colon.

### clntraw_create()

```
CLIENT *
clntraw_create(prognum, versnum)
        u_long prognum, versnum;
```

This routine creates a toy RPC client for the remote program prognum, version versnum. The transport used to pass messages to the service is actually a buffer within the process's address space, so the corresponding RPC server should live in the same address space; see svcraw_create(). This allows simulation of RPC and acquisition of RPC overheads, such as round trip times, without any kernel interference. This routine returns NULL if it fails.

### clnttcp_create()

```
CLIENT *
clnttcp_create(addr, prognum, versnum, sockp, sendsz, recvsz)
        struct sockaddr_in *addr;
        u_long prognum, versnum;
        int *sockp;
        u_int sendsz, recvsz;
```

This routine creates an RPC client for the remote program prognum, version versnum; the client uses TCP/IP as a transport. The remote program is located at Internet address *addr. If addr->sin_port is zero, then it is set to the actual port that the remote program is listening on (the remote ***portmap*** service is consulted for this information). The parameter *sockp is a socket; if it is RPC_ANYSOCK, then this routine opens a new one and sets *sockp. Since TCP-based RPC uses buffered I/O, the user may specify the size of the send and receive buffers with the parameters sendsz and recvsz; values of zero choose suitable defaults. This routine returns NULL if it fails.

### clntudp_create()

```
CLIENT *
clntudp_create(addr, prognum, versnum, wait, sockp)
        struct sockaddr_in *addr;
        u_long prognum, versnum;
        struct timeval wait;
        int *sockp;
```

This routine creates an RPC client for the remote program prognum, version versnum; the client uses use UDP/IP as a transport. The remote program is located at Internet address *addr. If addr->sin_port is zero, then it is set to actual port that the remote program is listening on (the remote ***portmap*** service is consulted for this information). The parameter *sockp is a socket; if it is RPC_ANYSOCK, then this routine opens a new one and sets *sockp. The UDP transport resends the call message in intervals of wait time until a response is received or until the call times out. Warning: since UDP-based RPC messages can only hold up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

### get_myaddress()

```
void
get_myaddress(addr)
        struct sockaddr_in *addr;
```

Stuffs the machine's IP address into **`*addr`**, without consulting the library routines that deal with */etc/hosts*. The port number is always set to **`htons(PMAPPORT)`**.

### pmap_getmaps()

```
struct pmaplist *
pmap_getmaps(addr)
        struct sockaddr_in *addr;
```

A user interface to the ***portmap*** service, which returns a list of the current RPC program-to-port mappings on the host located at IP address **`*addr`**. This routine can return NULL. The command `rpcinfo -p` uses this routine.

### pmap_getport()

```
u_short
pmap_getport(addr, prognum, versnum, protocol)
        struct sockaddr_in *addr;
        u_long prognum, versnum, protocol;
```

A user interface to the ***portmap*** service, which returns the port number on which waits a service that supports program number **`prognum`**, version **`versnum`**, and speaks the transport protocol associated with protocol. A return value of zero means that the mapping does not exist or that the RPC system failured to contact the remote ***portmap*** service. In the latter case, the global variable `rpc_createerr` contains the RPC status.

### pmap_rmtcall()

```
enum clnt_stat
pmap_rmtcall(addr, prognum, versnum, procnum,
             inproc, in, outproc, out, tout, portp)
        struct sockaddr_in *addr;
        u_long prognum, versnum, procnum;
        char *in, *out;
        xdrproc_t inproc, outproc;
        struct timeval tout;
        u_long *portp;
```

A user interface to the ***portmap*** service, which instructs ***portmap*** on the host at IP address `*addr` to make an RPC call on your behalf to a procedure on that host. The parameter `*portp` will be modified to the program's port number if the procedure succeeds. The definitions of other parameters are discussed in `callrpc()` and `clnt_call()`; see also `clnt_broadcast()`.

### pmap_set()

```
pmap_set(prognum, versnum, protocol, port)
        u_long prognum, versnum, protocol;
        u_short port;
```

A user interface to the ***portmap*** service, which establishes a mapping between the triple `[prognum,versnum,protocol]` and `port` on the machine's ***portmap*** service. The value of protocol is most likely IPPROTO_UDP or IPPROTO_TCP. This routine returns one if it succeeds, zero otherwise.

### pmap_unset()

```
pmap_unset(prognum, versnum)
        u_long prognum, versnum;
```

A user interface to the ***portmap*** service, which destroys all mappings between the triple `[prognum,versnum,*]` and `ports` on the machine's ***portmap*** service. This routine returns one if it succeeds, zero otherwise.

### registerrpc()

```
registerrpc(prognum, versnum, procnum, procname, inproc, outproc)
        u_long prognum, versnum, procnum;
        char *(*procname)();
        xdrproc_t inproc, outproc;
```

Registers procedure `procname` with the RPC service package. If a request arrives for program `prognum`, version `versnum`, and procedure `procnum`, `procname` is called with a pointer to its parameter(s); `progname` should return a pointer to its static result(s); `inproc` is used to decode the parameters while `outproc` is used to encode the results. This routine returns zero if the registration succeeded, −1 otherwise. Warning: remote procedures registered in this form are accessed using the UDP/IP transport; see `svcudp_create()` for restrictions.

### rpc_createerr

```
struct rpc_createerr    rpc_createerr;
```

A global variable whose value is set by any RPC client creation routine that does not succeed. Use the routine `clnt_pcreateerror()` to print the reason why.

### svc_destroy()

```
svc_destroy(xprt)
        SVCXPRT *xprt;
```

A macro that destroys the RPC service transport handle, `xprt`. Destruction usually involves deallocation of private data structures, including `xprt` itself. Use of `xprt` is undefined after calling this routine.

### svc_fds

```
int     svc_fds;
```

A global variable reflecting the RPC service side's read file descriptor bit mask; it is suitable as a parameter to the `select` system call. This is only of interest if a service implementor does not call `svc_run()`, but rather does his own asynchronous event processing. This variable is read-only (do not pass its address to `select`!), yet it may change after calls to `svc_getreq()` or any creation routines.

### svc_freeargs()

```
svc_freeargs(xprt, inproc, in)
        SVCXPRT *xprt;
        xdrproc_t inproc;
        char *in;
```

A macro that frees any data allocated by the RPC/XDR system when it decoded the arguments to a service procedure using `svc_getargs()` . This routine returns one if the results were successfully freed, and zero otherwise.

### svc_getargs()

```
svc_getargs(xprt, inproc, in)
        SVCXPRT *xprt;
        xdrproc_t inproc;
        char *in;
```

A macro that decodes the arguments of an RPC request associated with the RPC service transport handle, `xprt`. The parameter `in` is the address where the arguments will be placed; `inproc` is the XDR routine used to decode the arguments. This routine returns one if decoding succeeds, and zero otherwise.

### svc_getcaller()

```
struct sockaddr_in
svc_getcaller(xprt)
        SVCXPRT *xprt;
```

The approved way of getting the network address of the caller of a procedure associated with the RPC service transport handle, `xprt`.

### svc_getreq()

```
svc_getreq(rdfds)
        int rdfds;
```

This routine is only of interest if a service implementor does not call `svc_run()`, but instead implements custom asynchronous event processing. It is called when the `select` system call has determined that an RPC request has arrived on some RPC socket(s); `rdfds` is the resultant read file descriptor bit mask. The routine returns when all sockets associated with the value of `rdfds` have been serviced.

### svc_register()

```
svc_register(xprt, prognum, versnum, dispatch, protocol)
        SVCXPRT *xprt;
        u_long prognum, versnum;
        void (*dispatch)();
        u_long protocol;
```

Associates prognum and versnum with the service dispatch procedure, dispatch. If protocol is non-zero, then a mapping of the triple [prognum,versnum,protocol] to xprt->xp_port is also established with the local *portmap* service (generally protocol is zero, IPPROTO_UDP or IPPROTO_TCP). The procedure dispatch() has the following form:

```
dispatch(request, xprt)
        struct svc_req *request;
        SVCXPRT *xprt;
```

The svc_register routine returns one if it succeeds, and zero otherwise.

### svc_run()

```
svc_run()
```

This routine never returns. It waits for RPC requests to arrive and calls the appropriate service procedure (using svc_getreq) when one arrives. This procedure is usually waiting for a select system call to return.

### svc_sendreply()

```
svc_sendreply(xprt, outproc, out)
        SVCXPRT *xprt;
        xdrproc_t outproc;
        char *out;
```

Called by an RPC service's dispatch routine to send the results of a remote procedure call. The parameter xprt is the caller's associated transport handle; outproc is the XDR routine which is used to encode the results; and out is the address of the results. This routine returns one if it succeeds, zero otherwise.

### svc_unregister()

```
void
svc_unregister(prognum, versnum)
        u_long prognum, versnum;
```

Removes all mapping of the double [prognum,versnum] to dispatch routines, and of the triple [prognum,versnum,*] to port number.

### svcerr_auth()

```
void
svcerr_auth(xprt, why)
        SVCXPRT *xprt;
        enum auth_stat why;
```

Called by a service dispatch routine that refuses to perform a remote procedure call due to an authentication error.

### svcerr_decode()

```
void
svcerr_decode(xprt)
        SVCXPRT *xprt;
```

Called by a service dispatch routine that can't successfully decode its parameters. See also `svc_getargs()`.

### svcerr_noproc()

```
void
svcerr_noproc(xprt)
        SVCXPRT *xprt;
```

Called by a service dispatch routine that doesn't implement the desired procedure number the caller request.

### svcerr_noprog()

```
void
svcerr_noprog(xprt)
        SVCXPRT *xprt;
```

Called when the desired program is not registered with the RPC package. Service implementors usually don't need this routine.

### svcerr_progvers()

```
void
svcerr_progvers(xprt)
        SVCXPRT *xprt;
```

Called when the desired version of a program is not registered with the RPC package. Service implementors usually don't need this routine.

### svcerr_systemerr()

```
void
svcerr_systemerr(xprt)
        SVCXPRT *xprt;
```

Called by a service dispatch routine when it detects a system error not covered by any particular protocol. For example, if a service can no longer allocate storage, it may call this routine.

**svcerr_weakauth()**

```
        void
        svcerr_weakauth(xprt)
                SVCXPRT *xprt;
```

Called by a service dispatch routine that refuses to perform a remote procedure call due to insufficient (but correct) authentication parameters. The routine calls `svcerr_auth(xprt,AUTH_TOOWEAK)`.

**svcraw_create()**

```
        SVCXPRT *
        svcraw_create()
```

This routine creates a toy RPC service transport, to which it returns a pointer. The transport is really a buffer within the process's address space, so the corresponding RPC client should live in the same address space; see `clntraw_create()`. This routine allows simulation of RPC and acquisition of RPC overheads (such as round trip times), without any kernel interference. This routine returns NULL if it fails.

**svctcp_create()**

```
        SVCXPRT *
        svctcp_create(sock, send_buf_size, recv_buf_size)
                int sock;
                u_int send_buf_size, recv_buf_size;
```

This routine creates a TCP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket `sock`, which may be RPC_ANYSOCK, in which case a new socket is created. If the socket is not bound to a local TCP port, then this routine binds it to an arbitrary port. Upon completion, `xprt->xp_sock` is the transport's socket number, and `xprt->xp_port` is the transport's port number. This routine returns NULL if it fails. Since TCP-based RPC uses buffered I/O, users may specify the size of the `send` and `receive` buffers; values of zero choose suitable defaults.

**svcudp_create()**

```
        SVCXPRT *
        svcudp_create(sock)
                int sock;
```

This routine creates a UDP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket `sock`, which may be RPC_ANYSOCK, in which case a new socket is created. If the socket is not bound to a local UDP port, then this routine binds it to an arbitrary port. Upon completion, `xprt->xp_sock` is the transport's socket number, and `xprt->xp_port` is the transport's port number. This routine returns NULL if it fails. Warning: since UDP-based RPC messages can only hold up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

### xdr_accepted_reply()

```
xdr_accepted_reply(xdrs, ar)
        XDR *xdrs;
        struct accepted_reply *ar;
```

Used for describing RPC messages, externally. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

### xdr_array()

```
xdr_array(xdrs, arrp, sizep, maxsize, elsize, elproc)
        XDR *xdrs;
        char **arrp;
        u_int *sizep, maxsize, elsize;
        xdrproc_t elproc;
```

A filter primitive that translates between arrays and their corresponding external representations. The parameter **arrp** is the address of the pointer to the array, while **sizep** is the address of the element count of the array; this element count cannot exceed **maxsize**. The parameter **elsize** is the **sizeof()** each of the array's elements, and **elproc** is an XDR filter that translates between the array elements' C form, and their external representation. This routine returns one if it succeeds, zero otherwise.

### xdr_authunix_parms()

```
xdr_authunix_parms(xdrs, aupp)
        XDR *xdrs;
        struct authunix_parms *aupp;
```

Used for describing UNIX credentials, externally. This routine is useful for users who wish to generate these credentials without using the RPC authentication package.

### xdr_bool()

```
xdr_bool(xdrs, bp)
        XDR *xdrs;
        bool_t *bp;
```

A filter primitive that translates between booleans (C integers) and their external representations. When encoding data, this filter produces values of either one or zero. This routine returns one if it succeeds, zero otherwise.

### xdr_bytes()

```
xdr_bytes(xdrs, sp, sizep, maxsize)
        XDR *xdrs;
        char **sp;
        u_int *sizep, maxsize;
```

A filter primitive that translates between counted byte strings and their external representations. The parameter **sp** is the address of the string pointer. The length of the string is located at address **sizep**; strings cannot be longer than **maxsize**. This routine returns one if it succeeds, zero otherwise.

### xdr_callhdr()

```
void
xdr_callhdr(xdrs, chdr)
        XDR *xdrs;
        struct rpc_msg *chdr;
```

Used for describing RPC messages, externally. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

### xdr_callmsg()

```
xdr_callmsg(xdrs, cmsg)
        XDR *xdrs;
        struct rpc_msg *cmsg;
```

Used for describing RPC messages, externally. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

### xdr_double()

```
xdr_double(xdrs, dp)
        XDR *xdrs;
        double *dp;
```

A filter primitive that translates between C `double` precision numbers and their external representations. This routine returns one if it succeeds, zero otherwise.

### xdr_enum()

```
xdr_enum(xdrs, ep)
        XDR *xdrs;
        enum_t *ep;
```

A filter primitive that translates between C `enums` (actually integers) and their external representations. This routine returns one if it succeeds, zero otherwise.

### xdr_float()

```
xdr_float(xdrs, fp)
        XDR *xdrs;
        float *fp;
```

A filter primitive that translates between C `floats` and their external representations. This routine returns one if it succeeds, zero otherwise.

### xdr_inline()

```
long *
xdr_inline(xdrs, len)
        XDR *xdrs;
        int len;
```

A macro that invokes the in-line routine associated with the XDR stream, `xdrs`. The routine returns a pointer to a contiguous piece of the stream's buffer; `len` is the byte length of the desired buffer. Note that pointer is cast to `long *`. Warning: `xdr_inline()` may return 0

(NULL) if it cannot allocate a contiguous piece of a buffer. Therefore the behavior may vary among stream instances; it exists for the sake of efficiency.

### xdr_int()

```
xdr_int(xdrs, ip)
        XDR *xdrs;
        int *ip;
```

A filter primitive that translates between C integers and their external representations. This routine returns one if it succeeds, zero otherwise.

### xdr_long()

```
xdr_long(xdrs, lp)
        XDR *xdrs;
        long *lp;
```

A filter primitive that translates between C `long` integers and their external representations. This routine returns one if it succeeds, zero otherwise.

### xdr_opaque()

```
xdr_opaque(xdrs, cp, cnt)
        XDR *xdrs;
        char *cp;
        u_int cnt;
```

A filter primitive that translates between fixed size opaque data and its external representation. The parameter `cp` is the address of the opaque object, and `cnt` is its size in bytes. This routine returns one if it succeeds, zero otherwise.

### xdr_opaque_auth()

```
xdr_opaque_auth(xdrs, ap)
        XDR *xdrs;
        struct opaque_auth *ap;
```

Used for describing RPC messages, externally. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

### xdr_pmap()

```
xdr_pmap(xdrs, regs)
        XDR *xdrs;
        struct pmap *regs;
```

Used for describing parameters to various ***portmap*** procedures, externally. This routine is useful for users who wish to generate these parameters without using the **pmap** interface.

### xdr_pmaplist()

```
xdr_pmaplist(xdrs, rp)
        XDR *xdrs;
        struct pmaplist **rp;
```

Used for describing a list of port mappings, externally. This routine is useful for users who wish to generate these parameters without using the **pmap** interface.

### xdr_reference()

```
xdr_reference(xdrs, pp, size, proc)
        XDR *xdrs;
        char **pp;
        u_int size;
        xdrproc_t proc;
```

A primitive that provides pointer chasing within structures. The parameter pp is the address of the pointer; `size` is the `sizeof()` the structure that *pp points to; and `proc` is an XDR procedure that filters the structure between its C form and its external representation. This routine returns one if it succeeds, zero otherwise.

### xdr_rejected_reply()

```
xdr_rejected_reply(xdrs, rr)
        XDR *xdrs;
        struct rejected_reply *rr;
```

Used for describing RPC messages, externally. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

### xdr_replymsg()

```
xdr_replymsg(xdrs, rmsg)
        XDR *xdrs;
        struct rpc_msg *rmsg;
```

Used for describing RPC messages, externally. This routine is useful for users who wish to generate RPC style messages without using the RPC package.

### xdr_short()

```
xdr_short(xdrs, sp)
        XDR *xdrs;
        short *sp;
```

A filter primitive that translates between C `short` integers and their external representations. This routine returns one if it succeeds, zero otherwise.

### xdr_string()

```
xdr_string(xdrs, sp, maxsize)
        XDR *xdrs;
        char **sp;
        u_int maxsize;
```

A filter primitive that translates between C strings and their corresponding external representations. Strings cannot cannot be longer than **maxsize**. Note that **sp** is the address of the string's pointer. This routine returns one if it succeeds, zero otherwise.

### xdr_u_int()

```
xdr_u_int(xdrs, up)
        XDR *xdrs;
        unsigned *up;
```

A filter primitive that translates between C **unsigned** integers and their external representations. This routine returns one if it succeeds, zero otherwise.

### xdr_u_long()

```
xdr_u_long(xdrs, ulp)
        XDR *xdrs;
        unsigned long *ulp;
```

A filter primitive that translates between C **unsigned long** integers and their external representations. This routine returns one if it succeeds, zero otherwise.

### xdr_u_short()

```
xdr_u_short(xdrs, usp)
        XDR *xdrs;
        unsigned short *usp;
```

A filter primitive that translates between C **unsigned short** integers and their external representations. This routine returns one if it succeeds, zero otherwise.

### xdr_union()

```
xdr_union(xdrs, dscmp, unp, choices, dfault)
        XDR *xdrs;
        int *dscmp;
        char *unp;
        struct xdr_discrim *choices;
        xdrproc_t dfault;
```

A filter primitive that translates between a discriminated C **union** and its corresponding external representation. The parameter **dscmp** is the address of the union's discriminant, while **unp** in the address of the union. This routine returns one if it succeeds, zero otherwise.

### xdr_void()

```
        xdr_void()
```

This routine always returns one.

### xdr_wrapstring()

```
        xdr_wrapstring(xdrs, sp)
                XDR *xdrs;
                char **sp;
```

A primitive that calls `xdr_string(xdrs,sp,MAXUNSIGNED);` where MAXUNSIGNED is the maximum value of an unsigned integer. This is handy because the RPC package passes only two parameters XDR routines, whereas `xdr_string()`, one of the most frequently used primitives, requires three parameters. This routine returns one if it succeeds, zero otherwise.

### xprt_register()

```
        void
        xprt_register(xprt)
                SVCXPRT *xprt;
```

After RPC service transport handles are created, they should register themselves with the RPC service package. This routine modifies the global variable **svc_fds**. Service implementors usually don't need this routine.

### xprt_unregister()

```
        void
        xprt_unregister(xprt)
                SVCXPRT *xprt;
```

Before an RPC service transport handle is destroyed, it should unregister itself with the RPC service package. This routine modifies the global variable **svc_fds**. Service implementors usually don't need this routine.

# External Data Representation Protocol Specification

# Contents

# External Data Representation Protocol Specification

## 1. Introduction

This manual describes library routines that allow a C programmer to describe arbitrary data structures in a machine-independent fashion. The eXternal Data Representation (XDR) standard is the backbone of Sun's Remote Procedure Call package, in the sense that data for remote procedure calls is transmitted using the standard. XDR library routines should be used to transmit data that is accessed (read or written) by more than one type of machine.

This manual contains a description of XDR library routines, a guide to accessing currently available XDR streams, information on defining new streams and data types, and a formal definition of the XDR standard. XDR was designed to work across different languages, operating systems, and machine architectures. Most users (particularly RPC users) only need the information in sections 2 and 3 of this document. Programmers wishing to implement RPC and XDR on new machines will need the information in sections 4 through 6. Advanced topics, not necessary for all implementations, are covered in section 7.

On Sun systems, C programs that want to use XDR routines must include the file `<rpc/rpc.h>`, which contains all the necessary interfaces to the XDR system. Since the C library `libc.a` contains all the XDR routines, compile as normal.

```
cc program.c
```

## 2. Justification

Consider the following two programs, `writer`:

```
#include <stdio.h>

main()                     /* writer.c */
{
        long i;

        for (i = 0; i < 8; i++) {
                if (fwrite((char *)&i, sizeof(i), 1, stdout) != 1) {
                        fprintf(stderr, "failed!\n");
                        exit(1);
                }
        }
}
```

and `reader`:

```
#include <stdio.h>

main()                  /* reader.c */
{
        long i, j;

        for (j = 0; j < 8; j++) {
                if (fread((char *)&i, sizeof (i), 1, stdin) != 1) {
                        fprintf(stderr, "failed!\n");
                        exit(1);
                }
                printf("%ld ", i);
        }
        printf("\n");
}
```

The two programs appear to be portable, because (a) they pass `lint` checking, and (b) they exhibit the same behavior when executed on two different hardware architectures, a Sun and a VAX.

Piping the output of the `writer` program to the `reader` program gives identical results on a Sun or a VAX.[‡]

```
sun% writer | reader
0 1 2 3 4 5 6 7
sun%
---
vax% writer | reader
0 1 2 3 4 5 6 7
vax%
```

With the advent of local area networks and Berkeley's 4.2 BSD UNIX[†] came the concept of "network pipes" — a process produces data on one machine, and a second process consumes data on another machine. A network pipe can be constructed with `writer` and `reader`. Here are the results if the first produces data on a Sun, and the second consumes data on a VAX.

```
sun% writer | rsh vax reader
0 16777216 33554432 50331648 67108864 83886080 100663296 117440512
sun%
```

Identical results can be obtained by executing `writer` on the VAX and `reader` on the Sun. These results occur because the byte ordering of long integers differs between the VAX and the Sun, even though word size is the same. Note that 16777216 is $2^{24}$ — when four bytes are reversed, the 1 winds up in the 24th bit.

Whenever data is shared by two or more machine types, there is a need for portable data. Programs can be made data-portable by replacing the `read()` and `write()` calls with calls to an XDR library routine `xdr_long()`, a filter that knows the standard representation of a long integer in its external form. Here are the revised versions of `writer`:

[‡] VAX is a trademark of Digital Equipment Corporation.

[†] UNIX is a trademark of Bell Laboratories.

```
#include <stdio.h>
#include <rpc/rpc.h>    /* xdr is a sub-library of the rpc library */

main()           /* writer.c */
{
        XDR xdrs;
        long i;

        xdrstdio_create(&xdrs, stdout, XDR_ENCODE);
        for (i = 0; i < 8; i++) {
                if (! xdr_long(&xdrs, &i)) {
                        fprintf(stderr, "failed!\n");
                        exit(1);
                }
        }
}
```

and reader:

```
#include <stdio.h>
#include <rpc/rpc.h>    /* xdr is a sub-library of the rpc library */

main()           /* reader.c */
{
        XDR xdrs;
        long i, j;

        xdrstdio_create(&xdrs, stdin, XDR_DECODE);
        for (j = 0; j < 8; j++) {
                if (! xdr_long(&xdrs, &i)) {
                        fprintf(stderr, "failed!\n");
                        exit(1);
                }
                printf("%ld ", i);
        }
        printf("\n");
}
```

The new programs were executed on a Sun, on a VAX, and from a Sun to a VAX; the results are shown below.

```
sun% writer | reader
0 1 2 3 4 5 6 7
sun%
---
vax% writer | reader
0 1 2 3 4 5 6 7
vax%
---
sun% writer | rsh vax reader
0 1 2 3 4 5 6 7
sun%
```

Dealing with integers is just the tip of the portable-data iceberg. Arbitrary data structures present portability problems, particularly with respect to alignment and pointers. Alignment on word boundaries may cause the size of a structure to vary from machine to machine. Pointers are convenient to use, but have no meaning outside the machine where they are defined.

The XDR library package solves data portability problems. It allows you to write and read arbitrary C constructs in a consistent, specified, well-documented manner. Thus, it makes sense to use the library even when the data is not shared among machines on a network.

The XDR library has filter routines for strings (null-terminated arrays of bytes), structures, unions, and arrays, to name a few. Using more primitive routines, you can write your own specific XDR routines to describe arbitrary data structures, including elements of arrays, arms of unions, or objects pointed at from other structures. The structures themselves may contain arrays of arbitrary elements, or pointers to other structures.

Let's examine the two programs more closely. There is a family of XDR stream creation routines in which each member treats the stream of bits differently. In our example, data is manipulated using standard I/O routines, so we use `xdrstdio_create()`. The parameters to XDR stream creation routines vary according to their function. In our example, `xdrstdio_create()` takes a pointer to an XDR structure that it initializes, a pointer to a FILE that the input or output is performed on, and the operation. The operation may be XDR_ENCODE for serializing in the `writer` program, or XDR_DECODE for deserializing in the `reader` program.

Note: RPC clients never need to create XDR streams; the RPC system itself creates these streams, which are then passed to the clients.

The `xdr_long()` primitive is characteristic of most XDR library primitives and all client XDR routines. First, the routine returns FALSE (0) if it fails, and TRUE (1) if it succeeds. Second, for each data type, `xxx`, there is an associated XDR routine of the form:

```
xdr_xxx(xdrs, fp)
        XDR *xdrs;
        xxx *fp;
{
}
```

In our case, `xxx` is long, and the corresponding XDR routine is a primitive, `xdr_long`. The client could also define an arbitrary structure `xxx` in which case the client would also supply the routine `xdr_xxx`, describing each field by calling XDR routines of the appropriate type. In all cases the first parameter, `xdrs` can be treated as an opaque handle, and passed to the primitive routines.

XDR routines are direction independent; that is, the same routines are called to serialize or deserialize data. This feature is critical to software engineering of portable data. The idea is to call the same routine for either operation — this almost guarantees that serialized data can also be deserialized. One routine is used by both producer and consumer of networked data. This is implemented by always passing the address of an object rather than the object itself — only in the case of deserialization is the object modified. This feature is not shown in our trivial example, but its value becomes obvious when nontrivial data structures are passed among machines. If needed, you can obtain the direction of the XDR operation. See section 3.7 for details.

Let's look at a slightly more complicated example. Assume that a person's gross assets and liabilities are to be exchanged among processes. Also assume that these values are important enough to warrant their own data type:

```
struct gnumbers {
        long g_assets;
        long g_liabilities;
};
```

The corresponding XDR routine describing this structure would be:

```
bool_t                  /* TRUE is success, FALSE is failure */
xdr_gnumbers(xdrs, gp)
        XDR *xdrs;
        struct gnumbers *gp;
{
        if (xdr_long(xdrs, &gp->g_assets) &&
            xdr_long(xdrs, &gp->g_liabilities))
                return(TRUE);
        return(FALSE);
}
```

Note that the parameter **xdrs** is never inspected or modified; it is only passed on to the subcomponent routines. It is imperative to inspect the return value of each XDR routine call, and to give up immediately and return FALSE if the subroutine fails.

This example also shows that the type **bool_t** is declared as an integer whose only values are TRUE (1) and FALSE (0). This document uses the following definitions:

```
#define bool_t  int
#define TRUE    1
#define FALSE   0

#define enum_t int      /* enum_t's are used for generic enum's */
```

Keeping these conventions in mind, **xdr_gnumbers()** can be rewritten as follows:

```
xdr_gnumbers(xdrs, gp)
        XDR *xdrs;
        struct gnumbers *gp;
{
        return (xdr_long(xdrs, &gp->g_assets) &&
                xdr_long(xdrs, &gp->g_liabilities));
}
```

This document uses both coding styles.

# 3. XDR Library Primitives

This section gives a synopsis of each XDR primitive. It starts with basic data types and moves on to constructed data types. Finally, XDR utilities are discussed. The interface to these primitives and utilities is defined in the include file `<rpc/xdr.h>`, automatically included by `<rpc/rpc.h>`.

## 3.1. Number Filters

The XDR library provides primitives that translate between C numbers and their corresponding external representations. The primitives cover the set of numbers in:

*[signed,unsigned]*[short,int,long]*

Specifically, the six primitives are:

```
bool_t xdr_int(xdrs, ip)
        XDR *xdrs;
        int *ip;

bool_t xdr_u_int(xdrs, up)
        XDR *xdrs;
        unsigned *up;

bool_t xdr_long(xdrs, lip)
        XDR *xdrs;
        long *lip;

bool_t xdr_u_long(xdrs, lup)
        XDR *xdrs;
        u_long *lup;

bool_t xdr_short(xdrs, sip)
        XDR *xdrs;
        short *sip;

bool_t xdr_u_short(xdrs, sup)
        XDR *xdrs;
        u_short *sup;
```

The first parameter, `xdrs`, is an XDR stream handle. The second parameter is the address of the number that provides data to the stream or receives data from it. All routines return TRUE if they complete successfully, and FALSE otherwise.

## 3.2. Floating Point Filters

The XDR library also provides primitive routines for C's floating point types:

```
bool_t xdr_float(xdrs, fp)
        XDR *xdrs;
        float *fp;
```

```
bool_t xdr_double(xdrs, dp)
        XDR *xdrs;
        double *dp;
```

The first parameter, **xdrs** is an XDR stream handle. The second parameter is the address of the floating point number that provides data to the stream or receives data from it. All routines return TRUE if they complete successfully, and FALSE otherwise.

Note: Since the numbers are represented in IEEE floating point, routines may fail when decoding a valid IEEE representation into a machine-specific representation, or vice-versa.

## 3.3. Enumeration Filters

The XDR library provides a primitive for generic enumerations. The primitive assumes that a C **enum** has the same representation inside the machine as a C integer. The boolean type is an important instance of the **enum**. The external representation of a boolean is always one (TRUE) or zero (FALSE).

```
#define bool_t  int
#define FALSE   0
#define TRUE    1

#define enum_t int

bool_t xdr_enum(xdrs, ep)
        XDR *xdrs;
        enum_t *ep;

bool_t xdr_bool(xdrs, bp)
        XDR *xdrs;
        bool_t *bp;
```

The second parameters **ep** and **bp** are addresses of the associated type that provides data to, or receives data from, the stream **xdrs**. The routines return TRUE if they complete successfully, and FALSE otherwise.

## 3.4. No Data

Occasionally, an XDR routine must be supplied to the RPC system, even when no data is passed or required. The library provides such a routine:

```
bool_t xdr_void();  /* always returns TRUE */
```

## 3.5. Constructed Data Type Filters

Constructed or compound data type primitives require more parameters and perform more complicated functions then the primitives discussed above. This section includes primitives for strings, arrays, unions, and pointers to structures. Constructed data type primitives may use memory management. In many cases, memory is allocated when deserializing data with XDR_DECODE. Therefore, the XDR package must provide means to deallocate memory. This is done by an XDR operation, XDR_FREE. To review, the three XDR directional operations are XDR_ENCODE, XDR_DECODE, and XDR_FREE.

### *3.5.1. Strings*

In C, a string is defined as a sequence of bytes terminated by a null byte, which is not considered when calculating string length. However, when a string is passed or manipulated, a pointer to it is employed. Therefore, the XDR library defines a string to be a `char *`, and not a sequence of characters. The external representation of a string is drastically different from its internal representation. Externally, strings are represented as sequences of ASCII characters, while internally, they are represented with character pointers. Conversion between the two representations is accomplished with the routine `xdr_string()`:

```
bool_t xdr_string(xdrs, sp, maxlength)
        XDR *xdrs;
        char **sp;
        u_int maxlength;
```

The first parameter `xdrs` is the XDR stream handle. The second parameter `sp` is a pointer to a string (type `char **`). The third parameter `maxlength` specifies the maximum number of bytes allowed during encoding or decoding; its value is usually specified by a protocol. For example, a protocol specification may say that a file name may be no longer than 255 characters. The routine returns FALSE if the number of characters exceeds `maxlength`, and TRUE if it doesn't.

The behavior of `xdr_string()` is similar to the behavior of other routines discussed in this section. The direction XDR_ENCODE is easiest to understand. The parameter `sp` points to a string of a certain length; if it does not exceed `maxlength`, the bytes are serialized.

The effect of deserializing a string is subtle. First the length of the incoming string is determined; it must not exceed `maxlength`. Next `sp` is dereferenced; if the the value is NULL, then a string of the appropriate length is allocated and `*sp` is set to this string. If the original value of `*sp` is non-NULL, then the XDR package assumes that a target area has been allocated, which can hold strings no longer than `maxlength`. In either case, the string is decoded into the target area. The routine then appends a null character to the string.

In the XDR_FREE operation, the string is obtained by dereferencing `sp`. If the string is not NULL, it is freed and `*sp` is set to NULL. In this operation, `xdr_string` ignores the `maxlength` parameter.

### *3.5.2. Byte Arrays*

Often variable-length arrays of bytes are preferable to strings. Byte arrays differ from strings in the following three ways: 1) the length of the array (the byte count) is explicitly located in an unsigned integer, 2) the byte sequence is not terminated by a null character, and 3) the external representation of the bytes is the same as their internal representation. The primitive `xdr_bytes()` converts between the internal and external representations of byte arrays:

```
bool_t xdr_bytes(xdrs, bpp, lp, maxlength)
        XDR *xdrs;
        char **bpp;
        u_int *lp;
        u_int maxlength;
```

The usage of the first, second and fourth parameters are identical to the first, second and third parameters of `xdr_string()`, respectively. The length of the byte area is obtained by dereferencing `lp` when serializing; `*lp` is set to the byte length when deserializing.

### *3.5.3. Arrays*

The XDR library package provides a primitive for handling arrays of arbitrary elements. The `xdr_bytes()` routine treats a subset of generic arrays, in which the size of array elements is known to be 1, and the external description of each element is built-in. The generic array primitive, `xdr_array()` requires parameters identical to those of `xdr_bytes()` plus two more: the size of array elements, and an XDR routine to handle each of the elements. This routine is called to encode or decode each element of the array.

```
bool_t xdr_array(xdrs, ap, lp, maxlength, elementsize, xdr_element)
        XDR *xdrs;
        char **ap;
        u_int *lp;
        u_int maxlength;
        u_int elementsize;
        bool_t (*xdr_element)();
```

The parameter `ap` is the address of the pointer to the array. If `*ap` is NULL when the array is being deserialized, XDR allocates an array of the appropriate size and sets `*ap` to that array. The element count of the array is obtained from `*lp` when the array is serialized; `*lp` is set to the array length when the array is deserialized. The parameter `maxlength` is the maximum number of elements that the array is allowed to have; `elementsize` is the byte size of each element of the array (the C function `sizeof()` can be used to obtain this value). The routine `xdr_element` is called to serialize, deserialize, or free each element of the array.

*Examples*

Before defining more constructed data types, it is appropriate to present three examples.

*Example A*

A user on a networked machine can be identified by (a) the machine name, such as `krypton`: see *gethostname*(3); (b) the user's UID: see *geteuid*(2); and (c) the group numbers to which the user belongs: see *getgroups*(2). A structure with this information and its associated XDR routine could be coded like this:

```
struct netuser {
        char    *nu_machinename;
        int     nu_uid;
        u_int   nu_glen;
        int     *nu_gids;
};
#define NLEN 255 /* machine names must be shorter than 256 chars */
#define NGRPS 20 /* user can't be a member of more than 20 groups */

bool_t
xdr_netuser(xdrs, nup)
        XDR *xdrs;
        struct netuser *nup;
{
        return (xdr_string(xdrs, &nup->nu_machinename, NLEN) &&
            xdr_int(xdrs, &nup->nu_uid) &&
            xdr_array(xdrs, &nup->nu_gids, &nup->nu_glen, NGRPS,
                sizeof (int), xdr_int));
}
```

*Example B*

A party of network users could be implemented as an array of netuser structure. The declaration and its associated XDR routines are as follows:

```
struct party {
        u_int p_len;
        struct netuser *p_nusers;
};
#define PLEN 500 /* max number of users in a party */

bool_t
xdr_party(xdrs, pp)
        XDR *xdrs;
        struct party *pp;
{
        return (xdr_array(xdrs, &pp->p_nusers, &pp->p_len, PLEN,
            sizeof (struct netuser), xdr_netuser));
}
```

*Example C*

The well-known parameters to main(), argc and argv can be combined into a structure. An array of these structures can make up a history of commands. The declarations and XDR routines might look like:

```
struct cmd {
        u_int c_argc;
        char **c_argv;
};
#define ALEN 1000  /* args can be no longer than 1000 chars */
#define NARGC 100  /* commands may have no more than 100 args */

struct history {
        u_int h_len;
        struct cmd *h_cmds;
};
#define NCMDS 75  /* history is no more than 75 commands */

bool_t
xdr_wrap_string(xdrs, sp)
        XDR *xdrs;
        char **sp;
{
        return (xdr_string(xdrs, sp, ALEN));
}

bool_t
xdr_cmd(xdrs, cp)
        XDR *xdrs;
        struct cmd *cp;
{
        return (xdr_array(xdrs, &cp->c_argv, &cp->c_argc, NARGC,
            sizeof (char *), xdr_wrap_string));
}
```

```
bool_t
xdr_history(xdrs, hp)
        XDR *xdrs;
        struct history *hp;
{
        return (xdr_array(xdrs, &hp->h_cmds, &hp->h_len, NCMDS,
            sizeof (struct cmd), xdr_cmd));
}
```

The most confusing part of this example is that the routine `xdr_wrap_string()` is needed to package the `xdr_string()` routine, because the implementation of `xdr_array()` only passes two parameters to the array element description routine; `xdr_wrap_string()` supplies the third parameter to `xdr_string()`.

By now the recursive nature of the XDR library should be obvious. Let's continue with more constructed data types.

### *3.5.4. Opaque Data*

In some protocols, handles are passed from a server to client. The client passes the handle back to the server at some later time. Handles are never inspected by clients; they are obtained and submitted. That is to say, handles are opaque. The primitive `xdr_opaque()` is used for describing fixed sized, opaque bytes.

```
bool_t xdr_opaque(xdrs, p, len)
        XDR *xdrs;
        char *p;
        u_int len;
```

The parameter `p` is the location of the bytes; `len` is the number of bytes in the opaque object. By definition, the actual data contained in the opaque object are not machine portable.

### *3.5.5. Fixed Sized Arrays*

The XDR library does not provide a primitive for fixed-length arrays (the primitive `xdr_array()` is for varying-length arrays). Example A could be rewritten to use fixed-sized arrays in the following fashion:

```
#define NLEN 255  /* machine names must be shorter than 256 chars */
#define NGRPS 20  /* user cannot be a member of more than 20 groups */

struct netuser {
        char *nu_machinename;
        int nu_uid;
        int nu_gids[NGRPS];
};
```

```
bool_t
xdr_netuser(xdrs, nup)
        XDR *xdrs;
        struct netuser *nup;
{
        int i;
        if (! xdr_string(xdrs, &nup->nu_machinename, NLEN))
                return (FALSE);
        if (! xdr_int(xdrs, &nup->nu_uid))
                return (FALSE);
        for (i = 0; i < NGRPS; i++) {
                if (! xdr_int(xdrs, &nup->nu_gids[i]))
                        return (FALSE);
        }
        return (TRUE);
}
```

Exercise: Rewrite example A so that it uses varying-length arrays and so that the netuser structure contains the actual nu_gids array body as in the example above.

### *3.5.6. Discriminated Unions*

The XDR library supports discriminated unions. A discriminated union is a C union and an enum_t value that selects an "arm" of the union.

```
struct xdr_discrim {
        enum_t value;
        bool_t (*proc)();
};

bool_t xdr_union(xdrs, dscmp, unp, arms, defaultarm)
        XDR *xdrs;
        enum_t *dscmp;
        char *unp;
        struct xdr_discrim *arms;
        bool_t (*defaultarm)();  /* may equal NULL */
```

First the routine translates the discriminant of the union located at *dscmp. The discriminant is always an enum_t. Next the union located at *unp is translated. The parameter arms is a pointer to an array of xdr_discrim structures. Each structure contains an order pair of [value,proc]. If the union's discriminant is equal to the associated value, then the proc is called to translate the union. The end of the xdr_discrim structure array is denoted by a routine of value NULL (0). If the discriminant is not found in the arms array, then the defaultarm procedure is called if it is non-NULL; otherwise the routine returns FALSE.

*Example D*

Suppose the type of a union may be integer, character pointer (a string), or a gnumbers structure. Also, assume the union and its current type are declared in a structure. The declaration is:

```
enum utype { INTEGER=1, STRING=2, GNUMBERS=3 };

struct u_tag {
        enum utype utype;       /* this is the union's discriminant */
        union {
                int ival;
                char *pval;
                struct gnumbers gn;
        } uval;
};
```

The following constructs and XDR procedure (de)serialize the discriminated union:

```
struct xdr_discrim u_tag_arms[4] = {
        { INTEGER, xdr_int },
        { GNUMBERS, xdr_gnumbers }
        { STRING, xdr_wrap_string },
        { __dontcare__, NULL }
        /* always terminate arms with a NULL xdr_proc */
}

bool_t
xdr_u_tag(xdrs, utp)
        XDR *xdrs;
        struct u_tag *utp;
{
        return (xdr_union(xdrs, &utp->utype, &utp->uval, u_tag_arms,
            NULL));
}
```

The routine `xdr_gnumbers()` was presented in Section 2; `xdr_wrap_string()` was presented in example C. The default arm parameter to `xdr_union()` (the last parameter) is NULL in this example. Therefore the value of the union's discriminant legally may take on only values listed in the `u_tag_arms` array. This example also demonstrates that the elements of the arm's array do not need to be sorted.

It is worth pointing out that the values of the discriminant may be sparse, though in this example they are not. It is always good practice to assign explicitly integer values to each element of the discriminant's type. This practice both documents the external representation of the discriminant and guarantees that different C compilers emit identical discriminant values.

Exercise: Implement `xdr_union()` using the other primitives in this section.

### *3.5.7. Pointers*

In C it is often convenient to put pointers to another structure within a structure. The primitive `xdr_reference()` makes it easy to serialize, deserialize, and free these referenced structures.

```
bool_t xdr_reference(xdrs, pp, size, proc)
        XDR *xdrs;
        char **pp;
        u_int ssize;
        bool_t (*proc)();
```

Parameter pp is the address of the pointer to the structure; parameter ssize is the size in bytes of the structure (use the C function sizeof() to obtain this value); and proc is the XDR routine that describes the structure. When decoding data, storage is allocated if *pp is NULL.

There is no need for a primitive xdr_struct() to describe structures within structures, because pointers are always sufficient.

Exercise: Implement xdr_reference() using xdr_array(). Warning: xdr_reference() and xdr_array() are NOT interchangeable external representations of data.

*Example E*

Suppose there is a structure containing a person's name and a pointer to a gnumbers structure containing the person's gross assets and liabilities. The construct is:

```
struct pgn {
        char *name;
        struct gnumbers *gnp;
};
```

The corresponding XDR routine for this structure is:

```
bool_t
xdr_pgn(xdrs, pp)
        XDR *xdrs;
        struct pgn *pp;
{
        if (xdr_string(xdrs, &pp->name, NLEN) &&
            xdr_reference(xdrs, &pp->gnp, sizeof(struct gnumbers),
            xdr_gnumbers))
                return(TRUE);
        return(FALSE);
}
```

### *3.5.7.1. Pointer Semantics and XDR*

In many applications, C programmers attach double meaning to the values of a pointer. Typically the value NULL (or zero) means data is not needed, yet some application-specific interpretation applies. In essence, the C programmer is encoding a discriminated union efficiently by overloading the interpretation of the value of a pointer. For instance, in example E a NULL pointer value for gnp could indicate that the person's assets and liabilities are unknown. That is, the pointer value encodes two things: whether or not the data is known; and if it is known, where it is located in memory. Linked lists are an extreme example of the use of application-specific pointer interpretation.

The primitive xdr_reference() cannot and does not attach any special meaning to a NULL-value pointer during serialization. That is, passing an address of a pointer whose value is NULL to xdr_reference() when serialing data will most likely cause a memory fault and, on UNIX, a core dump for debugging.

It is the explicit responsibility of the programmer to expand non-dereferenceable pointers into their specific semantics. This usually involves describing data with a two-armed discriminated union. One arm is used when the pointer is valid; the other is used when the pointer is invalid (NULL). Section 7 has an example (linked lists encoding) that deals with invalid pointer interpretation.

Exercise: After reading Section 7, return here and extend example E so that it can correctly deal with null pointer values.

Exercise: Using the `xdr_union()`, `xdr_reference()` and `xdr_void()` primitives, implement a generic pointer handling primitive that implicitly deals with NULL pointers. The XDR library does not provide such a primitive because it does not want to give the illusion that pointers have meaning in the external world.

## 3.6. Non-filter Primitives

XDR streams can be manipulated with the primitives discussed in this section.

```
u_int xdr_getpos(xdrs)
        XDR *xdrs;

bool_t xdr_setpos(xdrs, pos)
        XDR *xdrs;
        u_int pos;

xdr_destroy(xdrs)
        XDR *xdrs;
```

The routine `xdr_getpos()` returns an unsigned integer that describes the current position in the data stream. Warning: In some XDR streams, the returned value of `xdr_getpos()` is meaningless; the routine returns a −1 in this case (though −1 should be a legitimate value).

The routine `xdr_setpos()` sets a stream position to `pos`. Warning: In some XDR streams, setting a position is impossible; in such cases, `xdr_setpos()` will return FALSE. This routine will also fail if the requested position is out-of-bounds. The definition of bounds varies from stream to stream.

The `xdr_destroy()` primitive destroys the XDR stream. Usage of the stream after calling this routine is undefined.

## 3.7. XDR Operation Directions

At times you may wish to optimize XDR routines by taking advantage of the direction of the operation (XDR_ENCODE, XDR_DECODE, or XDR_FREE). The value `xdrs->x_op` always contains the direction of the XDR operation. Programmers are not encouraged to take advantage of this information. Therefore, no example is presented here. However, an example in Section 7 demonstrates the usefulness of the `xdrs->x_op` field.

# 4. XDR Stream Access

An XDR stream is obtained by calling the appropriate creation routine. These creation routines take arguments that are tailored to the specific properties of the stream.

Streams currently exist for (de)serialization of data to or from standard I/O FILE streams, TCP/IP connections and UNIX files, and memory. Section 5 documents the XDR object and how to make new XDR streams when they are required.

## 4.1. Standard I/O Streams

XDR streams can be interfaced to standard I/O using the `xdrstdio_create()` routine as follows:

```
#include <stdio.h>
#include <rpc/rpc.h>  /* xdr streams are a part of the rpc library */

void
xdrstdio_create(xdrs, fp, x_op)
        XDR *xdrs;
        FILE *fp;
        enum xdr_op x_op;
```

The routine `xdrstdio_create()` initializes an XDR stream pointed to by `xdrs`. The XDR stream interfaces to the standard I/O library. Parameter `fp` is an open file, and `x_op` is an XDR direction.

## 4.2. Memory Streams

Memory streams allow the streaming of data into or out of a specified area of memory:

```
#include <rpc/rpc.h>

void
xdrmem_create(xdrs, addr, len, x_op)
        XDR *xdrs;
        char *addr;
        u_int len;
        enum xdr_op x_op;
```

The routine `xdrmem_create()` initializes an XDR stream in local memory. The memory is pointed to by parameter `addr`; parameter `len` is the length in bytes of the memory. The parameters `xdrs` and `x_op` are identical to the corresponding parameters of `xdrstdio_create()`. Currently, the UDP/IP implementation of RPC uses `xdrmem_create()`. Complete call or result messages are built in memory before calling the `sendto()` system routine.

## 4.3. Record (TCP/IP) Streams

A record stream is an XDR stream built on top of a record marking standard that is built on top of the UNIX file or 4.2 BSD connection interface.

```
#include <rpc/rpc.h>  /* xdr streams are a part of the rpc library */

xdrrec_create(xdrs, sendsize, recvsize, iohandle, readproc, writeproc)
        XDR *xdrs;
        u_int sendsize, recvsize;
        char *iohandle;
        int (*readproc)(), (*writeproc)();
```

The routine `xdrrec_create()` provides an XDR stream interface that allows for a bidirectional, arbitrarily long sequence of records. The contents of the records are meant to be data in XDR form. The stream's primary use is for interfacing RPC to TCP connections. However, it can be used to stream data into or out of normal UNIX files.

The parameter `xdrs` is similar to the corresponding parameter described above. The stream does its own data buffering similar to that of standard I/O. The parameters `sendsize` and `recvsize` determine the size in bytes of the output and input buffers, respectively; if their values are zero (0), then predetermined defaults are used. When a buffer needs to be filled or flushed, the routine `readproc` or `writeproc` is called, respectively. The usage and behavior of these routines are similar to the UNIX system calls `read()` and `write()`. However, the first parameter to each of these routines is the opaque parameter `iohandle`. The other two parameters (`buf` and `nbytes`) and the results (byte count) are identical to the system routines. If `xxx` is `readproc` or `writeproc`, then it has the following form:

```
/* returns the actual number of bytes transferred.
 * -1 is an error
 */
int
xxx(iohandle, buf, len)
        char *iohandle;
        char *buf;
        int nbytes;
```

The XDR stream provides means for delimiting records in the byte stream. The implementation details of delimiting records in a stream are discussed in appendix 1. The primitives that are specific to record streams are as follows:

```
bool_t
xdrrec_endofrecord(xdrs, flushnow)
        XDR *xdrs;
        bool_t flushnow;

bool_t
xdrrec_skiprecord(xdrs)
        XDR *xdrs;

bool_t
xdrrec_eof(xdrs)
        XDR *xdrs;
```

The routine `xdrrec_endofrecord()` causes the current outgoing data to be marked as a record. If the parameter `flushnow` is TRUE, then the stream's `writeproc()` will be called; otherwise, `writeproc()` will be called when the output buffer has been filled.

The routine `xdrrec_skiprecord()` causes an input stream's position to be moved past the current record boundary and onto the beginning of the next record in the stream.

If there is no more data in the stream's input buffer, then the routine `xdrrec_eof()` returns TRUE. That is not to say that there is no more data in the underlying file descriptor.

# 5. XDR Stream Implementation

This section provides the abstract data types needed to implement new instances of XDR streams.

## 5.1. The XDR Object

The following structure defines the interface to an XDR stream:

```
enum xdr_op { XDR_ENCODE = 0, XDR_DECODE = 1, XDR_FREE = 2 };

typedef struct {
        enum xdr_op     x_op;           /* operation; fast additional param */
        struct xdr_ops {
            bool_t  (*x_getlong)();     /* get a long from underlying stream */
            bool_t  (*x_putlong)();     /* put a long to " */
            bool_t  (*x_getbytes)();    /* get some bytes from " */
            bool_t  (*x_putbytes)();    /* put some bytes to " */
            u_int   (*x_getpostn)();    /* returns byte offset from beginning */
            bool_t  (*x_setpostn)();    /* repositions position in stream */
            caddr_t (*x_inline)();      /* buf quick ptr to buffered data */
            VOID    (*x_destroy)();     /* free privates of this xdr_stream */
        } *x_ops;
        caddr_t         x_public;       /* users' data */
        caddr_t         x_private;      /* pointer to private data */
        caddr_t         x_base;         /* private used for position info */
        int             x_handy;        /* extra private word */
} XDR;
```

The `x_op` field is the current operation being performed on the stream. This field is important to the XDR primitives, but should not affect a stream's implementation. That is, a stream's implementation should not depend on this value. The fields `x_private`, `x_base`, and `x_handy` are private to the particular stream's implementation. The field `x_public` is for the XDR client and should never be used by the XDR stream implementations or the XDR primitives.

Macros for accessing operations `x_getpostn()`, `x_setpostn()`, and `x_destroy()` were defined in Section 3.6. The operation `x_inline()` takes two parameters: an XDR *, and an unsigned integer, which is a byte count. The routine returns a pointer to a piece of the stream's internal buffer. The caller can then use the buffer segment for any purpose. From the stream's point of view, the bytes in the buffer segment have been consumed or put. The routine may return NULL if it cannot return a buffer segment of the requested size. (The `x_inline` routine is for cycle squeezers. Use of the resulting buffer is not data-portable. Users are encouraged not to use this feature.)

The operations `x_getbytes()` and `x_putbytes()` blindly get and put sequences of bytes from or to the underlying stream; they return TRUE if they are successful, and FALSE otherwise. The routines have identical parameters (replace xxx):

```
bool_t
xxxbytes(xdrs, buf, bytecount)
        XDR *xdrs;
        char *buf;
        u_int bytecount;
```

The operations x_getlong() and x_putlong() receive and put long numbers from and to the data stream. It is the responsibility of these routines to translate the numbers between the machine representation and the (standard) external representation. The UNIX primitives htonl() and ntohl() can be helpful in accomplishing this. Section 6 defines the standard representation of numbers. The higher-level XDR implementation assumes that signed and unsigned long integers contain the same number of bits, and that nonnegative integers have the same bit representations as unsigned integers. The routines return TRUE if they succeed, and FALSE otherwise. They have identical parameters:

```
bool_t
xxxlong(xdrs, lp)
        XDR *xdrs;
        long *lp;
```

Implementors of new XDR streams must make an XDR structure (with new operation routines) available to clients, using some kind of create routine.

# 6. XDR Standard

This section defines the external data representation standard. The standard is independent of languages, operating systems and hardware architectures. Once data is shared among machines, it should not matter that the data was produced on a Sun, but is consumed by a VAX (or vice versa). Similarly the choice of operating systems should have no influence on how the data is represented externally. For programming languages, data produced by a C program should be readable by a FORTRAN or Pascal program.

The external data representation standard depends on the assumption that bytes (or octets) are portable. A byte is defined to be eight bits of data. It is assumed that hardware that encodes bytes onto various media will preserve the bytes' meanings across hardware boundaries. For example, the Ethernet standard suggests that bytes be encoded "little endian" style. Both Sun and VAX hardware implementations adhere to the standard.

The XDR standard also suggests a language used to describe data. The language is a bastardized C; it is a data description language, not a programming language. (The Xerox Courier Standard uses bastardized Mesa as its data description language.)

## 6.1. Basic Block Size

The representation of all items requires a multiple of four bytes (or 32 bits) of data. The bytes are numbered 0 through $n-1$, where ($n$ mod 4)=0. The bytes are read or written to some byte stream such that byte $m$ always precedes byte $m+1$.

## 6.2. Integer

An XDR signed integer is a 32-bit datum that encodes an integer in the range `[-2147483648,2147483647]`. The integer is represented in two's complement notation. The most and least significant bytes are 0 and 3, respectively. The data description of integers is `integer`.

## 6.3. Unsigned Integer

An XDR unsigned integer is a 32-bit datum that encodes a nonnegative integer in the range `[0,4294967295]`. It is represented by an unsigned binary number whose most and least significant bytes are 0 and 3, respectively. The data description of unsigned integers is `unsigned`.

## 6.4. Enumerations

Enumerations have the same representation as integers. Enumerations are handy for describing subsets of the integers. The data description of enumerated data is as follows:

```
    typedef enum { name = value, .... } type-name;
```

For example the three colors red, yellow and blue could be described by an enumerated type:

```
        typedef enum { RED = 2, YELLOW = 3, BLUE = 5 } colors;
```

## 6.5. Booleans

Booleans are important enough and occur frequently enough to warrant their own explicit type in the standard. Boolean is an enumeration with the following form:

```
        typedef enum { FALSE = 0, TRUE = 1 } boolean;
```

## 6.6. Hyper Integer and Hyper Unsigned

The standard also defines 64-bit (8-byte) numbers called `hyper integer` and `hyper unsigned`. Their representations are the obvious extensions of the integer and unsigned defined above. The most and least significant bytes are 0 and 7, respectively.

## 6.7. Floating Point and Double Precision

The standard defines the encoding for the floating point data types `float` (32 bits or 4 bytes) and `double` (64 bits or 8 bytes). The encoding used is the IEEE standard for normalized single- and double-precision floating point numbers. See the IEEE floating point standard for more information. The standard encodes the following three fields, which describe the floating point number:

*S* The sign of the number. Values 0 and 1 represent positive and negative, respectively.

*E* The exponent of the number, base 2. Floats devote 8 bits to this field, while doubles devote 11 bits. The exponents for float and double are biased by 127 and 1023, respectively.

*F* The fractional part of the number's mantissa, base 2. Floats devote 23 bits to this field, while doubles devote 52 bits.

Therefore, the floating point number is described by:

$$(-1)^{S} * 2^{E-Bias} * 1.F$$

Just as the most and least significant bytes of a number are 0 and 3, the most and least significant bits of a single-precision floating point number are 0 and 31. The beginning bit (and most significant bit) offsets of *S*, *E*, and *F* are 0, 1, and 9, respectively.

Doubles have the analogous extensions. The beginning bit (and most significant bit) offsets of *S*, *E*, and *F* are 0, 1, and 12, respectively.

The IEEE specification should be consulted concerning the encoding for signed zero, signed infinity (overflow), and denormalized numbers (underflow). Under IEEE specifications, the "NaN" (not a number) is system dependent and should not be used.

## 6.8. Opaque Data

At times fixed-sized uninterpreted data needs to be passed among machines. This data is called `opaque` and is described as:

```
typedef opaque type-name[n];
opaque name[n];
```

where `n` is the (static) number of bytes necessary to contain the opaque data. If `n` is not a multiple of four, then the `n` bytes are followed by enough (up to 3) zero-valued bytes to make the total byte count of the opaque object a multiple of four.

## 6.9. Counted Byte Strings

The standard defines a string of $n$ (numbered 0 through $n-1$) bytes to be the number $n$ encoded as `unsigned`, and followed by the $n$ bytes of the string. If $n$ is not a multiple of four, then the $n$ bytes are followed by enough (up to 3) zero-valued bytes to make the total byte count a multiple of four. The data description of strings is as follows:

```
typedef string type-name<N>;
typedef string type-name<>;
string name<N>;
string name<>;
```

Note that the data description language uses angle brackets (< and >) to denote anything that is varying-length (as opposed to square brackets to denote fixed-length sequences of data).

The constant N denotes an upper bound of the number of bytes that a string may contain. If N is not specified, it is assumed to be $2^{32}-1$, the maximum length. The constant N would normally be found in a protocol specification. For example, a filing protocol may state that a file name can be no longer than 255 bytes, such as:

```
string filename<255>;
```

The XDR specification does not say what the individual bytes of a string represent; this important information is left to higher-level specifications. A reasonable default is to assume that the bytes encode ASCII characters.

## 6.10. Fixed Arrays

The data description for fixed-size arrays of homogeneous elements is as follows:

```
typedef elementtype type-name[n];
elementtype name[n];
```

Fixed-size arrays of elements numbered 0 through $n-1$ are encoded by individually encoding the elements of the array in their natural order, 0 through $n-1$.

## 6.11. Counted Arrays

Counted arrays provide the ability to encode varyiable-length arrays of homogeneous elements. The array is encoded as: the element count *n* (an unsigned integer), followed by the encoding of each of the array's elements, starting with element 0 and progressing through element *n*−1. The data description for counted arrays is similar to that of counted strings:

```
typedef elementtype type-name<N>;
typedef elementtype type-name<>;
elementtype name<N>;
elementtype name<>;
```

Again, the constant **N** specifies the maximum acceptable element count of an array; if **N** is not specified, it is assumed to be $2^{32}-1$.

## 6.12. Structures

The data description for structures is very similar to that of standard C:

```
typedef struct {
        component-type component-name;
        ...
} type-name;
```

The components of the structure are encoded in the order of their declaration in the structure.

## 6.13. Discriminated Unions

A discriminated union is a type composed of a discriminant followed by a type selected from a set of prearranged types according to the value of the discriminant. The type of the discriminant is always an enumeration. The component types are called "arms" of the union. The discriminated union is encoded as its discriminant followed by the encoding of the implied arm. The data description for discriminated unions is as follows:

```
typedef union switch (discriminant-type) {
        discriminant-value: arm-type;
        ...
        default: default-arm-type;
} type-name;
```

The default arm is optional. If it is not specified, then a valid encoding of the union cannot take on unspecified discriminant values. Most specifications neither need nor use default arms.

## 6.14. Missing Specifications

The standard lacks representations for bit fields and bitmaps, since the standard is based on bytes. This is not to say that no specification should be attempted.

## 6.15. Library Primitive / XDR Standard Cross Reference

The following table describes the association between the C library primitives discussed in Section 3, and the standard data types defined in this section:

| C Primitive | XDR Type | Sections |
|---|---|---|
| xdr_int<br>xdr_long<br>xdr_short | integer | 3.1, 6.2 |
| xdr_u_int<br>xdr_u_long<br>xdr_u_short | unsigned | 3.1, 6.3 |
| - | hyper integer<br>hyper unsigned | 6.6 |
| xdr_float | float | 3.2, 6.7 |
| xdr_double | double | 3.2, 6.7 |
| xdr_enum | enum_t | 3.3, 6.4 |
| xdr_bool | bool_t | 3.3, 6.5 |
| xdr_string<br>xdr_bytes | string | 3.5.1, 6.9<br>3.5.2 |
| xdr_array | (varying arrays) | 3.5.3, 6.11 |
| - | (fixed arrays) | 3.5.5, 6.10 |
| xdr_opaque | opaque | 3.5.4, 6.8 |
| xdr_union | union | 3.5.6, 6.13 |
| xdr_reference | - | 3.5.7 |
| - | struct | 6.6 |

# 7. Advanced Topics

This section describes techniques for passing data structures that are not covered in the preceding sections. Such structures include linked lists (of arbitrary lengths). Unlike the simpler examples covered in the earlier sections, the following examples are written using both the XDR C library routines and the XDR data description language. Section 6 describes the XDR data definition language used below.

## 7.1. Linked Lists

The last example in Section 2 presented a C data structure and its associated XDR routines for a person's gross assets and liabilities. The example is duplicated below:

```
struct gnumbers {
        long g_assets;
        long g_liabilities;
};

bool_t
xdr_gnumbers(xdrs, gp)
        XDR *xdrs;
        struct gnumbers *gp;
{
        if (xdr_long(xdrs, &(gp->g_assets)))
                return (xdr_long(xdrs, &(gp->g_liabilities)));
        return (FALSE);
}
```

Now assume that we wish to implement a linked list of such information. A data structure could be constructed as follows:

```
typedef struct gnnode {
        struct gnumbers gn_numbers;
        struct gnnode *nxt;
};

typedef struct gnnode *gnumbers_list;
```

The head of the linked list can be thought of as the data object; that is, the head is not merely a convenient shorthand for a structure. Similarly the `nxt` field is used to indicate whether or not the object has terminated. Unfortunately, if the object continues, the `nxt` field is also the address of where it continues. The link addresses carry no useful information when the object is serialized.

The XDR data description of this linked list is described by the recursive type declaration of gnumbers_list:

```
struct gnumbers {
        unsigned g_assets;
        unsigned g_liabilities;
};
```

```
typedef union switch (boolean) {
        case TRUE: struct {
                struct gnumbers current_element;
                gnumbers_list rest_of_list;
        };
        case FALSE: struct {};
} gnumbers_list;
```

In this description, the boolean indicates whether there is more data following it. If the boolean is FALSE, then it is the last data field of the structure. If it is TRUE, then it is followed by a `gnumbers` structure and (recursively) by a `gnumbers_list` (the rest of the object). Note that the C declaration has no boolean explicitly declared in it (though the `nxt` field implicitly carries the information), while the XDR data description has no pointer explicitly declared in it.

Hints for writing a set of XDR routines to successfully (de)serialize a linked list of entries can be taken from the XDR description of the pointer-less data. The set consists of the mutually recursive routines `xdr_gnumbers_list`, `xdr_wrap_list`, and `xdr_gnnode`.

```
bool_t
xdr_gnnode(xdrs, gp)
        XDR *xdrs;
        struct gnnode *gp;
{
        return (xdr_gnumbers(xdrs, &(gp->gn_numbers)) &&
                xdr_gnumbers_list(xdrs, &(gp->nxt)) );
}

bool_t
xdr_wrap_list(xdrs, glp)
        XDR *xdrs;
        gnumbers_list *glp;
{
        return (xdr_reference(xdrs, glp, sizeof(struct gnnode),
            xdr_gnnode));
}

struct xdr_discrim choices[2] = {
        /* called if another node needs (de)serializing */
        { TRUE, xdr_wrap_list },
         /* called when there are no more nodes to be (de)serialized */
        { FALSE, xdr_void }
}
bool_t
xdr_gnumbers_list(xdrs, glp)
        XDR *xdrs;
        gnumbers_list *glp;
{
        bool_t more_data;

        more_data = (*glp != (gnumbers_list)NULL);
        return (xdr_union(xdrs, &more_data, glp, choices, NULL);
}
```

The entry routine is `xdr_gnumbers_list()`; its job is to translate between the boolean value `more_data` and the list pointer values. If there is no more data, the `xdr_union()` primitive calls xdr_void and the recursion is terminated. Otherwise, xdr_union() calls xdr_wrap_list(), whose job is to dereference the list pointers. The xdr_gnnode() routine actually (de)serializes data of the current node of the linked list, and recursively calls xdr_gnumbers_list() to handle the remainder of the list.

You should convince yourself that these routines function correctly in all three directions (XDR_ENCODE, XDR_DECODE and XDR_FREE) for linked lists of any length (including zero). Note that the boolean more_data is always initialized, but in the XDR_DECODE case it is overwritten by an externally generated value. Also note that the value of the bool_t is lost in the stack. The essence of the value is reflected in the list's pointers.

The unfortunate side effect of (de)serializing a list with these routines is that the C stack grows linearly with respect to the number of nodes in the list. This is due to the recursion. The routines are also hard to code (and understand) due to the number and nature of primitives involved (such as xdr_reference, xdr_union, and xdr_void).

The following routine collapses the recursive routines. It also has other optimizations that are discussed below.

```
bool_t
xdr_gnumbers_list(xdrs, glp)
        XDR *xdrs;
        gnumbers_list *glp;
{
        bool_t more_data;

        while (TRUE) {
                more_data = (*glp != (gnumbers_list)NULL);
                if (! xdr_bool(xdrs, &more_data))
                        return (FALSE);
                if (! more_data)
                        return (TRUE);  /* we are done */
                if (! xdr_reference(xdrs, glp, sizeof(struct gnnode),
                    xdr_gnumbers))
                        return (FALSE);
                glp = &((*glp)->nxt);
        }
}
```

The claim is that this one routine is easier to code and understand than the three recursive routines above. (It is also buggy, as discussed below.) The parameter glp is treated as the address of the pointer to the head of the remainder of the list to be (de)serialized. Thus, glp is set to the address of the current node's nxt field at the end of the while loop. The discriminated union is implemented in-line; the variable more_data has the same use in this routine as in the routines above. Its value is recomputed and re-(de)serialized each iteration of the loop. Since *glp is a pointer to a node, the pointer is dereferenced using xdr_reference(). Note that the third parameter is truly the size of a node (data values plus nxt pointer), while xdr_gnumbers() only (de)serializes the data values. We can get away with this tricky optimization only because the nxt data comes after all legitimate external data.

The routine is buggy in the XDR_FREE case. The bug is that xdr_reference() will free the node *glp. Upon return the assignment glp = &((*glp)->nxt) cannot be guaranteed to work since *glp is no longer a legitimate node. The following is a rewrite that works in all cases. The hard part is to avoid dereferencing a pointer which has not been initialized or which has been freed.

```
bool_t
xdr_gnumbers_list(xdrs, glp)
        XDR *xdrs;
        gnumbers_list *glp;
{
        bool_t more_data;
        bool_t freeing;
        gnumbers_list *next;  /* the next value of glp */

        freeing = (xdrs->x_op == XDR_FREE);
        while (TRUE) {
                more_data = (*glp != (gnumbers_list)NULL);
                if (! xdr_bool(xdrs, &more_data))
                        return (FALSE);
                if (! more_data)
                        return (TRUE);  /* we are done */
                if (freeing)
                        next = &((*glp)->nxt);
                if (! xdr_reference(xdrs, glp, sizeof(struct gnnode),
                    xdr_gnumbers))
                        return (FALSE);
                glp = (freeing) ? next : &((*glp)->nxt);
        }
}
```

Note that this is the first example in this document that actually inspects the direction of the operation (xdrs->x_op). The claim is that the correct iterative implementation is still easier to understand or code than the recursive implementation. It is certainly more efficient with respect to C stack requirements.

# Appendix A. The Record Marking Standard

A record is composed of one or more record fragments. A record fragment is a four-byte header followed by 0 to $2^{31}-1$ bytes of fragment data. The bytes encode an unsigned binary number; as with XDR integers, the byte order is from highest to lowest. The number encodes two values — a boolean that indicates whether the fragment is the last fragment of the record (bit value 1 implies the fragment is the last fragment), and a 31-bit unsigned binary value which is the length in bytes of the fragment's data. The boolean value is the high-order bit of the header; the length is the 31 low-order bits.

(Note that this record specification is `not` in XDR standard form and cannot be implemented using XDR primitives!)

# Appendix B. Synopsis of XDR Routines

### xdr_array()

```
xdr_array(xdrs, arrp, sizep, maxsize, elsize, elproc)
        XDR *xdrs;
        char **arrp;
        u_int *sizep, maxsize, elsize;
        xdrproc_t elproc;
```

A filter primitive that translates between arrays and their corresponding external representations. The parameter **arrp** is the address of the pointer to the array, while **sizep** is the address of the element count of the array; this element count cannot exceed **maxsize**. The parameter **elsize** is the **sizeof()** each of the array's elements, and **elproc** is an XDR filter that translates between the array elements' C form, and their external representation. This routine returns one if it succeeds, zero otherwise.

### xdr_bool()

```
xdr_bool(xdrs, bp)
        XDR *xdrs;
        bool_t *bp;
```

A filter primitive that translates between booleans (C integers) and their external representations. When encoding data, this filter produces values of either one or zero. This routine returns one if it succeeds, zero otherwise.

### xdr_bytes()

```
xdr_bytes(xdrs, sp, sizep, maxsize)
        XDR *xdrs;
        char **sp;
        u_int *sizep, maxsize;
```

A filter primitive that translates between counted byte strings and their external representations. The parameter **sp** is the address of the string pointer. The length of the string is located at address **sizep**; strings cannot be longer than **maxsize**. This routine returns one if it succeeds, zero otherwise.

### xdr_destroy()

```
void
xdr_destroy(xdrs)
        XDR *xdrs;
```

A macro that invokes the destroy routine associated with the XDR stream, **xdrs**. Destruction usually involves freeing private data structures associated with the stream. Using **xdrs** after invoking **xdr_destroy()** is undefined.

### xdr_double()

```
xdr_double(xdrs, dp)
        XDR *xdrs;
        double *dp;
```

A filter primitive that translates between C **double** precision numbers and their external representations. This routine returns one if it succeeds, zero otherwise.

### xdr_enum()

```
xdr_enum(xdrs, ep)
        XDR *xdrs;
        enum_t *ep;
```

A filter primitive that translates between C **enums** (actually integers) and their external representations. This routine returns one if it succeeds, zero otherwise.

### xdr_float()

```
xdr_float(xdrs, fp)
        XDR *xdrs;
        float *fp;
```

A filter primitive that translates between C **floats** and their external representations. This routine returns one if it succeeds, zero otherwise.

### xdr_getpos()

```
u_int
xdr_getpos(xdrs)
        XDR *xdrs;
```

A macro that invokes the get-position routine associated with the XDR stream, **xdrs**. The routine returns an unsigned integer, which indicates the position of the XDR byte stream. A desirable feature of XDR streams is that simple arithmetic works with this number, although the XDR stream instances need not guarantee this.

### xdr_inline()

```
long *
xdr_inline(xdrs, len)
        XDR *xdrs;
        int len;
```

A macro that invokes the in-line routine associated with the XDR stream, **xdrs**. The routine returns a pointer to a contiguous piece of the stream's buffer; **len** is the byte length of the desired buffer. Note that the pointer is cast to **long ***. Warning: **xdr_inline()** may return 0 (NULL) if it cannot allocate a contiguous piece of a buffer. Therefore the behavior may vary among stream instances; it exists for the sake of efficiency.

### xdr_int()

```
xdr_int(xdrs, ip)
        XDR *xdrs;
        int *ip;
```

A filter primitive that translates between C integers and their external representations. This routine returns one if it succeeds, zero otherwise.

### xdr_long()

```
xdr_long(xdrs, lp)
        XDR *xdrs;
        long *lp;
```

A filter primitive that translates between C `long` integers and their external representations. This routine returns one if it succeeds, zero otherwise.

### xdr_opaque()

```
xdr_opaque(xdrs, cp, cnt)
        XDR *xdrs;
        char *cp;
        u_int cnt;
```

A filter primitive that translates between fixed size opaque data and its external representation. The parameter `cp` is the address of the opaque object, and `cnt` is its size in bytes. This routine returns one if it succeeds, zero otherwise.

### xdr_reference()

```
xdr_reference(xdrs, pp, size, proc)
        XDR *xdrs;
        char **pp;
        u_int size;
        xdrproc_t proc;
```

A primitive that provides pointer chasing within structures. The parameter `pp` is the address of the pointer; `size` is the `sizeof()` the structure that `*pp` points to; and `proc` is an XDR procedure that filters the structure between its C form and its external representation. This routine returns one if it succeeds, zero otherwise.

### xdr_setpos()

```
xdr_setpos(xdrs, pos)
        XDR *xdrs;
        u_int pos;
```

A macro that invokes the set position routine associated with the XDR stream `xdrs`. The parameter `pos` is a position value obtained from `xdr_getpos()`. This routine returns one if the XDR stream could be repositioned, and zero otherwise. Warning: it is difficult to reposition some types of XDR streams, so this routine may fail with one type of stream and succeed with another.

### xdr_short()

```
xdr_short(xdrs, sp)
        XDR *xdrs;
        short *sp;
```

A filter primitive that translates between C **short** integers and their external representations. This routine returns one if it succeeds, zero otherwise.

### xdr_string()

```
xdr_string(xdrs, sp, maxsize)
        XDR *xdrs;
        char **sp;
        u_int maxsize;
```

A filter primitive that translates between C strings and their corresponding external representations. Strings cannot cannot be longer than **maxsize**. Note that **sp** is the address of the string's pointer. This routine returns one if it succeeds, zero otherwise.

### xdr_u_int()

```
xdr_u_int(xdrs, up)
        XDR *xdrs;
        unsigned *up;
```

A filter primitive that translates between C **unsigned** integers and their external representations. This routine returns one if it succeeds, zero otherwise.

### xdr_u_long()

```
xdr_u_long(xdrs, ulp)
        XDR *xdrs;
        unsigned long *ulp;
```

A filter primitive that translates between C **unsigned long** integers and their external representations. This routine returns one if it succeeds, zero otherwise.

### xdr_u_short()

```
xdr_u_short(xdrs, usp)
        XDR *xdrs;
        unsigned short *usp;
```

A filter primitive that translates between C **unsigned short** integers and their external representations. This routine returns one if it succeeds, zero otherwise.

### xdr_union()

```
xdr_union(xdrs, dscmp, unp, choices, dfault)
        XDR *xdrs;
        int *dscmp;
        char *unp;
        struct xdr_discrim *choices;
        xdrproc_t dfault;
```

A filter primitive that translates between a discriminated C **union** and its corresponding external representation. The parameter **dscmp** is the address of the union's discriminant, while **unp** in the address of the union. This routine returns one if it succeeds, zero otherwise.

### xdr_void()

```
xdr_void()
```

This routine always returns one. It may be passed to RPC routines that require a function parameter, where nothing is to be done.

### xdr_wrapstring()

```
xdr_wrapstring(xdrs, sp)
        XDR *xdrs;
        char **sp;
```

A primitive that calls **xdr_string(xdrs,sp,MAXUNSIGNED);** where MAXUNSIGNED is the maximum value of an unsigned integer. This is handy because the RPC package passes only two parameters XDR routines, whereas **xdr_string()**, one of the most frequently used primitives, requires three parameters. This routine returns one if it succeeds, zero otherwise.

### xdrmem_create()

```
void
xdrmem_create(xdrs, addr, size, op)
        XDR *xdrs;
        char *addr;
        u_int size;
        enum xdr_op op;
```

This routine initializes the XDR stream object pointed to by **xdrs**. The stream's data is written to, or read from, a chunk of memory at location **addr** whose length is no more than **size** bytes long. The **op** determines the direction of the XDR stream (either XDR_ENCODE, XDR_DECODE, or XDR_FREE).

### xdrrec_create()

```
void
xdrrec_create(xdrs, sendsize, recvsize, handle, readit, writeit)
        XDR *xdrs;
        u_int sendsize, recvsize;
        char *handle;
        int (*readit)(), (*writeit)();
```

This routine initializes the XDR stream object pointed to by **xdrs**. The stream's data is written to a buffer of size **sendsize**; a value of zero indicates the system should use a suitable default. The stream's data is read from a buffer of size **recvsize**; it too can be set to a suitable default by passing a zero value. When a stream's output buffer is full, **writeit()** is called. Similarly, when a stream's input buffer is empty, **readit()** is called. The behavior of these two routines is similar to the UNIX system calls **read** and **write**, except that **handle** is passed to the former routines as the first parameter. Note that the XDR stream's **op** field must be set by the caller. Warning: this XDR stream implements an intermediate record stream. Therefore there are additional bytes in the stream to provide record boundary information.

### xdrrec_endofrecord()

```
xdrrec_endofrecord(xdrs, sendnow)
        XDR *xdrs;
        int sendnow;
```

This routine can be invoked only on streams created by **xdrrec_create()**. The data in the output buffer is marked as a completed record, and the output buffer is optionally written out if **sendnow** is non-zero. This routine returns one if it succeeds, zero otherwise.

### xdrrec_eof()

```
xdrrec_eof(xdrs)
        XDR *xdrs;
        int empty;
```

This routine can be invoked only on streams created by **xdrrec_create()**. After consuming the rest of the current record in the stream, this routine returns one if the stream has no more input, zero otherwise.

### xdrrec_skiprecord()

```
xdrrec_skiprecord(xdrs)
        XDR *xdrs;
```

This routine can be invoked only on streams created by **xdrrec_create()**. It tells the XDR implementation that the rest of the current record in the stream's input buffer should be discarded. This routine returns one if it succeeds, zero otherwise.

### xdrstdio_create()

```
void
xdrstdio_create(xdrs, file, op)
        XDR *xdrs;
        FILE *file;
        enum xdr_op op;
```

This routine initializes the XDR stream object pointed to by **xdrs**. The XDR stream data is written to, or read from, the Standard I/O stream **file**. The parameter **op** determines the direction of the XDR stream (either XDR_ENCODE, XDR_DECODE, or XDR_FREE). Warning: the destroy routine associated with such XDR streams calls **fflush()** on the **file** stream, but never **fclose()**.

# Remote Procedure Call Protocol Specification

# Contents

# Remote Procedure Call
# Protocol Specification

## 1. Introduction

This document specifies a message protocol used in implementing Sun's Remote Procedure Call (RPC) package. The message protocol is specified with the eXternal Data Representation (XDR) language.

This document assumes that the reader is familiar with both RPC and XDR. It does not attempt to justify RPC or its uses. Also, the casual user of RPC does not need to be familiar with the information in this document.

## 1.1. Terminology

The document discusses servers, services, programs, procedures, clients and versions. A server is a machine where some number of network services are implemented. A service is a collection of one or more remote programs. A remote program implements one or more remote procedures; the procedures, their parameters and results are documented in the specific program's protocol specification (see Appendix C for an example). Network clients are pieces of software that initiate remote procedure calls to services. A server may support more than one version of a remote program in order to be forward compatible with changing protocols.

For example, a network file service may be composed of two programs. One program may deal with high level applications such as file system access control and locking. The other may deal with low-level file I/O, and have procedures like "read" and "write". A client machine of the network file service would call the procedures associated with the two programs of the service on behalf of some user on the client machine.

## 1.2. The RPC Model

The remote procedure call model is similar to the local procedure call model. In the local case, the caller places arguments to a procedure in some well-specified location (such as a result register). It then transfers control to the procedure, and eventually gains back control. At that point, the results of the procedure are extracted from the well-specified location, and the caller continues execution.

The remote procedure call is similar, except that one thread of control winds through two processes — one is the caller's process, the other is a server's process. That is, the caller process sends a call message to the server process and waits (blocks) for a reply message. The call message contains the procedure's parameters, among other things. The reply message contains the procedure's results, among other things. Once the reply message is received, the results of the procedure are extracted, and caller's execution is resumed.

On the server side, a process is dormant awaiting the arrival of a call message. When one arrives the server process extracts the procedure's parameters, computes the results, sends a reply message, and then awaits the next call message. Note that in this model, only one of the two processes is active at any given time. That is, the RPC protocol does not explicitly support multi-threading of caller or server processes.

## 1.3. Transports and Semantics

The RPC protocol is independent of transport protocols. That is, RPC does not care how a message is passed from one process to another. The protocol only deals with the specification and interpretation of messages.

Because of transport independence, the RPC protocol does not attach specific semantics to the remote procedures or their execution. Some semantics can be inferred from (but should be explicitly specified by) the underlying transport protocol. For example, RPC message passing using UDP/IP is unreliable. Thus, if the caller retransmits call messages after short time-outs, the only thing he can infer from no reply message is that the remote procedure was executed zero or more times (and from a reply message, one or more times). On the other hand, RPC message passing using TCP/IP is reliable. No reply message means that the remote procedure was executed at most once, whereas a reply message means that the remote procedure was exactly once. (Note: At Sun, RPC is currently implemented on top of TCP/IP and UDP/IP transports.)

## 1.4. Binding and Rendezvous Independence

The act of binding a client to a service is NOT part of the remote procedure call specification. This important and necessary function is left up to some higher level software. (The software may use RPC itself; see Appendix C.)

Implementors should think of the RPC protocol as the jump-subroutine instruction ("JSR") of a network; the loader (binder) makes JSR useful, and the loader itself uses JSR to accomplish its task. Likewise, the network makes RPC useful, using RPC to accomplish this task.

## 1.5. Message Authentication

The RPC protocol provides the fields necessary for a client to identify himself to a service and vice versa. Security and access control mechanisms can be built on top of the message authentication.

## 2. Requirements

The RPC protocol must provide for the following:

1. Unique specification of a procedure to be called.
2. Provisions for matching response messages to request messages.
3. Provisions for authenticating the caller to service and vice versa.

Besides these requirements, features that detect the following are worth supporting because of protocol roll-over errors, implementation bugs, user error, and network administration:

1. RPC protocol mismatches.
2. Remote program protocol version mismatches.
3. Protocol errors (like mis-specification of a procedure's parameters).
4. Reasons why remote authentication failed.
5. Any other reasons why the desired procedure was not called.

### 2.1. Remote Programs and Procedures

The RPC call message has three unsigned fields: remote program number, remote program version number, and remote procedure number. The three fields uniquely identify the procedure to be called. Program numbers are administered by some central authority (like Sun). Once an implementor has a program number, he can implement his remote program; the first implementation would most likely have the version number of 1. Because most new protocols evolve into better, stable and mature protocols, a version field of the call message identifies which version of the protocol the caller is using. Version numbers make speaking old and new protocols through the same server process possible.

The procedure number identifies the procedure to be called. These numbers are documented in the specific program's protocol specification. For example, a file service's protocol specification may state that its procedure number 5 is `read` and procedure number 12 is `write`.

Just as remote program protocols may change over several versions, the actual RPC message protocol could also change. Therefore, the call message also has the RPC version number in it; this field must be two (2).

The reply message to a request message has enough information to distinguish the following error conditions:

1) The remote implementation of RPC does speak protocol version 2. The lowest and highest supported RPC version numbers are returned.

2) The remote program is not available on the remote system.

3) The remote program does not support the requested version number. The lowest and highest supported remote program version numbers are returned.

4) The requested procedure number does not exist (this is usually a caller side protocol or programming error).

5) The parameters to the remote procedure appear to be garbage from the server's point of view. (Again, this is caused by a disagreement about the protocol between client and service.)

## 2.2. Authentication

Provisions for authentication of caller to service and vice versa are provided as a wart on the side of the RPC protocol. The call message has two authentication fields, the credentials and verifier. The reply message has one authentication field, the response verifier. The RPC protocol specification defines all three fields to be the following opaque type:

```
enum auth_flavor {
        AUTH_NULL       = 0,
        AUTH_UNIX       = 1,
        AUTH_SHORT      = 2
        /* and more to be defined */
};

struct opaque_auth {
        union switch (enum auth_flavor) {
                default: string auth_body<400>;
        };
};
```

In simple English, any `opaque_auth` structure is an `auth_flavor` enumeration followed by a counted string, whose bytes are opaque to the RPC protocol implementation.

The interpretation and semantics of the data contained within the authentication fields is specified by individual, independent authentication protocol specifications. Appendix A defines three authentication protocols.

If authentication parameters were rejected, the response message contains information stating why they were rejected.

## 2.3. Program Number Assignment

Program numbers are given out in groups of 0x20000000 (536870912) according to the following chart:

| | |
|---|---|
| 0 - 1fffffff | defined by Sun |
| 20000000 - 3fffffff | defined by user |
| 40000000 - 5fffffff | transient |
| 60000000 - 7fffffff | reserved |
| 80000000 - 9fffffff | reserved |
| a0000000 - bfffffff | reserved |
| c0000000 - dfffffff | reserved |
| e0000000 - ffffffff | reserved |

The first group is a range of numbers administered by Sun Microsystems, and should be identical for all Sun customers. The second range is for applications peculiar to a particular customer. This range is intended primarily for debugging new programs. When a customer develops an application that might be of general interest, that application should be given an assigned number in the first range. The third group is for applications that generate program numbers dynamically. The final groups are reservered for future use, and should not be used.

The exact registration process for Sun defined numbers is yet to be established.

# 3. Other Uses and Abuses of the RPC Protocol

The intended use of this protocol is for calling remote procedures. That is, each call message is matched with a response message. However, the protocol itself is a message passing protocol with which other (non-RPC) protocols can be implemented. Sun currently uses (abuses) the RPC message protocol for the following two (non-RPC) protocols: batching (or pipelining) and broadcast RPC. These two protocols are discussed (but not defined) below.

## 3.1. Batching

Batching allows a client to send an arbitrarily large sequence of call messages to a server; batching uses reliable bytes stream protocols (like TCP/IP) for their transport. In the case of batching, the client never waits for a reply from the server and the server does not send replies to batch requests. A sequence of batch calls is usually terminated by a legitimate RPC in order to flush the pipeline (with positive acknowledgement).

## 3.2. Broadcast RPC

In broadcast RPC based protocols, the client sends an a broadcast packet to the network and waits for numerous replies. Broadcast RPC uses unreliable, packet based protocols (like UDP/IP) as their transports. Servers that support broadcast protocols only respond when the request is successfully processed, and are silent in the face of errors.

# 4. The RPC Message Protocol

This section defines the RPC message protocol in the XDR data description language. The message is defined in a top down style. Note: This is an XDR specification, not C code.

```
enum msg_type {
        CALL = 0,
        REPLY = 1
};

/*
 * A reply to a call message can take on two forms:
 * the message was either accepted or rejected.
 */
enum reply_stat {
        MSG_ACCEPTED = 0,
        MSG_DENIED = 1
};
```

```
/*
 * Given that a call message was accepted, the following is the status of
 * an attempt to call a remote procedure.
 */
enum accept_stat {
        SUCCESS = 0,       /* remote procedure was successfully executed */
        PROG_UNAVAIL = 1,  /* remote machine exports the program number */
        PROG_MISMATCH = 2, /* remote machine can't support version number */
        PROC_UNAVAIL = 3,  /* remote program doesn't know about procedure */
        GARBAGE_ARGS = 4   /* remote procedure can't figure out parameters */
};

/*
 * Reasons why a call message was rejected:
 */
enum reject_stat {
        RPC_MISMATCH = 0,  /* RPC version number was not two (2) */
        AUTH_ERROR = 1     /* caller not authenticated on remote machine */
};

/*
 * Why authentication failed:
 */
enum auth_stat {
        AUTH_BADCRED = 1,       /* bogus credentials (seal broken) */
        AUTH_REJECTEDCRED = 2,  /* client should begin new session */
        AUTH_BADVERF = 3,       /* bogus verifier (seal broken) */
        AUTH_REJECTEDVERF = 4,  /* verifier expired or was replayed */
        AUTH_TOOWEAK = 5,       /* rejected due to security reasons */
};

/*
 * The RPC message:
 * All messages start with a transaction identifier, xid,  followed by
 * a two-armed discriminated union.  The union's discriminant is a msg_type
 * which switches to one of the two types of the message.  The xid of a
 * REPLY message always matches that of the initiating CALL message.
 * NB: The xid field is only used for clients matching reply messages with
 * call messages; the service side cannot treat this id as any type of
 * sequence number.
 */
struct rpc_msg {
        unsigned        xid;
        union switch (enum msg_type) {
                CALL:   struct call_body;
                REPLY:  struct reply_body;
        };
};
```

```
/*
 * Body of an RPC request call:
 * In version 2 of the RPC protocol specification, rpcvers must be equal to 2.
 * The fields prog, vers, and proc specify the remote program, its version,
 * and the procedure within the remote program to be called.  These fields are
 * followed by two authentication parameters, cred (authentication credentials)
 * and verf (authentication verifier).  The authentication parameters are
 * followed * by the parameters to the remote procedure; these parameters are
 * specified by the specific program protocol.
 */
struct call_body {
        unsigned rpcvers;       /* must be equal to two (2) */
        unsigned prog;
        unsigned vers;
        unsigned proc;
        struct opaque_auth cred;
        struct opaque_auth verf;
        /* procedure specific parameters start here */
};

/*
 * Body of a reply to an RPC request.
 * The call message was either accepted or rejected.
 */
struct reply_body {
        union switch (enum reply_stat) {
                MSG_ACCEPTED:   struct accepted_reply;
                MSG_DENIED:     struct rejected_reply;
        };
};
```

```
/*
 * Reply to an RPC request that was accepted by the server.
 * Note: there could be an error even though the request was accepted.
 * The first field is an authentication verifier which the server generates
 * in order to validate itself to the caller.  It is followed by a union
 * whose discriminant is an enum accept_stat.  The SUCCESS arm of the union is
 * protocol specific.  The PROG_UNAVAIL, PROC_UNAVAIL, and GARBAGE_ARGS arms
 * of the union are void.  The PROG_MISMATCH arm specifies the lowest and
 * highest version numbers of the remote program that are supported by the
 * server.
 */
struct accepted_reply {
        struct opaque_auth      verf;
        union switch (enum accept_stat) {
                SUCCESS: struct {
                        /*
                         * procedure-specific results start here
                         */
                };
                PROG_MISMATCH: struct {
                        unsigned low;
                        unsigned  high;
                };
                default: struct {
                        /*
                         * void. Cases include PROG_UNAVAIL,
                         * PROC_UNAVAIL, and GARBAGE_ARGS.
                         */
                };
        };
};

/*
 * Reply to an RPC request that was rejected by the server.
 * The request can be rejected because of two reasons - either the server is
 * not running a compatible version of the RPC protocol (RPC_MISMATCH), or
 * the server refused to authenticate the caller (AUTH_ERROR).  In the case of
 * an RPC version mismatch, the server returns the lowest and highest supported
 * RPC version numbers. In the case of refused authentication, the failure
 * status is returned.
 */
struct rejected_reply {
        union switch (enum reject_stat) {
                RPC_MISMATCH: struct {
                        unsigned low;
                        unsigned high;
                };
                AUTH_ERROR: enum auth_stat;
        };
};
```

# Appendix A. Authentication Parameter Specification

As previously stated, authentication parameters are opaque, but open-ended to the rest of the RPC protocol. This section defines some "flavors" of authentication which have been implemented at (and supported by) Sun.

## A.1. Null Authentication

Often calls must be made where the caller does not know who he is and the server does not care who the caller is. In this case, the auth_flavor value (the discriminant of the opaque_auth's union) of the RPC message's credentials, verifier, and response verifier is AUTH_NULL (0). The bytes of the auth_body string are undefined. It is recommended that the string length be zero.

## A.2. UNIX Authentication

The caller of a remote procedure may wish to identify himself as he is identified on a UNIX† system. The value of the *credential's* discriminant of an RPC call message is AUTH_UNIX (1). The bytes of the *credential's* string encode the the following (XDR) structure:

```
struct auth_unix {
        unsigned        stamp;
        string          machinename<255>;
        unsigned        uid;
        unsigned        gid;
        unsigned        gids<10>;
};
```

The `stamp` is an arbitrary id which the caller machine may generate. The `machinename` is the name of the caller's machine (like "krypton"). The `uid` is the caller's effective user id. The `gid` is the callers effective group id. The `gids` is a counted array of groups which contain the caller as a member. The *verifier* accompanying the credentials should be of AUTH_NULL (defined above).

The value of the discriminate of the *response verifier* received in the reply message from the server may be AUTH_NULL or AUTH_SHORT (2). In the case of AUTH_SHORT, the bytes of the *response verifier*'s string encode an `auth_opaque` structure. This new `auth_opaque` structure may now be passed to the server instead of the original AUTH_UNIX flavor credentials. The server keeps a cache which maps short hand *auth_opaque* structures (passed back via a AUTH_SHORT style `response verifier`) to the original credentials of the caller. The caller can save network bandwidth and server cpu cycles by using the new credentials.

The server may flush the short hand *auth_opaque* structure at any time. If this happens, the remote procedure call message will be rejected due to an authentication error. The reason for the failure will be AUTH_REJECTEDCRED. At this point, the caller may wish to try the original AUTH_UNIX style of credentials.

† UNIX is a trademark of Bell Laboratories.

# Appendix B. Record Marking Standard

When RPC messages are passed on top of a byte stream protocol (like TCP/IP), it is necessary, or at least desirable, to delimit one message from another in order to detect and possibly recover from user protocol errors. This is called record marking (RM). Sun uses this RM/TCP/IP transport for passing RPC messages on TCP streams. One RPC message fits into one RM record.

A record is composed of one or more record fragments. A record fragment is a four-byte header followed by $0$ to $2^{31}-1$ bytes of fragment data. The bytes encode an unsigned binary number; as with XDR integers, the byte order is from highest to lowest. The number encodes two values — a boolean which indicates whether the fragment is the last fragment of the record (bit value 1 implies the fragment is the last fragment) and a 31-bit unsigned binary value which is the length in bytes of the fragment's data. The boolean value is the highest-order bit of the header; the length is the 31 low-order bits. (Note that this record specification is ***not*** in XDR standard form!)

# Appendix C. Port Mapper Program Protocol

The port mapper program maps RPC program and version numbers to UDP/IP or TCP/IP port numbers. This program makes dynamic binding of remote programs possible.

This is desirable because the range of reserved port numbers is very small and the number of potential remote programs is very large. By running only the port mapper on a reserved port, the port numbers of other remote programs can be ascertained by querying the port mapper.

## C.1. The Port Mapper RPC Protocol

The protocol is specified by the XDR description language.

```
Port Mapper RPC Program Number: 100000
        Version Number: 1
        Supported Transports:
                UDP/IP on port 111
                RM/TCP/IP on port 111

/*
 * Handy transport protocol numbers
 */
#define IPPROTO_TCP     6       /* protocol number used for rpc/rm/tcp/ip */
#define IPPROTO_UDP     17      /* protocol number used for rpc/udp/ip */

/* Procedures */

/*
 * Convention: procedure zero of any protocol takes no parameters
 * and returns no results.
 */
0. PMAPPROC_NULL () returns ()

/*
 * Procedure 1, setting a mapping:
 * When a program first becomes available on a
 * machine, it registers itself with the port mapper program on the
 * same machine.  The program passes its program number (prog),
 * version number (vers), transport protocol number (prot),
 * and the port (port) on which it awaits service request.  The
 * procedure returns success whose value is TRUE if the procedure
 * successfully established the mapping and FALSE otherwise.  The
 * procedure will refuse to establish a mapping if one already exists
 * for the tuple [prog, vers, prot].
 */
1. PMAPPROC_SET (prog, vers, prot, port) returns (success)
        unsigned prog;
        unsigned vers;
        unsigned prot;
        unsigned port;
        boolean success;
```

```
/*
 * Procedure 2, Unsetting a mapping:
 * When a program becomes unavailable, it should unregister itself
 * with the port mapper program on the same machine.  The parameters
 * and results have meanings identical to those of PMAPPROC_SET.
 */
2. PMAPPROC_UNSET (prog, vers, dummy1, dummy2) returns (success)
        unsigned prog;
        unsigned vers;
        unsigned dummy1;  /* this value is always ignored */
        unsigned dummy2;  /* this value is always ignored */
        boolean success;

/*
 * Procedure 3, looking-up a mapping:
 * Given a program number (prog), version number (vers) and
 * transport protocol number (prot), this procedure returns the port
 * number on which the program is awaiting call requests.  A port
 * value of zeros means that the program has not been registered.
 */
3. PMAPPROC_GETPORT (prog, vers, prot, dummy) returns (port)
        unsigned prog;
        unsigned vers;
        unsigned prot;
        unsigned dummy;  /* this value is always ignored */
        unsigned port;  /* zero means the program is not registered */

/*
 * Procedure 4, dumping the mappings:
 * This procedure enumerates all entries in the port mapper's database.
 * The procedure takes no parameters and returns a ``list'' of
 * [program, version, prot, port] values.
 */
4. PMAPPROC_DUMP () returns (maplist)
        struct maplist {
                union switch (boolean) {
                        FALSE: struct { /* void, end of list */ };
                        TRUE: struct {
                                unsigned prog;
                                unsigned vers;
                                unsigned prot;
                                unsigned port;
                                struct maplist the_rest;
                        };
                };
        } maplist;
```

```
/*
 * Procedure 5, indirect call routine:
 * The procedures allows a caller to call another remote procedure
 * on the same machine without knowing the remote procedure's port
 * number.  Its intended use is for supporting broadcasts to arbitrary
 * remote programs via the well-known port mapper's port.  The parameters
 * prog, vers, proc, and the bytes of args are the program number,
 * version number, procedure number, and  parameters the the remote
 * procedure.
 *
 * NB:
 * 1. This procedure only sends a response if the procedure was
 * successfully executed and is silent (No response) otherwise.
 * 2. The port mapper communicates with the remote program via
 * UDP/IP only.
 *
 * The procedure returns the port number of the remote program and
 * the bytes of results are the results of the remote procedure.
 */
5. PMAPPROC_CALLIT (prog, vers, proc, args) returns (port, results)
        unsigned prog;
        unsigned vers;
        unsigned proc;
        string args<>;
        unsigned port;
        string results<>;
```

# Network File System Protocol Specification

# Contents

# Network File System
# Protocol Specification

## 1. Introduction

The Sun Network Filesystem (NFS) protocol provides transparent remote access to shared filesystems over local area networks. The NFS protocol is designed to be machine, operating system, network architecture, and transport protocol independent. This independence is achieved through the use of Remote Procedure Call (RPC) primitives built on top of an eXternal Data Representation (XDR).

The supporting mount protocol allows the server to hand out remote access privileges to a restricted set of clients. Thus, it allows clients to attach a remote directory tree at any point on some local filesystem.

## 1.1. Remote Procedure Call

Sun's remote procedure call specification, described in the *RPC Programming Guide*, provides a clean, procedure-oriented interface to remote services. Each server supplies a program that is a set of procedures. The combination of host address, program number, and procedure number specifies one remote service procedure.

RPC is a high-level protocol built on top of low-level transport protocols. It does not depend on services provided by specific protocols, so it can be used easily with any underlying transport protocol. Currently the only supported transport protocol is UDP/IP.

The RPC protocol includes a slot for authentication parameters on every call. The contents of the authentication parameters are determined by the "flavor" (type) of authentication used by the server and client. A server may support several different flavors of authentication at once: AUTH_NONE passes no authentication information (this is called null authentication); AUTH_UNIX passes the UNIX† uid, gid, and groups with each call.

Servers have been known to change over time, and so can the protocol that they use. So RPC provides a version number with each RPC request. Thus, one server can service requests for several different versions of the protocol at the same time.

## 1.2. External Data Representation

Sun's external data representation specification, described in the *XDR Protocol Specification*, provides a common way of representing a set of data types over a network. This takes care of problems such as different byte ordering on different communicating machines. It also defines

† UNIX is a trademark of Bell Laboratories.

the size of each data type so that machines with different structure alignment algorithms can share a common format over the network.

In this document we use the XDR data definition language to specify the parameters and results of each RPC service procedure that a NFS server provides. The XDR data definition language reads a lot like C, although a few new constructs have been added. The notation

```
string  name[SIZE];
string  data<DSIZE>;
```

defines `name`, which is a fixed size block of `SIZE` bytes, and `data`, which is a variable size block of up to DSIZE bytes. This same notation is used to indicate fixed length arrays, and arrays with a variable number of elements up to some maximum.

The discriminated union definition

```
union switch (enum status) {
    NFS_OK:
        struct {
                filename        file1;
                filename        file2;
                integer         count;
        }
    NFS_ERROR:
        struct {
                errstat         error;
                integer         errno;
        }
    default:
        struct {}
}
```

means the first thing over the network is an enumeration type called `status`; if its value is NFS_OK, the next thing on the network will be the structure containing `file1`, `file2`, and `count`. If the value of `status` is neither NFS_OK nor NFS_ERROR, then there is no more data to look at.

## 1.3. Stateless Servers

The NFS protocol is stateless. That is, a server does not need to maintain state about any of its clients in order to function correctly. Stateless servers have a distinct advantage over stateful servers in the event of a crash. With stateless servers, a client need only retry a request until the server responds; it does not even need to know that the server has crashed. The client of a stateful server, on the other hand, needs to detect a server crash and rebuild the server's state when it comes back up.

This may not sound like an important issue, but it affects the protocol in some strange ways. We feel that it is worth a bit of extra complexity in the protocol to be able to write very simple servers that don't need fancy crash recovery.

# 2. NFS Protocol Definition

The NFS protocol is designed to be operating system independent, but let's face it, it was designed in a UNIX environment. As such, it has some features which are very UNIXish. When in doubt about how something should work, a quick look at how it is done on UNIX will probably put you on the right track.

The protocol definition is given as a set of procedures with arguments and results defined using XDR. A brief description of the function of each procedure should provide enough information to allow implementation on most machines. There is a different section provided for each supported version of the protocol. Most of the procedures, and their parameters and results, are self-explanatory. A few do not fit into the normal UNIX mold, however.

The LOOKUP procedure looks up one component of a pathname at a time. It is not obvious at first why it does not just take the whole pathname, traipse down the directories, and return a file handle when it is done. There are two good reasons not to do this. First, pathnames need separators between the directory components, and different operating systems use different separators. We could define a Network Standard Pathname Representation, but then every pathname would have to be parsed and converted at each end. Second, if pathnames were passed, the server would have to keep track of the mounted filesystems for all of its clients, so that it could break the pathname at the right point and pass the remainder on to the correct server.

Another procedure which might seem strange to UNIX people is the READDIR procedure. What READDIR does is provide a network standard format for representing directories. The same argument as above could have been used to justify a READDIR procedure that returns only one directory entry per call. The problem is efficiency. Directories can contain many entries, and a remote call to return each would just be too slow.

## 2.1. Version 2

The released version of the NFS protocol is actually the second. Even in the second version, there are various obsolete procedures and parameters, which will probably be removed in later versions.

### *2.1.1. Server/Client Relationship*

The NFS protocol is designed to allow servers to be as simple and general as possible. Sometimes the simplicity of the server can be a problem, if the client wants to implement complicated filesystem semantics.

For example, UNIX allows removal of open files. A process can open a file and, while it is open, remove it from the directory. The file can be read and written as long as the process keeps it open, even though the file has no name in the filesystem. It is impossible for a stateless server to implement these semantics. The client can do some tricks like renaming the file on remove, and only removing it on close. We believe that the server provides enough functionality to implement most filesystem semantics on the client.

Every NFS client can also be a server, and remote and local mounted filesystems can be freely intermixed. This leads to some interesting problems when a client travels down the directory tree of a remote filesystem and reaches the mount point on the server for another remote filesystem. Allowing the server to following the second remote mount means it must do loop detection, server lookup, and user revalidation. Instead, we decided not to let clients cross a server's mount point. When a client does a LOOKUP on a directory that the server has mounted a filesystem on, the client sees the underlying directory instead of the mounted directory. A client can do remote mounts that match the server's mount points to maintain the server's view.

### *2.1.2. Permission Issues*

The NFS protocol, strictly speaking, does not define the permission checking used by servers. However, it is expected that a server will do normal UNIX permission checking using AUTH_UNIX style authentication as the basis of its protection mechanism. The server gets the client's effective ***uid***, effective ***gid*** and groups on each call, and uses them to check permission. There are various problems with this method that can been resolved in interesting ways.

Using ***uid*** and ***gid*** implies that the client and server share the same ***uid*** list. Every server and client pair must have the same mapping from user to ***uid*** and from group to ***gid***. Since every client can also be a server this tends to imply that the whole network shares the same ***uid/gid*** space. This is acceptable for the short term, but a more workable network authentication method will be necessary before long.

Another problem arises due to the semantics of open. UNIX does its permission checking at open time and then that the file is open, and has been checked on later read and write requests. With stateless servers this breaks down, because the server has no idea that the file is open and it must do permission checking on each read and write call. On a local filesystem, a user can open a file then change the permissions so that no one is allowed to touch it, but will still be able to write to the file because it is open. On a remote filesystem, by contrast, the write would fail. To get around this problem the server's permission checking algorithm should allow the owner of a file to access it no matter what the permissions are set to.

A similar problem has to do with paging in from a file over the network. The UNIX kernel checks for execute permission before opening a file for demand paging, then reads blocks from the open file. The file may not have read permission but after it is opened it doesn't matter. An NFS server can't tell the difference between a normal file read and a demand page-in read. To make this work the server allows reading of files if the ***uid*** given in the call has execute or read permission on the file.

In UNIX, the user ID zero has access to all files no matter what permission and ownership they have. This super-user permission is not allowed on the server since anyone who can become super-user on their workstation could gain access to all remote files. Instead, the server maps ***uid*** 0 to −2 before doing its access checking. This works as long as the NFS is not used to supply root filesystems, where super-user access cannot be avoided. Eventually servers will have to allow some kind of limited super-user access.

### *2.1.3. RPC Information*

Authentication

The NFS service uses AUTH_UNIX style authentication except in the NULL procedure where AUTH_NONE is also allowed.

Protocols

NFS currently is supported on UDP/IP only.

Constants

These are the RPC constants needed to call the NFS service. They are given in decimal.

| | |
|---|---|
| `PROGRAM` | `100003` |
| `VERSION` | `2` |

Port Number

The NFS protocol currently uses the UDP port number 2049. This is a bug in the protocol and will be changed very shortly.

### *2.1.4. Sizes*

These are the sizes, given in decimal bytes, of various XDR structures used in the protocol.

MAXDATA 8192

The maximum number of bytes of data in a `READ` or `WRITE` request.

MAXPATHLEN 1024

The maximum number of bytes in a pathname argument.

MAXNAMLEN 255

The maximum number of bytes in a file name argument.

COOKIESIZE 4

The size in bytes of the opaque "cookie" passed by `READDIR`.

FHSIZE 32

The size in bytes of the opaque file handle.

### *2.1.5. Basic Data Types*

The following XDR definitions are basic structures and types used in other structures later on.

#### *2.1.5.1. stat*

```
typedef enum {
        NFS_OK = 0,
        NFSERR_PERM=1,
        NFSERR_NOENT=2,
        NFSERR_IO=5,
        NFSERR_NXIO=6,
        NFSERR_ACCES=13,
        NFSERR_EXIST=17,
        NFSERR_NODEV=19,
        NFSERR_NOTDIR=20,
        NFSERR_ISDIR=21,
        NFSERR_FBIG=27,
        NFSERR_NOSPC=28,
        NFSERR_ROFS=30,
        NFSERR_NAMETOOLONG=63,
        NFSERR_NOTEMPTY=66,
        NFSERR_DQUOT=69,
        NFSERR_STALE=70,
        NFSERR_WFLUSH=99
} stat;
```

The `stat` type is returned with every procedure's results. A value of NFS_OK indicates that the call completed successfully and the results are valid. The other values indicate some kind of error occurred on the server side during the servicing of the procedure. The error values are derived from UNIX error numbers.

NFSERR_PERM
: Not owner. The caller does not have correct ownership to perform the requested operation.

NFSERR_NOENT
: No such file or directory. The file or directory specified does not exist.

NFSERR_IO
: I/O error. Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

NFSERR_NXIO
: No such device or address.

NFSERR_ACCES
: Permission denied. The caller does not have the correct permission to perform the requested operation.

NFSERR_EXIST
: File exists. The file specified already exists.

NFSERR_NODEV
: No such device.

NFSERR_NOTDIR
Not a directory. The caller specified a non-directory in a directory operation.

NFSERR_ISDIR
Is a directory. The caller specified a directory in a non-directory operation.

NFSERR_FBIG
File too large. The operation caused a file to grow beyond the server's limit.

NFSERR_NOSPC
No space left on device. The operation caused the server's filesystem to reach its limit.

NFSERR_ROFS
Read-only filesystem. Write attempted on a read-only filesystem.

NFSERR_NAMETOOLONG
File name too long. The file name in an operation was too long.

NFSERR_NOTEMPTY
Directory not empty. Attempted to remove a directory that was not empty.

NFSERR_DQUOT
Disk quota exceeded. The client's disk quota on the server has been exceeded.

NFSERR_STALE
The `fhandle` given in the arguments was invalid. That is, the file referred to by that file handle no longer exists, or access to it has been revoked.

NFSERR_WFLUSH
The server's write cache used in the `WRITECACHE` call got flushed to disk.

#### *2.1.5.2. ftype*

```
typedef enum {
        NFNON = 0,
        NFREG = 1,
        NFDIR = 2,
        NFBLK = 3,
        NFCHR = 4,
        NFLNK = 5
} ftype;
```

The enumeration `ftype` gives the type of a file. The type `NFNON` indicates a non-file, `NFREG` is a regular file, `NFDIR` is a directory, `NFBLK` is a block-special device, `NFCHR` is a character-special device, and `NFLNK` is a symbolic link.

#### *2.1.5.3. fhandle*

```
typedef opaque  fhandle[FHSIZE];
```

The `fhandle` is the file handle that the server passes to the client. All file operations are done using file handles to refer to a file or directory. The file handle can contain whatever information the server needs to distinguish an individual file.

### *2.1.5.4. timeval*

```
typedef struct {
        unsigned seconds;
        unsigned useconds;
} timeval;
```

The timeval structure is the number of seconds and microseconds since midnight January 1, 1970 Greenwich Mean Time. It is used to pass time and date information.

### *2.1.5.5. fattr*

```
typedef struct {
        ftype    type;
        unsigned mode;
        unsigned nlink;
        unsigned uid;
        unsigned gid;
        unsigned size;
        unsigned blocksize;
        unsigned rdev;
        unsigned blocks;
        unsigned fsid;
        unsigned fileid;
        timeval  atime;
        timeval  mtime;
        timeval  ctime;
} fattr;
```

The fattr structure contains the attributes of a file; type is the type of the file; nlink is the number of hard links to the file, that is, the number of different names for the same file; uid is the user identification number of the owner of the file; gid is the group identification number of the group of the file; size is the size in bytes of the file; blocksize is the size in bytes of a block of the file; rdev is the device number of the file if it is type NFCHR or NFBLK; blocks is the number of blocks that the file takes up on disk; fsid is the file system identifier for the filesystem that contains the file; fileid is a number that uniquely identifies the file within its filesystem; atime is the time when the file was last accessed for either read or write; mtime is the time when the file data was last modified (written); and ctime is the time when the status of the file was last changed. Writing to the file also changes ctime if the size of the file changes.

Mode is the access mode encoded as a set of bits. The bits are the same as the mode bits returned by the *stat*(2) system call in UNIX. Notice that the file type is specified both in the mode bits and in the file type. This is really a bug in the protocol and should be fixed in future versions. The descriptions given below specify the bit positions using octal numbers.

0040000 This is a directory. The type field should be NFDIR.

0020000 This is a character special file. The type field should be NFCHR.

0060000 This is a block special file. The type field should be NFBLK.

0100000 This is a regular file. The type field should be NFREG.

0120000 This is a symbolic link file. The type field should be NFLNK.

0140000 This is a named socket. The `type` field should be `NFNON`.

0004000 Set user id on execution.

0002000 Set group id on execution.

0001000 Save swapped text even after use.

0000400 Read permission for owner.

0000200 Write permission for owner.

0000100 Execute and search permission for owner.

0000040 Read permission for group.

0000020 Write permission for group.

0000010 Execute and search permission for group.

0000004 Read permission for others.

0000002 Write permission for others.

0000001 Execute and search permission for others.

### *2.1.5.6. sattr*

```
typedef struct {
    unsigned    mode;
    unsigned    uid;
    unsigned    gid;
    unsigned    size;
    timeval     atime;
    timeval     mtime;
} sattr;
```

The `sattr` structure contains the file attributes which can be set from the client. The fields are the same as for `fattr` above. A `size` of zero means the file should be truncated. A value of −1 indicates a field that should be ignored.

### *2.1.5.7. filename*

```
typedef string  filename<MAXNAMLEN>;
```

The type `filename` is used for passing file names or pathname components.

### *2.1.5.8. path*

```
typedef string  path<MAXPATHLEN>;
```

The type `path` is a pathname. The server considers it as a string with no internal structure, but to the client it is the name of a node in a filesystem tree.

### *2.1.5.9. attrstat*

```
typedef union switch (stat status) {
    NFS_OK:
        fattr   attributes;
    default:
        struct {}
} attrstat;
```

The `attrstat` structure is a common procedure result. It contains a `status` and, if the call succeeded, it also contains the attributes of the file on which the operation was done.

### *2.1.5.10. diropargs*

```
typedef struct {
    fhandle     dir;
    filename    name;
} diropargs;
```

The `diropargs` structure is used in directory operations. The `fhandle dir` is the directory in which to find the file `name`. A directory operation is one in which the directory is affected.

### *2.1.5.11. diropres*

```
typedef union switch (stat status) {
    NFS_OK:
        struct {
                fhandle file;
                fattr   attributes;
        }
    default:
        struct {}
} diropres;
```

The results of a directory operation are returned in a `diropres` structure. If the call succeeded a new file handle `file` and the `attributes` associated with that file are returned along with the `status`.

### *2.1.6. Server Procedures*

The following sections define the RPC procedures supplied by a NFS server. The RPC procedure number and version are given in the header, along with the name of the prodedure. The synopsis of prodecures has this format:

```
<proc #>. <proc name> ( <arguments> ) returns ( <results> )
        <argument declarations>
        <results declarations>
```

In the first line, ***proc name*** is the name of the procedure, ***arguments*** is a list of the names of the arguments, and ***results*** is a list of the names of the results. The second and third lines give the XDR ***argument declarations*** and ***results declarations***. Afterwards, there is a description of what the procedure is expected to do, and how its arguments and results are used. If there are bugs or problems with the procedure, they are listed at the end.

All of the procedures in the NFS protocol are assumed to be synchronous. When a procedure returns to the client, the client can assume that the operation has completed and any data associated with the request is now on stable storage. For example, a client WRITE request may cause the server to update data blocks, filesystem information blocks (such as indirect blocks in UNIX), and file attribute information (size and modify times). When the WRITE returns to the client, it can assume that the write is safe, even in case of a server crash, and it can discard the data written. This is a very important part of the statelessness of the server. If the server waited to flush data from remote requests the client would have to save those requests so that it could resend them in case of a server crash.

#### *2.1.6.1. Do Nothing (Procedure 0, Version 2)*

```
0. NFSPROC_NULL ( ) returns ( )
```

This procedure does no work. It is made available in all RPC services to allow server response testing and timing.

#### *2.1.6.2. Get File Attributes (Procedure 1, Version 2)*

```
1. NFSPROC_GETATTR (file) returns (reply)
        fhandle  file;
        attrstat reply;
```

If `reply.status` is NFS_OK then `reply.attributes` contains the attributes for the file given by `file`.

Bugs: the `rdev` field in the attributes structure is a UNIX device specifier. It should be removed or generalized.

### *2.1.6.3. Set File Attributes (Procedure 2, Version 2)*

```
2. NFSPROC_SETATTR (file, attributes) returns (reply)
        fhandle  file;
        sattr    attributes;
        attrstat reply;
```

The `attributes` argument contains fields which are either −1 or are the new value for the attributes of `file`. If `reply.status` is NFS_OK then `reply.attributes` has the attributes of the file after the `setattr` operation has completed.

Bugs: the use of −1 to indicate an unused field in `attributes` is wrong.

### *2.1.6.4. Get Filesystem Root (Procedure 3, Version 2)*

```
3. NFSPROC_ROOT ( ) returns ( )
```

Obsolete. This procedure is no longer used because finding the root file handle of a filesystem requires moving pathnames between client and server. To do this right we would have to define a network standard representation of pathnames. Instead, the function of looking up the root file handle is done by the MNTPROC_MNT procedure (see section entitled *Mount Protocol Definition* for details).

### *2.1.6.5. Look Up File Name (Procedure 4, Version 2)*

```
4. NFSPROC_LOOKUP (which) returns (reply)
        diropargs which;
        diropres  reply;
```

If `reply.status` is NFS_OK then `reply.file` and `reply.attributes` are the file handle and attributes for the file `which.name` in the directory given by `which.dir`.

Bugs: there is some question as to what is the correct reply to a LOOKUP request when `which.name` is a mount point on the server for a remote mounted filesystem. Currently, we return the `fhandle` of the underlying directory. This is not completely acceptable, as the clients see a different view of the filesystem than the server does.

### *2.1.6.6. Read From Symbolic Link (Procedure 5, Version 2)*

```
5. NFSPROC_READLINK (file) returns (reply)
        fhandle file;
        union switch (stat status) {
            NFS_OK:
                path   data;
            default:
                struct {}
        } reply;
```

If `status` has the value NFS_OK then `reply.data` is the data in the symbolic link given by `file`.

### *2.1.6.7. Read From File (Procedure 6, Version 2)*

```
6. NFSPROC_READ (file, offset, count, totalcount) returns (reply)
        fhandle  file;
        unsigned offset;
        unsigned count;
        unsigned totalcount;
        union switch (stat status) {
            NFS_OK:
                fattr    attributes;
                string   data<MAXDATA>;
            default:
                struct {}
        } reply;
```

Returns up to `count` bytes of `data` from the file given by `file`, starting at `offset` bytes from the beginning of the file. The first byte of the file is at offset zero. The file attributes after the read takes place are returned in `attributes`.

Bugs: the argument `totalcount` is unused, and should be removed.

### *2.1.6.8. Write to Cache (Procedure 7, Version 2)*

```
7. NFSPROC_WRITECACHE ( ) returns ( )
```

Obsolete.

### *2.1.6.9. Write to File (Procedure 8, Version 2)*

```
8. NFSPROC_WRITE (file,beginoffset,offset,totalcount,data) returns (reply)
        fhandle  file;
        unsigned beginoffset;
        unsigned offset;
        unsigned totalcount;
        string   data<MAXDATA>;
        attrstat reply;
```

Writes `data` beginning `offset` bytes from the beginning of `file`. The first byte of the file is at offset zero. If `reply.status` is NFS_OK then `reply.attributes` contains the attributes of the file after the write has completed. The write operation is atomic. Data from this `WRITE` will not be mixed with data from another client's `WRITE`.

Bugs: the arguments `beginoffset` and `totalcount` are ignored and should be removed.

### *2.1.6.10. Create File (Procedure 9, Version 2)*

```
9. NFSPROC_CREATE (where, attributes) returns (dir)
        diropargs where;
        sattr     attributes;
        diropres  dir;
```

The file `where.name` is created in the directory given by `where.dir`. The initial attributes of the new file are given by `attributes`. A `reply.status` of NFS_OK indicates that the file was created and `reply.file` and `reply.attributes` are its file handle and attributes. Any other `reply.status` means that the operation failed and no file was created.

Bugs: this routine should pass an exclusive create flag meaning, create the file only if it is not already there.

### *2.1.6.11. Remove File (Procedure 10, Version 2)*

```
10. NFSPROC_REMOTE (which) returns (status)
        diropargs which;
        stat      status;
```

The file `which.name` is removed from the directory given by `which.dir`. A `status` of NFS_OK means the directory entry was removed.

### *2.1.6.12. Rename File (Procedure 11, Version 2)*

```
11. NFSPROC_RENAME (from, to) returns (status)
        diropargs from;
        diropargs to;
        stat      status;
```

The existing file `from.name` in the directory given by `from.dir` is renamed to `to.name` in the directory given by `to.dir`. If `status` is NFS_OK the file was renamed. The RENAME operation is atomic on the server; it cannot be interrupted in the middle.

### *2.1.6.13. Create Link to File (Procedure 12, Version 2)*

```
12. NFSPROC_LINK (from, to) returns (status)
        fhandle   from;
        diropargs to;
        stat      status;
```

Creates the file `to.name` in the directory given by `to.dir`, which is a hard link to the existing file given by `from`. If the return value of `status` is NFS_OK a link was created. Any other return value indicates an error and the link is not created.

A hard link should have the property that changes to either of the linked files are reflected in both files. When a hard link is made to a file, the attributes for the file should have a value for `nlink` which is one greater than the value before the link.

### *2.1.6.14. Create Symbolic Link (Procedure 13, Version 2)*

```
13. NFSPROC_SYMLINK (from, to, attributes) returns (status)
        diropargs from;
        path      to;
        sattr     attributes;
        stat      status;
```

Creates the file from.name with ftype NFLNK in the directory given by from.dir. The new file contains the pathname to and has initial attributes given by attributes. If the return value of status is NFS_OK a link was created. Any other return value indicates an error and the link is not created.

A symbolic link is a pointer to another file. The name given in to is not interpreted by the server, just stored in the newly created file. A READLINK operation returns the data to the client for interpretation.

Bugs: on UNIX servers the attributes are never used, since symbolic links always have mode 0777.

### *2.1.6.15. Create Directory (Procedure 14, Version 2)*

```
14. NFSPROC_MKDIR (where, attributes) returns (reply)
        diropargs where;
        sattr     attributes;
        diropres  reply;
```

The new directory where.name is created in the directory given by where.dir. The initial attributes of the new directory are given by attributes. A reply.status of NFS_OK indicates that the new directory was created and reply.file and reply.attributes are its file handle and attributes. Any other reply.status means that the operation failed and no directory was created.

### *2.1.6.16. Remove Directory (Procedure 15, Version 2)*

```
15. NFSPROC_RMDIR (which) returns (status)
        diropargs       which;
        stat            status;
```

The existing, empty directory which.name in the directory given by which.dir is removed. If status is NFS_OK the directory was removed.

### *2.1.6.17. Read From Directory (Procedure 16, Version 2)*

```
16. NFSPROC_READDIR (dir, cookie, count) returns (entries)
        fhandle  dir;
        opaque   cookie[COOKIESIZE];
        unsigned count;
        union switch (stat status) {
            NFS_OK:
                typedef union switch (boolean valid) {
                    TRUE:
                        struct {
                            unsigned    fileid;
                            filename    name;
                            opaque      cookie[COOKIESIZE];
                            entry       nextentry;
                        }
                    FALSE:
                        struct {}
                } entry;
                boolean eof;
            default:
        } entries;
```

Returns a variable number of directory entries, with a total size of up to **count** bytes, from the directory given by **dir**. Each **entry** contains a **fileid** which is a unique number to identify the file within a filesystem, the **name** of the file, and a **cookie** which is an opaque pointer to the next entry in the directory. The cookie is used in the next READDIR call to get more entries starting at a given point in the directory. The special cookie zero (all bits zero) can be used to get the entries starting at the beginning of the directory. The **fileid** field should be the same number as the **fileid** in the the attributes of the file (see the section entitled *fattr* under *Basic Data Types*). The **eof** flag has a value of TRUE if there are no more entries in the directory; **valid** is used to mark the end of the entries. If the returned value of **status** is NFS_OK then it is followed by a variable number of **entries**.

### *2.1.6.18. Get Filesystem Attributes (Procedure 17, Version 2)*

```
17. NFSPROC_STATFS (file) returns (reply)
        fhandle  file;
        union switch (stat status) {
            NFS_OK:
                struct {
                    unsigned    tsize;
                    unsigned    bsize;
                    unsigned    blocks;
                    unsigned    bfree;
                    unsigned    bavail;
                } fsattr;
            default:
                struct {}
        } reply;
```

If **reply.status** is NFS_OK then **reply.fsattr** gives the attributes for the filesystem that contains **file**. The attribute fields contain the following values:

tsize The optimum transfer size of the server in bytes. This is the number of bytes the server would like to have in the data part of *READ* and WRITE requests.

bsize The block size in bytes of the filesystem.

blocks The total number of **bsize** blocks on the filesystem.

bfree The number of free **bsize** blocks on the filesystem.

bavail The number of **bsize** blocks available to non-privileged users.

Bugs: this call does not work well if a filesystem has variable size blocks.

# 3. Mount Protocol Definition

The mount protocol is separate from, but related to, the NFS protocol. It provides all of the operating system specific services to get the NFS off the ground — looking up path names, validating user identity, and checking access permissions. Clients use the mount protocol to get the first file handle, which allows them entry into a remote filesystem.

The mount protocol is kept separate from the NFS protocol to make it easy to plug in new access checking and validation methods without changing the NFS server protocol.

Notice that the protocol definition implies stateful servers because the server maintains a list of client's mount requests. The mount list information is not critical for the correct functioning of either the client or the server. It is intented for advisory use only, for example, to warn possible clients when a server is going down.

## 3.1. Version 1

Version one of the mount protocol communicates with the version two of the NFS protocol. The only connecting point is the `fhandle` structure, which is the same for both protocols.

### *3.1.1. RPC Information*

Authentication
The mount service uses **AUTH_UNIX** style authentication only.

Protocols
The mount service is currently supported on UDP/IP only.

Constants
These are the RPC constants needed to call the MOUNT service. They are given in decimal.

```
PROGRAM 100005
VERSION 1
```

Port Number
Consult the server's portmapper, described in the ***RPC Protocol Specification***, to find which port number the mount service is registered on.

### *3.1.2. Sizes*

These are the sizes given in decimal bytes of various XDR structures used in the protocol.

MNTPATHLEN 1024
The maximum number of bytes in a pathname argument.

MNTNAMLEN 255
The maximum number of bytes in a name argument.

FHSIZE 32
The size in bytes of the opaque file handle.

### *3.1.3. Basic Data Types*

#### *3.1.3.1. fhandle*

```
        typedef opaque fhandle[FHSIZE];
```

The `fhandle` is the file handle that the server passes to the client. All file operations are done using file handles to refer to a file or directory. The file handle can contain whatever information the server needs to distinguish an individual file.

This is the same as the `fhandle` XDR definition in version 2 of the NFS protocol; see the section on `fhandle` under ***Basic Data Types***.

#### *3.1.3.2. fhstatus*

```
        typedef union switch (unsigned status) {
            0:
                fhandle directory;
            default:
                struct {}
        }
```

If a `status` of zero is returned, the call completed successfully, and a file handle for the `directory` follows. A non-zero status indicates some sort of error. In this case the status is a UNIX error number.

#### *3.1.3.3. dirpath*

```
        typedef string dirpath<MNTPATHLEN>;
```

The type `dirpath` is a normal UNIX pathname of a directory.

#### *3.1.3.4. name*

```
        typedef string name<MNTNAMLEN>;
```

The type `name` is an arbitrary string used for various names.

### *3.1.4. Server Procedures*

The following sections define the RPC procedures supplied by a mount server. The RPC procedure number and version are given in the header, along with the name of the procedure. The synopsis of procedures has this format:

```
<proc #>. <proc name> ( <arguments> ) returns ( <results> )
        <argument declarations>
        <results declarations>
```

In the first line, ***proc name*** is the name of the procedure, ***arguments*** is a list of the names of the arguments, and ***results*** is a list of the names of the results. The second and third lines give the XDR ***argument declarations*** and ***results declarations***. Afterwards, there is a description of what the procedure is expected to do, and how its arguments and results are used. If there are bugs or problems with the procedure, they are listed at the end.

#### *3.1.4.1. Do Nothing (Procedure 0, Version 1)*

```
0. MNTPROC_NULL ( ) returns ( )
```

This procedure does no work. It is made available in all RPC services to allow server response testing and timing.

#### *3.1.4.2. Add Mount Entry (Procedure 1, Version 1)*

```
1. MNTPROC_MNT (directory) returns (reply)
        dirpath  dirname;
        fhstatus reply;
```

If `reply.status` is 0, `reply.directory` contains the file handle for the directory `dirname`. This file handle may be used in the NFS protocol. This procedure also adds a new entry to the mount list for this client mounting `dirname`.

#### *3.1.4.3. Return Mount Entries (Procedure 2, Version 1)*

```
2. MNTPROC_DUMP ( ) returns (mountlist)
        union switch (boolean more_entries) {
            TRUE:
                struct {
                        name      hostname;
                        dirpath   directory;
                        mountlist nextentry;
                }
            FALSE:
                struct {}
        } mountlist;
```

Returns the list of remote mounted filesystems. The `mountlist` contains one entry for each `hostname` and `directory` pair.

### *3.1.4.4. Remove Mount Entry (Procedure 3, Version 1)*

```
3. MNTPROC_UMNT (directory) returns ( )
        dirpath directory;
```

Removes the mount list entry for `directory`.

### *3.1.4.5. Remove All Mount Entries (Procedure 4, Version 1)*

```
4. MNTPROC_UMNTALL ( ) returns ( )
```

Removes all of the mount list entries for this client.

### *3.1.4.6. Return Export List (Procedure 5, Version 1)*

```
5. MNTPROC_EXPORT ( ) returns (exportlist)
        union switch (boolean more_entries) {
            TRUE:
                struct {
                    dirpath     filesys;
                    typedef union switch (boolean more_groups) {
                        TRUE:
                            struct {
                                name    grname;
                                groups  nextgroup;
                            }
                        FALSE:
                            struct {}
                    } groups;
                    mountlist   nextentry;
                }
            FALSE:
                struct {}
        } exportlist;
```

Returns in `exportlist` a variable number of export list entries. Each entry contains a filesystem name and a list of groups that are allowed to import it. The filesystem name is in `exportlist.filesys`, and the group name is in `exportlist.groups.grname`.

Bugs: the exportlist should contain more information about the status of the filesystem, such as a read-only flag.

# Index

# Yellow Pages
# Protocol Specification

# Contents

# Yellow Pages
# Protocol Specification

## 1. Introduction and Terminology

The Yellow Pages (YP), Sun's distributed lookup service, is a network service providing read access to a replicated database. The lookup service is provided by a set of YP database servers, which communicate among themselves to keep their databases consistent. The client interface to this service uses the Remote Procedure Call (RPC) mechanism.

Translating or mapping a name to its value is one of the most common operations performed in computer systems. Common examples are the translation of a variable name to a virtual memory address, the translation of a user name to a system ID or list of capabilities, and the translation of a network node name to an internet address. There are two fundamental read-only operations that can be performed on a map: matching and enumeration. Match means to look up a name (which we call a **key**) and return its current value. Enumerate means to return each key-value pair in turn.

The YP supplies matching and enumeration operations in a network environment, in which high availability and reliability are required. It provides that availability and reliability by replicating both databases and database servers on multiple nodes within a single local net, and within the internet. The database is replicated, but not distributed: all changes are made at a single server and eventually propagate to the remaining servers without locking. The YP is appropriate for an environment in which changes to the mapping databases occur on the order of tens per day.

The YP operates on an arbitrary number of map databases. Map names provide the lower of two levels of a naming hierarchy. Maps are themselves grouped into named sets, called **domains**. Domain names provides a second, higher level of naming. Map names must be unique within a domain, but may be duplicated in different domains. The YP client interface requires that both a map name and a domain name be supplied to perform match and enumeration operations.

The YP achieves high availability by replication. One area not addressed by the protocol which has to be addressed by the implementors is global consistency among the replicated copies of the database. Every implementation should be designed so that at steady state a request yields the same result when it is made of any YP database server. Update and update-propagation mechanisms must be implemented to supply the required degree of consistency.

### 1.1. RPC — Remote Procedure Call

Sun's Remote Procedure Call (RPC) mechanism defines a paradigm for interprocess communication modeled on function calls. Clients call functions that optionally return values. All inputs and outputs to the functions are in the client's address space. The function is executed by a server program.

Using RPC, clients address servers by a program number (this identifies the application level protocol that the server speaks), and a version number. Additionally, each server procedure has a procedure number assigned to it.

In an internet environment, a client must also know the server's host internet address, and the server's rendezvous port. The server listens for service requests at ports that are associated with a particular transport protocol — TCP/IP and/or UDP/IP.

The format of the data structures used as inputs to and outputs from the remotely-executed procedures are typically defined by header files that are included when the client interface functions are compiled. Levels above the client interface package need not know any particulars of the RPC interface to the server.

## 1.2. XDR — External Data Representation

The Sun External Data Representation (XDR) specification establishes standard representations for basic data types (such as strings, signed and unsigned integers, and structures and unions) in a way that allows them to be transferred among machines with varying architectures. XDR provides primitives to encode (that is, translate from the local host's representation to the standard representation) and decode (translate from the standard representation to the local host's representation) basic data types. Constructor primitives allow arbitrarily complex data types to be made from the basic types.

The YP's RPC input and output data structures are described using XDR's data description language. In general, the data description language looks like the C language, with a few extra constructs. One such extra construct is the ***discriminated union***. This is like a C language union, in that it can hold various objects, but differs from it in that a discriminant indicates which object it currently holds. The discriminant is the first thing across the wire. Consider a simple example:

```
union switch (long int) {
    1:
        string exmpl_name<16>
    0:
        unsigned int exmpl_error_code
    default:
        struct {}
}
```

The example should be interpreted as follows: the first object to be encoded/decoded (that is, the discriminant) is a long integer. If it has the value one, the next object is a string. If the discriminant has the value zero, the next object is an unsigned integer. If the discriminant takes any other value, don't encode or decode any more data.

A ***string*** data type in the XDR data definition language adds the ability to specify the maximum number of elements in an byte array or string of potentially variable size. For instance:

```
string domain<YPMAXDOMAIN>;
```

states that the byte sequence ***domain*** may be less than or equal to YPMAXDOMAIN bytes long.

An additional primitive data type is a ***boolean***, which takes the value one to mean TRUE and zero to mean FALSE.

# 2. YP Data Base Servers

## 2.1. Maps and Operations on Maps

### 2.1.1. *Map Structure*

Maps are named sets of key-value pairs. The keys and their values are counted binary objects. The keys and their values may be ASCII information, but they need not be. The data comprising a map is determined by the client applications that are the final customers for the data, not by the YP. The YP has no syntactic nor semantic knowledge of the map contents. Neither does the YP determine or know any map's name. Map names are managed by the YP's clients. Conflict in the map namespace must be resolved by human administrators outside the YP system.

Typical implementations for YP maps are files or DBMS systems. The design of the YP's map database is an implementation detail, and is unspecified by the protocol.

### 2.1.2. *YP Private Key Symbols*

It is useful to be able to embed key-value pairs that may be used by the YP subsystem itself, or by human administrators or administration programs within all maps. Keys beginning with **YP_** may be conventionally used to embed out-of-band information within a map, and should be considered to be YP-private. The client interface to the YP's enumeration functions should be implemented to filter out YP-private keys. Client programs should not see them; they won't know what to do with them, and client parsers should not be forced to do the filtration.

A unfiltered interface to the YP enumeration functions may also be supplied for programs that need to see YP-private keys. Alternatively, it could be assumed that any client that needs to see a YP-private key knows the name of that key. If that assumption is made, the YP match operation is sufficient, and no unfiltered flavor of the YP enumeration operations needs to be supplied.

The price paid for the ability to imbed administrative information within maps is that the key namespace is reduced.

### 2.1.3. *Match Operation*

The YP supports an exact match operation in the YPPROC_MATCH procedure. That is, if a match string and some key in the map are exactly the same, the value of the key is returned. No pattern matching, case conversion, or wildcarding is supported.

### 2.1.4. *Map Entry Enumeration Operations*

The two operations which exist to enumerate the entries of a map are a "get first key-value pair" operation (the YPPROC_FIRST procedure), and a "get next key-value pair" operation (the YPPROC_NEXT procedure). If "get first" is called once, and then "get next" is called until the return value indicates that there are no more entries in the map, each entry in the map will be seen exactly once. Further, if the same sequence of calls is made again on the same map at the same YP database server, the order in which the entries will be seen is the same.

The actual ordering function is unspecified, and may not be assumed. It also may not be assumed that enumerating a map at a different YP database server will return the entries in the same order, whether that server represents the same implementation or not.

### *2.1.5. Map Update*

The update of YP maps is an implementation detail which is outside the specification of the YP service.

## 2.2. Master and Slave YP Data Base Servers

The protocols assume that for each map there is one distinguished YP database server, called the map's ***master***. Map updates take place only on the master. An updated map should be transferred from the master to the rest of the YP database servers, which are ***slave*** servers for this map.

It is possible for each map to have a different YP database server as its master, or for all maps to have the same master, or any other combination. The choice of how to set up map masters is one of implementation and administrative policy.

## 2.3. Map Propagation, and Consistency

Getting map updates from the master to the slaves is called map propagation. Neither technology nor algorithms for map propagation are specified by the protocol. Map propagation may be entirely manual: for instance, a person could copy the maps from the master to the slaves at a regular interval, or when a change is made on the master. This is unnecessarily labor intensive. There are hooks within the protocol for automatic convergence. The procedures designed for server-to-server communications are described in the next section.

In order to escape from the idiosyncrasies of any particular implementation, all maps should be uniformly timestamped internally. An internal timestamp allows the map to be copied to or reconstructed at any number of nodes, without the time format, local clock time, or file creation or modification algorithms at that site having any effect on the map's version.

The timestamp should be created at the site where the map was created, or was last modified. The timestamp is out-of-band data, as far as the applications using the map are concerned, and should be associated with the YP-private key `YP_LAST_MODIFIED`. Its value should be an ASCII numeric sequence representing the time the map was created or last modified as the number of seconds since January 1, 1970 (GMT). The ASCII numeric sequence may be zero-padded to the left, up to a total length of ten characters. Each YP database server can read the `YP_LAST_MODIFIED` entry from each map it serves, and compare it with the version its peers have.

The intent is for a slave to try to get the current copy from the master. If the master is unreachable, the subnet can still converge at the highest available order number. The slaves communicate among themselves to guarantee that all agree on the current version.

### *2.3.1. Functions to Aid in Map Propagation*

Any YP database server can communicate with any other. Any server may call YPPROC_MATCH, YPPROC_FIRST, or YPPROC_NEXT in a second server, in which case the first server is a client of the second. The protocol also has four functions that exist to help servers converge on a single version of a map.

YPPROC_GET is called by a master server in a peer slave server. It tells the slave server to get a new version of a map from the master.

YPPROC_PUSH is called by an administrative program in a master. It tells the master to notice that a new version of the map exists, and tell the peer slaves to get the new version.

YPPROC_PULL is called by an administrative program in a slave. It tells the slave to get a new version of a map.

YPPROC_POLL can be called either by a server or by an administrative program in any server. It is called to find out what the server's current map version is, and which server it thinks is the map's master.

### *2.3.2. Map Transfer Mechanism*

The way a map is transferred from one server to another is not specified by the protocol. One possibility is the manual process described above. Another might be that a YP database server could activate some other process that would exist only to do the map transfer. A third might be for a server to enumerate the more recent version of the map, by using the normal client map enumeration functions.

If the enumeration method is used, it will take several functions to transfer the whole map, and the map version may change at the supplying site. A version change over the lifetime of the transfer can be detected by the consumer server if the consumer brackets the enumeration with calls to the YPPROC_POLL procedure in the supplier.

## 2.4. Domains

Domain provide a second level for naming within the YP subsystem. They are names for sets of maps, therefore create separate map name spaces. Domains provide an opportunity to break large organizations up into administerable chunks, and the ability to create parallel, non-interfering test and production environments.

Ideally, the domain of interest to a client ought to be associated with the invoking user, but in practice it is useful for client machines to be in a default domain. Implementations of the YP client interface should supply some mechanism for telling processes the domain name they should use. This is needed not only because the concept of domain is a useless one as far as most programs are concerned, but, more importantly, so that programs can be written that are insensitive to both location and the invoking user.

Information logically associated with all domains (or to no domain) can be held in a domain that is really a meta-domain. This domain may have a well-known name, so that information within it can be accessed regardless of the machine's default domain, or of the domain of the invoking user.

## 2.5. Non-features

The following capabilities are not included in the current YP protocols:

### *2.5.1. Map Update Within the YP*

All write (and delete) access to the YP's map database is assumed to be outside of the YP subsystem. It is probable that write access to the map database will be included in later versions of the YP protocols.

### *2.5.2. Version Commitment Across Multiple Requests*

The YP protocol was designed to keep the YP database server stateless with regard to its clients. Therefore, there is no facility for contracting with a server to preallocate any resource beyond that required to service any single request. In particular, there is no way to get a server to commit to use a single version of a map while trying to enumerate that map's entries.

### *2.5.3. Guaranteed Global Consistency*

There is no facility for locking maps during the update or propagation phases, therefore it is virtually guaranteed that the map database be globally inconsistent during those phases. The set of client applications for which the YP is an appropriate lookup service is one that (by definition) must be tolerant of transient inconsistencies.

### *2.5.4. Access Control*

The YP database servers make no attempt to restrict access to the map data by any means. All syntactically correct requests are serviced.

## 2.6. YP Data Base Server Protocol Definition

This section describes version 1 of the protocol. It is likely that changes will be made to successive versions as the service matures.

### *2.6.1. RPC Constants*

All numbers are in decimal.

YPPROG 100004
    The YP database server protocol program number.

YPVERS 1
    The current YP protocol version.

### *2.6.2. Other Manifest Constants*

All numbers are in decimal.

YPMAXRECORD 1024
  The total maximum size of key and value for any pair. The absolute sizes of the key and value may divide this maximum arbitrarily.

YPMAXDOMAIN 64
  The maximum number of characters in a domain name.

YPMAXMAP 64
  The maximum number of characters in a map name.

YPMAXPEER 256
  The maximum number of characters in a YP server host name.

### *2.6.3. Remote Procedure Return Values*

This section presents the return status values returned by several of the YP remote procedures. All numbers are in decimal.

```
typedef enum {
        YP_TRUE = 1,              /* General purpose success code. */
        YP_NOMORE = 2,            /* No more entries in map.  */
        YP_FALSE = 0,             /* General purpose failure code.*/
        YP_NOMAP = -1,            /* No such map in domain.*/
        YP_NODOM = -2,            /* Domain not supported.*/
        YP_NOKEY = -3,            /* No such key in map.*/
        YP_BADOP = -4,            /* Invalid operation.*/
        YP_BADDB = -5,            /* Server database is bad.*/
        YP_YPERR = -6,            /* YP server error.*/
        YP_BADARGS = -7           /* Request arguments bad.*/
} ypstat;
```

### *2.6.4. Basic Data Structures*

This section defines the data structures used as inputs to and outputs from the YP remote procedures.

domainname

```
typedef string domainname<YPMAXDOMAIN>
```

mapname

```
typedef string mapname<YPMAXMAP>
```

peername

```
typedef string peername<YPMAXPEER>
```

keydat

```
typedef string keydat<YPMAXRECORD>
```

valdat

```
typedef string valdat<YPMAXRECORD>
```

ypmap_parms

```
struct ypmap_parms {
        domainname
        mapname
        unsigned long int ordernum
        peername
}
```

This contains parameters giving information about map ***mapname*** within domain ***domainname***. The ***peername*** parameter is the name of the map's master YP database server. If any of the three string pointers represent unknown (or unavailable) information, the parameters will be null strings. The ***ordernum*** parameter contains a binary value representing the value of the map's `YP_LAST_MODIFIED` key. If the `YP_LAST_MODIFIED` value is unavailable, ***ordernum*** contains the value 0.

yprequest

```
struct yprequest {
        union switch (enum ypreqtype) {
                YPREQ_KEY:
                        struct {
                                domainname
                                mapname
                                keydat
                        }
                YPREQ_NOKEY:
                        struct {
                                domainname
                                mapname
                        }
                YPREQ_MAP_PARMS:
                        struct ypmap_parms
                default:
                        {}
        }
}
```

ypresponse

```
struct ypresponse {
        union switch (enum ypresptype) {
                YPRESP_VAL:
                        struct {
                                ypstat
                                valdat
                        }
                YPRESP_KEY_VAL:
                        struct {
                                ypstat
                                valdat
                                keydat
                        }
                YPRESP_MAP_PARMS:
                        struct ypmap_parms
                default:
                        {}
        }
}
```

### *2.6.5. YP Data Base Server Remote Procedures*

This section contains a specification for each function that can be called as a remote procedure. The input and output parameters are described using the XDR data definition language. Whenever the input parameter is a `struct yprequest`, the ***mapname*** and ***domainname*** parameters fully specify the map.

#### *2.6.5.1. Do Nothing (Procedure 0, Version 1)*

```
0. YPPROC_NULL ( ) returns ( )
```

This does no work. It is made available in all RPC services to allow server response testing and timing.

#### *2.6.5.2. Do You Serve This Domain? (Procedure 1, Version 1)*

```
1. YPPROC_DOMAIN (domain) returns (servesp)
        domainname domain;
        boolean servesp;
```

The server returns TRUE if it serves the passed `domain`, and FALSE otherwise. This function allows a potential client to ascertain whether or not a given server supports a named domain.

#### *2.6.5.3. Answer Only If You Serve This Domain (Procedure 2, Version 1)*

```
2. YPPROC_DOMAIN_NONACK (domain) returns (servesp)
        domainname domain;
        boolean servesp;
```

The server returns TRUE if it serves the passed `domain`; otherwise it does not return. The intent of the function is that it be called in a broadcast environment, in which it is useful to restrict the number of useless messages. If this function is called, the client interface implementation must be written so as to regain control in the negative case, for instance by means of a timeout on the response.

Sun's current implementation currently does `return` in the FALSE case by forcing an RPC decode error.

#### *2.6.5.4. Return Value of a Key (Procedure 3, Version 1)*

```
3. YPPROC_MATCH (req) returns (resp)
        struct yprequest  req;
        struct ypresponse resp;
```

The type of the `req` must be YPREQ_KEY. This returns the value associated with the key `keydat`. The type of the `resp` is YPRESP_VAL. If the `ypstat` parameter in the `resp` has the value YP_TRUE, the value data are returned in `valdat`.

### *2.6.5.5. Get First Key-Value Pair in Map (Procedure 4, Version 1)*

```
4. YPPROC_FIRST (req) returns (resp)
        struct yprequest  req;
        struct ypresponse resp;
```

The type of the req must be YPREQ_NOKEY. The *resp* is of type YPRESP_KEY_VAL. If the value of the ypstat is YP_TRUE, this returns the first key-value pair from the map named in the req to the keydat and valdat parameters. An empty map is indicated by ypstat containing the value YP_NOMORE.

### *2.6.5.6. Get Next Key-Value Pair in Map (Procedure 5, Version 1)*

```
5. YPPROC_NEXT (req) returns (resp)
        struct yprequest req;
        struct ypresponse resp;
```

The type of the req must be YPREQ_KEY. The resp is type YPRESP_KEY_VAL. If the value of the ypstat is YP_TRUE, this returns the key-value pair following the key-value named in the req parameter to the keydat and valdat parameters within resp. If the passed key is the last key in the map, the value of ypstat is YP_NOMORE.

### *2.6.5.7. Return Map Parameters (Procedure 6, Version 1)*

```
6. YPPROC_POLL (req) returns (resp)
        struct yprequest  req;
        struct ypresponse resp;
```

The type of the req must be YPREQ_NOKEY. The resp is of type YPREQ_MAP_PARMS. The YP server returns the order number (binary timestamp value) and master server name for the map. If the domain is not supported, the domainname is a null string. If the map is unknown, the mapname is a null string. If unknown, the ordernum parameter has the value zero. If unknown, the peername is a null string.

### *2.6.5.8. Tell Peers About New Map (Procedure 7, Version 1)*

```
7. YPPROC_PUSH (req) returns ( )
        struct yprequest req;
```

The type of the *req* must be YPREQ_NOKEY. The master server rechecks the named map to make sure that the map parameters are up-to-date. It then calls the YPPROC_GET procedure in each reachable peer. If the server is not the master of the named map, it takes no action.

### *2.6.5.9. Get Latest Version of Map (Procedure 8, Version 1)*

```
        8. YPPROC_PULL (req) returns ( )
                struct yprequest req;
```

The type of the `req` must be YPREQ_NOKEY. The slave server attempts to get a more recent version of the named map from a peer. The master, if reachable, is checked first. If the master's version is not greater than the slave's version, the slave does not try any further. If the master's version is greater than the slave's, the slave attempts to transfer the map. If the master is not reachable, the slave attempts to find a greater version held at some other peer. If the server is the master of the named map, it takes no action.

### *2.6.5.10. Get New Map Version From Here (Procedure 9, Version 1)*

```
        9. YPPROC_GET (req) returns ( )
                struct yprequest req;
```

The type of the `req` must be YPREQ_NOKEY. The server assumes that the caller is the master of the map, and tries to get a new version from that master server. In terms of version numbers and peer reachability, it follows the course of action described for YPPROC_PULL. If the server is the master of the named map after replacing the master peer's name with the caller's name, it takes no action. That is, if a master calls YPPROC_GET in itself, it takes no action.

# 3. YP Binders

## 3.1. Introduction

In order that any network service be usable, there must be some way for potential clients to find the servers. This section describes the YP binder, an optional element in the YP subsystem that supplies YP database server addressing information to potential YP clients.

In order to address a YP server in an ARPA internet environment, a client must know the server's internet address, and the port at which the server is listening for service requests. No contract is negotiated between a YP server and a potential client, therefore the addressing information is sufficient to bind the client to the server.

Of the many possible ways for a client to get the addressing information, one alternative is to supply an entity to cache the bindings, and to serve that binding database to potential YP clients. The theory is that if finding the service takes a lot of work, allocate a specialist to do it, rather than burden every client with a job that is irrelevant to its real function. A YP binder only makes sense if it is easier for a client to find the YP binder than to find a YP database server, and if the YP binder can itself find a YP database server.

We make the assumption that a YP binder is present at every network node, and because of this, addressing the YP binder is easier than addressing a YP database server. The scheme for finding a local resource is implementation-specific, but given that a resource is guaranteed to be local, there may be some efficient way of finding it. We further assume that the YP binder can find a YP database server in some way, but that that way is either complicated or time-consuming to do. If either of these assumptions is untrue, then probably your implementation is not a good bet for a YP binder.

If a YP binder is implemented, it can provide added value beyond the binding: it can verify that the binding is correct and that the YP database server is alive and well, for instance. The degree of sureness in a binding that the YP binder gives to a client is a parameter that can be tuned appropriately in the implementation.

## 3.2. YP Binder Protocol Definition

This section describes version 1 of the protocol. It is likely that changes will be made to successive versions as the service matures.

### *3.2.1. RPC Constants*

All numbers are decimal.

YPBINDPROG 100007
The YP binder protocol program number.

YPBINDVERS 1
The current YP binder protocol version.

### *3.2.2. Other Manifest Constants*

All numbers are decimal.

YPMAXDOMAIN 64

The maximum number of characters in a domain name. This is identical to the constant defined above within the YP database server protocol section.

ypbind_resptype

```
enum ypbind_resptype {
        YPBIND_SUCC_VAL = 1,
        YPBIND_FAIL_VAL = 2
}
```

This discriminates between success responses and failure responses to a YPBINDPROC_DOMAIN request.

ypbinderr

```
typedef enum {
        YPBIND_ERR_ERR 1         /* Internal error */
        YPBIND_ERR_NOSERV 2      /* No bound server for passed domain */
        YPBIND_ERR_RESC 3        /* System resource allocation failure */
} ypbinderr
```

The error case of most interest to a YP binder client is YPBIND_ERR_NOSERV; it means that the binding request cannot be satisfied because the YP binder doesn't know how to address any YP database server in the named domain.

### *3.2.3. Basic Data Structures*

This section defines the data structures used as inputs to and outputs from the YP binder remote procedures.

domainname

```
typedef string domainname<YPMAXDOMAIN>
```

This is identical to the domainname string defined above within the YP database server protocol section.

ypbind_binding

```
struct ypbind_binding {
        unsigned long int ypbind_binding_addr
        unsigned short int ypbind_binding_port
}
```

This contains the information necessary to bind a client to a YP database server in the ARPA internet environment. ***ypbind_binding_addr*** holds the host IP address (4 bytes), and ***ypbind_binding_port*** holds the port address (2 bytes). Both IP address and port address must be in ARPA network byte order (most significant byte first, or big endian), regardless of the host machine's native architecture.

ypbind_resp

```
struct ypbind_resp {
        union switch (enum ypbind_resptype status) {
                YPBIND_SUCC_VAL:
                        struct ypbind_binding
                YPBIND_FAIL_VAL:
                        ypbinderr
                default:
                        {}
        }
}
```

This is the response to a YPBINDPROC_DOMAIN request.

ypbind_setdom

```
struct ypbind_setdom {
        domainname
        struct ypbind_binding
}
```

This is the input data structure for the YPBINDPROC_SETDOM procedure.

### *3.2.4. YP Binder Remote Procedures*

Like the YP procedures earlier, these procedures are described using the XDR data definition language.

#### *3.2.4.1. Do Nothing (Procedure 0, Version 1)*

```
        0. YPBINDPROC_NULL ( ) returns ( )
```

This does no work. It is made available in all RPC services to allow server response testing and timing.

#### *3.2.4.2. Get Current Binding for a Domain (Procedure 1, Version 1)*

```
        1. YPBINDPROC_DOMAIN (domain) returns (resp)
                domainname domain;
                struct ypbind_resp resp;
```

This returns the binding information necessary to address a YP database server within the ARPA internet environment.

#### *3.2.4.3. Set Domain Binding (Procedure 2, Version 1)*

```
        2. YPBINDPROC_SETDOM (setdom) returns ( )
                struct ypbind_setdom setdom;
```

This instructs a YP binder to use the passed information as its current binding information for the passed domain.

# Index

# Inter-Process Communication Primer

# Contents

# Inter-Process Communication Primer

This document provides an introduction to the inter-process communication (IPC) facilities on Sun's version of the UNIX† operating system. It discusses the overall model for IPC, and introduces IPC primitives that have been added to the system. The majority of the document considers the use of these primitives in developing applications. The reader is expected to be familiar with the C programming language, as all examples are written in C.

## 1. Introduction

One of the most important features added in the Berkeley 4.2 release of the UNIX operating system is substantial new IPC facilities. These facilities are the result of more than two years of discussion and research. The facilities provided in this release incorporate many of the ideas from current research, while trying to maintain simplicity and conciseness. These IPC facilities have already established a *de facto* standard.

UNIX has previously been weak in doing IPC. Until recently, the only standard mechanism that allowed two processes to communicate were pipes (the `mpx` files in Version 7 were experimental). Unfortunately, pipes are restrictive in that two communicating processes must be related through a common ancestor. Further, the semantics of pipes makes them impossible to maintain in a distributed environment.

Earlier attempts at extending the IPC facilities of UNIX have met with mixed reaction. The majority of problems have been related to these facilities being tied to the UNIX filesystem, either through naming or implementation. Consequently, the IPC facilities provided in this release have been designed as a totally independent subsystem, and allow processes to rendezvous in many ways. Processes may rendezvous through a UNIX filesystem-like name space (a space where all names are path names) as well as through a network name space. In fact, new name spaces may be added at a future time with only minor changes visible to users. Furthermore, the communication facilities have been extended to include more than the simple byte stream provided by pipes. These extensions have resulted in a completely new part of the system, which users will need time to familiarize themselves with. It is likely that as more use is made of these facilities, they will be refined; only time will tell.

The remainder of this document is organized in four sections. Section 2 introduces new system calls and the basic model of communication. Section 3 describes some of the supporting library routines users may find useful in constructing distributed applications. Section 4 is concerned with the client/server model used in developing applications; it includes examples of the two major types of servers. Section 5 delves into advanced topics that sophisticated users may need to know when using IPC facilities.

† UNIX is a trademark of Bell Laboratories.

# 2. Basics

The basic building block for communication is the ***socket***. A socket is an endpoint of communication to which a name may be ***bound***. Each socket in use has a ***type*** and one or more associated processes. Sockets exist within ***communication domains***. A communication domain is an abstraction introduced to bundle common properties of processes communicating through sockets. One such property is the scheme used to name sockets. For example, in the UNIX communication domain sockets are named with UNIX path names; e.g. a socket may be named */dev/foo*. Sockets normally exchange data only with sockets in the same domain (it may be possible to cross domain boundaries, but only if some translation process is performed). The IPC supports two separate communication domains: the UNIX domain, and the Internet domain is used by processes which communicate using the the DARPA standard communication protocols. The underlying communication facilities provided by these domains have a significant influence on the internal system implementation as well as the interface to socket facilities available to a user. An example of the latter is that a socket operating in the UNIX domain sees a subset of the possible error conditions which are possible when operating in the Internet domain.

## 2.1. Socket Types

Sockets are typed according to the communication properties visible to a user. Processes are presumed to communicate only between sockets of the same type, although there is nothing that prevents communication between sockets of different types should the underlying communication protocols support this.

Three types of sockets are currently available to a user. A ***stream*** socket provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. Aside from the bidirectionality of data flow, a pair of connected stream sockets provides an interface nearly identical to that of pipes.[1]

A ***datagram*** socket supports bidirectional flow of data that is not promised to be sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket may find duplicate messages, and possibly in an order different from the order in which it was sent. An important characteristic of a datagram socket is that record boundaries in data are preserved. Datagram sockets closely model the facilities found in many contemporary packet switched networks such as the Ethernet.

A ***raw*** socket provides access to underlying communication protocols that support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics depend on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, who must gain access to the more esoteric facilities of an existing protocol.

Two interesting, but implemented, socket types are the ***sequenced packet*** socket and the ***reliably delivered message*** socket. The first is identical to a stream socket, except that record boundaries are preserved; it is similar to the Xerox NS Sequenced Packet protocol. The second has similar properties to a datagram socket, but with reliable delivery. This document discusses only implemented sockets.

[1] In the UNIX domain, in fact, the semantics are identical and, as one might expect, pipes have been implemented internally as simply a pair of connected stream sockets.

## 2.2. Socket Creation

To create a socket, use the ***socket*** system call:

```
        s = socket(domain, type, protocol);
```

This call requests that the system create a socket in the specified ***domain*** and of the specified ***type***. A particular protocol may also be requested. If the protocol is left unspecified (a value of 0), the system will select an appropriate protocol from those protocols which comprise the communication domain and which may be used to support the requested socket type. The user is returned a descriptor (a small integer number) which may be used in later system calls which operate on sockets. The domain is specified as one of the manifest constants defined in the file <*sys/socket.h*>. For the UNIX domain the constant is AF_UNIX;[2] for the Internet domain AF_INET. The socket types are also defined in this file and one of SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW must be specified. To create a stream socket in the Internet domain the following call might be used:

```
        s = socket(AF_INET, SOCK_STREAM, 0);
```

This call would result in a stream socket being created with the TCP protocol providing the underlying communication support. To create a datagram socket for on-machine use a sample call might be:

```
        s = socket(AF_UNIX, SOCK_DGRAM, 0);
```

To obtain a particular protocol one selects the protocol number, as defined within the communication domain. For the Internet domain the available protocols are defined in <*netinet/in.h*> or, better yet, one may use one of the library routines discussed in section 3, such as ***getprotobyname***:

```
        #include <sys/types.h>
        #include <sys/socket.h>
        #include <netinet/in.h>
        #include <netdb.h>
                 ...
        pp = getprotobyname("tcp");
        s = socket(AF_INET, SOCK_STREAM, pp->p_proto);
```

There are several reasons a socket call may fail. Aside from the rare occurrence of lack of memory (ENOBUFS), a socket request may fail due to a request for an unknown protocol (EPROTONOSUPPORT), or a request for a type of socket for which there is no supporting protocol (EPROTOTYPE).

## 2.3. Binding Names

A socket is created without a name. Until a name is bound to a socket, processes have no way to reference it and, consequently, no messages may be received on it. The ***bind*** call is used to assign a name to a socket:

[2] The manifest constants are named AF_whatever as they indicate the *address format* to use in interpreting names.

```
bind(s, name, namelen);
```

The bound name is a variable length byte string which is interpreted by the supporting protocol(s). Its interpretation may vary from communication domain to communication domain (this is one of the properties which comprise the ***domain***). In the UNIX domain names are path names while in the Internet domain names contain an Internet address and port number. If one wanted to bind the name ***/dev/foo*** to a UNIX domain socket, the following would be used:

```
#include <sys/un.h>
struct sockaddr_un sun;
sun.sun_family = AF_UNIX;
strcpy(sun.sun_path, "/dev/foo");
bind(s, &sun, strlen("/dev/foo")+2);
```

In binding an Internet address things become more complicated. The actual call is simple,

```
#include <sys/types.h>
#include <netinet/in.h>
        ...
struct sockaddr_in sin;
        ...
bind(s, &sin, sizeof (sin));
```

but the selection of what to place in the address ***sin*** requires some discussion. We will come back to the problem of formulating Internet addresses in section 3 when the library routines used in name resolution are discussed.

## 2.4. Connection Establishment

With a bound socket it is possible to rendezvous with an unrelated process. This operation is usually asymmetric with one process a ***client*** and the other a ***server***. The client requests services from the server by initiating a ***connection*** to the server's socket. The server, when willing to offer its advertised services, passively ***listens*** on its socket. On the client side the ***connect*** call is used to initiate a connection. Using the UNIX domain, this might appear as,

```
struct sockaddr_un server;
connect(s, &server, strlen(server.sun_path)+2);
```

while in the Internet domain,

```
struct sockaddr_in server;
connect(s, &server, sizeof (server));
```

If the client process's socket is unbound at the time of the connect call, the system will automatically select and bind a name to the socket; c.f. section 5.4.[3] An error is returned when the connection was unsuccessful (any name automatically bound by the system, however, remains). Otherwise, the socket is associated with the server and data transfer may begin.

Many errors can be returned when a connection attempt fails. The most common are:

ETIMEDOUT

After failing to establish a connection for a period of time, the system decided there was no point in retrying the connection attempt any more. This usually occurs because the

[3] You must do a ***getsockname***(2) call to retrieve the binding.

destination host is down, or because problems in the network resulted in transmissions being lost.

ECONNREFUSED
The host refused service for some reason. When connecting to a host running the 0.9 release version of UNIX this is usually due to a server process not being present at the requested name.

ENETDOWN or EHOSTDOWN
These operational errors are returned based on status information delivered to the client host by the underlying communication services.

ENETUNREACH or EHOSTUNREACH
These operational errors can occur either because the network or host is unknown (no route to the network or host is present), or because of status information returned by intermediate gateways or switching nodes. Many times the status returned is not sufficient to distinguish a network being down from a host being down. In these cases the system is conservative and indicates the entire network is unreachable.

For the server to receive a client's connection it must perform two steps after binding its socket. The first is to indicate a willingness to listen for incoming connection requests:

```
listen(s, 5);
```

The second parameter to the *listen* call specifies the maximum number of outstanding connections which may be queued awaiting acceptance by the server process. Should a connection be requested while the queue is full, the connection will not be refused, but rather the individual messages which comprise the request will be ignored. This gives a harried server time to make room in its pending connection queue while the client retries the connection request. Had the connection been returned with the ECONNREFUSED error, the client would be unable to tell if the server was up or not. As it is now it is still possible to get the ETIMEDOUT error back, though this is unlikely. The backlog figure supplied with the listen call is limited by the system to a maximum of 5 pending connections on any one queue. This avoids the problem of processes hogging system resources by setting an infinite backlog, then ignoring all connection requests.

With a socket marked as listening, a server may *accept* a connection:

```
fromlen = sizeof (from);
snew = accept(s, &from, &fromlen);
```

A new descriptor is returned on receipt of a connection (along with a new socket). If the server wishes to find out who its client is, it may supply a buffer for the client socket's name. The value-result parameter *fromlen* is initialized by the server to indicate how much space is associated with *from*, then modified on return to reflect the true size of the name. If the client's name is not of interest, the second parameter may be zero.

Accept normally blocks. That is, the call to accept will not return until a connection is available or the system call is interrupted by a signal to the process. Further, there is no way for a process to indicate it will accept connections from only a specific individual, or individuals. It is up to the user process to consider who the connection is from and close down the connection if it does not wish to speak to the process. If the server process wants to accept connections on more than one socket, or not block on the accept call there are alternatives; they will be considered in section 5.

## 2.5. Data Transfer

With a connection established, data may begin to flow. To send and receive data there are a number of possible calls. With the peer entity at each end of a connection anchored, a user can send or receive a message without specifying the peer. As one might expect, in this case, then the normal ***read*** and ***write*** system calls are useable,

```
write(s, buf, sizeof (buf));
read(s, buf, sizeof (buf));
```

In addition to ***read*** and ***write***, the new calls ***send*** and ***recv*** may be used:

```
send(s, buf, sizeof (buf), flags);
recv(s, buf, sizeof (buf), flags);
```

While ***send*** and ***recv*** are virtually identical to ***read*** and ***write***, the extra ***flags*** argument is important. The flags may be specified as a non-zero value if one or more of the following is required:

| | |
|---|---|
| MSG_OOB | send/receive out of band data |
| MSG_PEEK | look at data without reading |
| MSG_DONTROUTE | send data without routing packets |

Out of band data is a notion specific to stream sockets, and one which we will not immediately consider. The option to have data sent without routing applied to the outgoing packets is currently used only by the routing table management process, and is unlikely to be of interest to the casual user. The ability to preview data is, however, of interest. When MSG_PREVIEW is specified with a ***recv*** call, any data present is returned to the user, but treated as still unread. That is, the next ***read*** or ***recv*** call to the socket will return data previously previewed.

## 2.6. Discarding Sockets

Once a socket is no longer of interest, it may be discarded by applying a ***close*** to the descriptor,

```
close(s);
```

If data is associated with a socket which promises reliable delivery (e.g. a stream socket) when a close takes place, the system will continue to attempt to transfer the data. However, after a fairly long period of time, if the data is still undelivered, it will be discarded. Should a user have no use for any pending data, it may perform a ***shutdown*** on the socket prior to closing it. This call is of the form:

```
shutdown(s, how);
```

where ***how*** is 0 if the user is no longer interested in reading data, 1 if no more data will be sent, or 2 if no data is to be sent or received. Applying shutdown to a socket causes any data queued to be immediately discarded.

## 2.7. Connectionless Sockets

To this point we have been concerned mostly with sockets which follow a connection oriented model. There is also support for connectionless interactions typical of datagram facilities found in contemporary packet switched networks. A datagram socket provides a symmetric interface to data exchange. While processes are still likely to be client and server, there is no requirement for connection establishment. Instead, each message includes the destination address.

Datagram sockets are created as before, and each should have a name bound to it in order that the recipient of a message may identify the sender. To send data, the *sendto* primitive is used,

```
        sendto(s, buf, buflen, flags, &to, tolen);
```

The *s*, *buf*, *buflen*, and *flags* parameters are used as before. The *to* and *tolen* values are used to indicate the intended recipient of the message. When using an unreliable datagram interface, it is unlikely any errors will be reported to the sender. Where information is present locally to recognize a message which may never be delivered (for instance when a network is unreachable), the call will return −1 and the global value *errno* will contain an error number.

To receive messages on an unconnected datagram socket, the *recvfrom* primitive is provided:

```
        recvfrom(s, buf, buflen, flags, &from, &fromlen);
```

Once again, the *fromlen* parameter is handled in a value-result fashion, initially containing the size of the *from* buffer.

In addition to the two calls mentioned above, datagram sockets may also use the *connect* call to associate a socket with a specific address. In this case, any data sent on the socket will automatically be addressed to the connected peer, and only data received from that peer will be delivered to the user. Only one connected address is permitted for each socket (i.e. no multicasting). Connect requests on datagram sockets return immediately, as this simply results in the system recording the peer's address (as compared to a stream socket where a connect request initiates establishment of an end to end connection). Other of the less important details of datagram sockets are described in section 5.

## 2.8. Input/Output Multiplexing

One last facility often used in developing applications is the ability to multiplex I/O requests among multiple sockets and/or files. This is done using the *select* call:

```
        select(nfds, &readfds, &writefds, &execptfds, &timeout);
```

*Select* takes as arguments three bit masks, one for the set of file descriptors for which the caller wishes to be able to read data on, one for those descriptors to which data is to be written, and one for which exceptional conditions are pending. Bit masks are created by or-ing bits of the form `1<<fd`. That is, a descriptor *fd* is selected if a 1 is present in the *fd*'th bit of the mask. The parameter *nfds* specifies the range of file descriptors (i.e. one plus the value of the largest descriptor) specified in a mask.

A timeout value may be specified if the selection is not to last more than a predetermined period of time. If *timeout* is set to 0, the selection takes the form of a *poll*, returning immediately. If the last parameter is a null pointer, the selection will block indefinitely.[4] *Select* normally returns the number of file descriptors selected. If the *select* call returns due to the timeout expiring, then a value of −1 is returned along with the error number EINTR.

*Select* provides a synchronous multiplexing scheme. Asynchronous notification of output completion, input availability, and exceptional conditions is possible through use of the SIGIO and SIGURG signals described in section 5.

[4] To be more specific, a return takes place only when a descriptor is selectable, or when a signal is received by the caller, interrupting the system call.

# 3. Network Library Routines

The discussion in section 2 indicated the possible need to locate and construct network addresses when using the IPC facilities in a distributed environment. To aid in this task a number of routines have been added to the standard C run-time library. In this section we will consider the new routines provided to manipulate network addresses. While the Sun system release networking facilities support only the DARPA standard Internet protocols, these routines have been designed with flexibility in mind. As more communication protocols become available, we hope the same user interface will be maintained in accessing network-related address data bases. The only difference should be the values returned to the user. Since these values are normally supplied the system, users should not need to be directly aware of the communication protocol and/or naming conventions in use.

Locating a service on a remote host requires many levels of mapping before client and server may communicate. A service is assigned a name which is intended for human consumption; for example, "the *login server* on host monet". This name, and the name of the peer host, must then be translated into network *addresses* which are not necessarily suitable for human consumption. Finally, the address must then used in locating a physical *location* and *route* to the service. The specifics of these three mappings is likely to vary between network architectures. For instance, it is desirable for a network to not require hosts be named in such a way that their physical location is known by the client host. Instead, underlying services in the network may discover the actual location of the host at the time a client host wishes to communicate. This ability to have hosts named in a location independent manner may induce overhead in connection establishment, as a discovery process must take place, but allows a host to be physically mobile without requiring it to notify its clientele of its current location.

Standard routines are provided for: mapping host names to network addresses, network names to network numbers, protocol names to protocol numbers, and service names to port numbers and the appropriate protocol to use in communicating with the server process. The file *<netdb.h>* must be included when using any of these routines.

## 3.1. Host Names

A host name to address mapping is represented by the *hostent* structure:

```
struct  hostent {
        char    *h_name;        /* official name of host */
        char    **h_aliases;    /* alias list */
        int     h_addrtype;     /* host address type */
        int     h_length;       /* length of address */
        char    *h_addr;        /* address */
};
```

Note that the *h_addr* field in the structure definition is defined as a pointer to **char**. In the case of Internet addresses (the only case implemeted to date) you should cast this to a (**struct** in_addr *) when using the item.

The official name of the host and its public aliases are returned, along with a variable length address and address type. The routine *gethostbyname*(3N) takes a host name and returns a *hostent* structure, while the routine *gethostbyaddr*(3N) maps host addresses into a *hostent* structure. It is possible for a host to have many addresses, all having the same name. *Gethostybyname* returns the first matching entry in the data base file */etc/hosts*; if this is unsuitable, the lower level routine *gethostent*(3N) may be used. For example, to obtain a ***hostent*** structure for a host on a particular network the following routine might be used (for simplicity, only Internet addresses are considered):

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
        ...
struct hostent *
gethostbynameandnet(name, net)
        char *name;
        int net;
{
        register struct hostent *hp;
        register char **cp;

        sethostent(0);
        while ((hp = gethostent()) != NULL) {
                if (hp->h_addrtype != AF_INET)
                        continue;
                if (strcmp(name, hp->h_name)) {
                        for (cp = hp->h_aliases; cp && *cp != NULL; cp++)
                                if (strcmp(name, *cp) == 0)
                                goto found;
                        continue;
                }
        found:
                if (in_netof(*(struct in_addr *)hp->h_addr)) == net)
                        break;
        }
        endhostent(0);
        return (hp);
}
```

(*in_netof*(3N) is a standard routine which returns the network portion of an Internet address.)

## 3.2. Network Names

As for host names, routines for mapping network names to numbers, and back, are provided. These routines return a ***netent*** structure:

```
/*
 * Assumption here is that a network number
 * fits in 32 bits -- probably a poor one.
 */
struct  netent {
        char    *n_name;        /* official name of net */
        char    **n_aliases;    /* alias list */
        int     n_addrtype;     /* net address type */
        int     n_net;  /* network # */
};
```

The routines *getnetbyname*(3N), *getnetbynumber*(3N), and *getnetent*(3N) are the network counterparts to the host routines described above.

## 3.3. Protocol Names

For protocols the ***protoent*** structure defines the protocol-name mapping used with the routines ***getprotobyname***(3N), ***getprotobynumber***(3N), and ***getprotoent***(3N):

```
struct  protoent {
        char    *p_name;        /* official protocol name */
        char    **p_aliases;    /* alias list */
        int     p_proto;        /* protocol # */
};
```

## 3.4. Service Names

Information regarding services is a bit more complicated. A service is expected to reside at a specific ***port*** and employ a particular communication protocol. This view is consistent with the Internet domain, but inconsistent with other network architectures. Further, a service may reside on multiple ports or support multiple protocols. If either of these occurs, the higher level library routines will have to be bypassed in favor of homegrown routines similar in spirit to the ***gethostbynameandnet*** routine described above. A service mapping is described by the ***servent*** structure,

```
struct  servent {
        char    *s_name;        /* official service name */
        char    **s_aliases;    /* alias list */
        int     s_port;         /* port # */
        char    *s_proto;       /* protocol to use */
};
```

The routine ***getservbyname***(3N) maps service names to a servent structure by specifying a service name and, optionally, a qualifying protocol. Thus the call

```
sp = getservbyname("telnet", (char *)0);
```

returns the service specification for a telnet server using any protocol, while the call

```
sp = getservbyname("telnet", "tcp");
```

returns only that telnet server which uses the TCP protocol. The routines ***getservbyport***(3N) and ***getservent***(3N) are also provided. The ***getservbyport*** routine has an interface similar to that provided by ***getservbyname***; an optional protocol name may be specified to qualify lookups.

## 3.5. Miscellaneous

With the support routines described above, an application program should rarely have to deal directly with addresses. This allows services to be developed as much as possible in a network independent fashion. It is clear, however, that purging all network dependencies is very difficult. So long as the user is required to supply network addresses when naming services and sockets there will always some network dependency in a program. For example, the normal code included in client programs, such as the remote login program, is of the form shown in Figure 1.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
        ...
main(argc, argv)
        char *argv[];
{
        struct sockaddr_in sin;
        struct servent *sp;
        struct hostent *hp;
        int s;
        ...
        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogin: tcp/login: unknown service\n");
                exit(1);
        }
        hp = gethostbyname(argv[1]);
        if (hp == NULL) {
                fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
                exit(2);
        }
        bzero((char *)&sin, sizeof (sin));
        bcopy(hp->h_addr, (char *)&sin.sin_addr, hp->h_length);
        sin.sin_family = hp->h_addrtype;
        sin.sin_port = sp->s_port;
        s = socket(AF_INET, SOCK_STREAM, 0);
        if (s < 0) {
                perror("rlogin: socket");
                exit(3);
        }
        ...
        if (connect(s, (char *)&sin, sizeof (sin)) < 0) {
                perror("rlogin: connect");
                exit(5);
        }
        ...
}
```

Figure 1: Remote login client code

This example will be considered in more detail in section 4.

If we wanted to make the remote login program independent of the Internet protocols and addressing scheme we would be forced to add a layer of routines which masked the network dependent aspects from the mainstream login code. For the current facilities available in the system this does not appear to be worthwhile. Perhaps when the system is adapted to different network architectures the utilities will be reorganized more cleanly.

Aside from the address-related data base routines, there are several other routines available in the run-time library which are of interest to users. These are intended mostly to simplify manipulation of names and addresses. The following table summarizes the routines for manipulating variable length byte strings and handling byte swapping of network addresses and values.

| C Run-Time Routines | |
|---|---|
| Call | Synopsis |
| `bcmp(s1,s2,n)` | compare byte-strings; 0 if same, not 0 otherwise |
| `bcopy(s1,s2,n)` | copy n bytes from s1 to s2 |
| `bzero(base,n)` | zero-fill n bytes starting at base |
| `htonl(val)` | convert 32-bit quantity from host to network byte order |
| `htons(val)` | convert 16-bit quantity from host to network byte order |
| `ntohl(val)` | convert 32-bit quantity from network to host byte order |
| `ntohs(val)` | convert 16-bit quantity from network to host byte order |

The byte swapping routines are provided because the operating system expects addresses to be supplied in network order. On a VAX, or machine with similar architecture, this is usually reversed. Consequently, programs are sometimes required to byte swap quantities. The library routines which return network addresses provide them in network order so that they may simply be copied into the structures provided to the system. This implies users should encounter the byte swapping problem only when ***interpreting*** network addresses. For example, if an Internet port is to be printed out the following code would be required:

```
    printf("port number %d\n", ntohs(sp->s_port));
```

On machines other than the VAX these routines are defined as null macros.

# 4. Client/Server Model

The most commonly used paradigm in constructing distributed applications is the client/server model. In this scheme client applications request services from a server process. This implies an asymmetry in establishing communication between the client and server which has been examined in section 2. In this section we will look more closely at the interactions between client and server, and consider some of the problems in developing client and server applications.

Client and server require a well known set of conventions before service may be rendered (and accepted). This set of conventions comprises a protocol which must be implemented at both ends of a connection. Depending on the situation, the protocol may be symmetric or asymmetric. In a symmetric protocol, either side may play the master or slave roles. In an asymmetric protocol, one side is immutably recognized as the master, with the other the slave. An example of a symmetric protocol is the TELNET protocol used in the Internet for remote terminal emulation. An example of an asymmetric protocol is the Internet file transfer protocol, FTP. No matter whether the specific protocol used in obtaining a service is symmetric or asymmetric, when accessing a service there is a ***client process*** and a ***server process***. We will first consider the properties of server processes, then client processes.

A server process normally listens at a well know address for service requests. Alternative schemes which use a service server may be used to eliminate a flock of server processes clogging the system while remaining dormant most of the time. The Xerox Courier protocol uses the latter scheme. When using Courier, a Courier client process contacts a Courier server at the remote host and identifies the service it requires. The Courier server process then creates the appropriate server process based on a data base and splices the client and server together, voiding its part in the transaction. This scheme is attractive in that the Courier server process may provide a single contact point for all services, as well as carrying out the initial steps in authentication. However, while this is an attractive possibility for standardizing access to services, it does introduce a certain amount of overhead due to the intermediate process involved. Implementations which provide this type of service within the system can minimize the cost of client server rendezvous.

## 4.1. Servers

In this release, most servers are accessed at well known Internet addresses or UNIX domain names. When a server is started at boot time it advertises it services by listening at a well know location. For example, the remote login server's main loop is of the form shown in Figure 2.

The first step taken by the server is look up its service definition:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogind: tcp/login: unknown service\n");
        exit(1);
}
```

This definition is used in later portions of the code to define the Internet port at which it listens for service requests (indicated by a connection).

Step two is to disassociate the server from the controlling terminal of its invoker. This is important as the server will likely not want to receive signals delivered to the process group of the controlling terminal.

```
main(argc, argv)
        int argc;
        char **argv;
{
        int f;
        struct sockaddr_in from;
        struct servent *sp;

        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogind: tcp/login: unknown service\n");
                exit(1);
        }
        ...
#ifndef DEBUG
        <<disassociate server from controlling terminal>>
#endif
        ...
        sin.sin_port = sp->s_port;
        ...
        f = socket(AF_INET, SOCK_STREAM, 0);
        ...
        if (bind(f, (caddr_t)&sin, sizeof (sin)) < 0) {
                ...
        }
        ...
        listen(f, 5);
        for (;;) {
                int g, len = sizeof (from);

                g = accept(f, &from, &len);
                if (g < 0) {
                        if (errno != EINTR)
                                perror("rlogind: accept");
                        continue;
                }
                if (fork() == 0) {
                        close(f);
                        doit(g, &from);
                }
                close(g);
        }
}
```

Figure 2: Remote login server

Once a server has established a pristine environment, it creates a socket and begins accepting service requests. The *bind* call is required to insure the server listens at its expected location. The main body of the loop is fairly simple:

```
	for (;;) {
		int g, len = sizeof (from);

		g = accept(f, &from, &len);
		if (g < 0) {
			if (errno != EINTR)
				perror("rlogind: accept");
			continue;
		}
		if (fork() == 0) {
			close(f);
			doit(g, &from);
		}
		close(g);
	}
```

An *accept* call blocks the server until a client requests service. This call could return a failure status if the call is interrupted by a signal such as SIGCHLD (to be discussed in section 5). Therefore, the return value from *accept* is checked to insure a connection has actually been established. With a connection in hand, the server then forks a child process and invokes the main body of the remote login protocol processing. Note how the socket used by the parent for queueing connection requests is closed in the child, while the socket created as a result of the accept is closed in the parent. The address of the client is also handed the *doit* routine because it requires it in authenticating clients.

## 4.2. Clients

The client side of the remote login service was shown earlier in Figure 1. One can see the separate, asymmetric roles of the client and server clearly in the code. The server is a passive entity, listening for client connections, while the client process is an active entity, initiating a connection when invoked.

Let us consider more closely the steps taken by the client remote login process. As in the server process the first step is to locate the service definition for a remote login:

```
	sp = getservbyname("login", "tcp");
	if (sp == NULL) {
		fprintf(stderr, "rlogin: tcp/login: unknown service\n");
		exit(1);
	}
```

Next the destination host is looked up with a *gethostbyname* call:

```
	hp = gethostbyname(argv[1]);
	if (hp == NULL) {
		fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
		exit(2);
	}
```

With this accomplished, all that is required is to establish a connection to the server at the requested host and start up the remote login protocol. The address buffer is cleared, then filled in with the Internet address of the foreign host and the port number at which the login process resides:

```
bzero((char *)&sin, sizeof (sin));
bcopy(hp->h_addr, (char *)sin.sin_addr, hp->h_length);
sin.sin_family = hp->h_addrtype;
sin.sin_port = sp->s_port;
```

A socket is created, and a connection initiated.

```
s = socket(hp->h_addrtype, SOCK_STREAM, 0);
if (s < 0) {
        perror("rlogin: socket");
        exit(3);
}
        ...
if (connect(s, (char *)&sin, sizeof (sin)) < 0) {
        perror("rlogin: connect");
        exit(4);
}
```

The details of the remote login protocol will not be considered here.

## 4.3. Connectionless Servers

While connection-based services are the norm, some services are based on the use of datagram sockets. One, in particular, is the `rwho` service which provides users with status information for hosts connected to a local area network. This service, while predicated on the ability to ***broadcast*** information to all hosts connected to a particular network, is of-interest as an example usage of datagram sockets.

A user on any machine running the rwho server may find out the current status of a machine with the *ruptime*(1) program. The output generated is illustrated in Figure 3.

```
arpa     up     9:45,       5 users, load 1.15, 1.39, 1.31
cad      up     2+12:04,    8 users, load 4.67, 5.13, 4.59
calder   up     10:10,      0 users, load 0.27, 0.15, 0.14
dali     up     2+06:28,    9 users, load 1.04, 1.20, 1.65
degas    up     25+09:48,   0 users, load 1.49, 1.43, 1.41
ear      up     5+00:05,    0 users, load 1.51, 1.54, 1.56
ernie    down   0:24
esvax    down   17:04
ingres   down   0:26
kim      up     3+09:16,    8 users, load 2.03, 2.46, 3.11
matisse  up     3+06:18,    0 users, load 0.03, 0.03, 0.05
medea    up     3+09:39,    2 users, load 0.35, 0.37, 0.50
merlin   down   19+15:3
miro     up     1+07:20,    7 users, load 4.59, 3.28, 2.12
monet    up     1+00:43,    2 users, load 0.22, 0.09, 0.07
oz       down   16:09
statvax  up     2+15:57,    3 users, load 1.52, 1.81, 1.86
ucbvax   up     9:34,       2 users, load 6.08, 5.16, 3.28
```

Figure 3: ruptime output

Status information for each host is periodically broadcast by rwho server processes on each machine. The same server process also receives the status information and uses it to update a database. This database is then interpreted to generate the status information for each host. Servers operate autonomously, coupled only by the local network and its broadcast capabilities.

The rwho server, in a simplified form, is pictured in Figure 4. There are two separate tasks performed by the server. The first task is to act as a receiver of status information broadcast by other hosts on the network. This job is carried out in the main loop of the program. Packets received at the rwho port are interrogated to insure they've been sent by another rwho server process, then are time stamped with their arrival time and used to update a file indicating the status of the host. When a host has not been heard from for an extended period of time, the database interpretation routines assume the host is down and indicate such on the status reports. This algorithm is prone to error as a server may be down while a host is actually up, but serves our current needs.

```
main()
{
        ...
        sp = getservbyname("who", "udp");
        net = getnetbyname("localnet");
        sin.sin_addr = inet_makeaddr(INADDR_ANY, net);
        sin.sin_port = sp->s_port;
        ...
        s = socket(AF_INET, SOCK_DGRAM, 0);
        ...
        bind(s, &sin, sizeof (sin));
        ...
        sigset(SIGALRM, onalrm);
        onalrm();
        for (;;) {
                struct whod wd;
                int cc, whod, len = sizeof (from);

                cc = recvfrom(s, (char *)&wd, sizeof (struct whod), 0, &from, &len)
                if (cc <= 0) {
                        if (cc < 0 && errno != EINTR)
                                perror("rwhod: recv");
                        continue;
                }
                if (from.sin_port != sp->s_port) {
                        fprintf(stderr, "rwhod: %d: bad from port\n",
                                ntohs(from.sin_port));
                        continue;
                }
                ...
                if (!verify(wd.wd_hostname)) {
                        fprintf(stderr, "rwhod: malformed host name from %x\n",
                                ntohl(from.sin_addr.s_addr));
                        continue;
                }
                (void) sprintf(path, "%s/whod.%s", RWHODIR, wd.wd_hostname);
                whod = open(path, O_FWRONLY|O_FCREATE|O_FTRUNCATE, 0666);
                ...
                (void) time(&wd.wd_recvtime);
                (void) write(whod, (char *)&wd, cc);
                (void) close(whod);
        }
}
```

Figure 4: rwho server

The second task performed by the server is to supply information regarding the status of its host. This involves periodically acquiring system status information, packaging it up in a message and broadcasting it on the local network for other rwho servers to hear. The supply function is triggered by a timer and runs off a signal. Locating the system status information is somewhat involved, but uninteresting. Deciding where to transmit the resultant packet does, however, indicates some problems with the current protocol.

Status information is broadcast on the local network. For networks which do not support the notion of broadcast another scheme must be used to simulate or replace broadcasting. One possibility is to enumerate the known neighbors (based on the status received). This, unfortunately, requires some bootstrapping information, as a server started up on a quiet network will have no known neighbors and thus never receive, or send, any status information. This is the identical problem faced by the routing table management process in propagating routing status information. The standard solution, unsatisfactory as it may be, is to inform one or more servers of known neighbors and request that they always communicate with these neighbors. If each server has at least one neighbor supplying it, status information may then propagate through a neighbor to hosts which are not (possibly) directly neighbors. If the server is able to support networks which provide a broadcast capability, as well as those which do not, then networks with an arbitrary topology may share status information.[5]

The second problem with the current scheme is that the rwho process services only a single local network, and this network is found by reading a file. It is important that software operating in a distributed environment not have any site-dependent information compiled into it. This would require a separate copy of the server at each host and make maintenance a severe headache. The Sun system attempts to isolate host-specific information from applications by providing system calls which return the necessary information.[6] The rwho server performs a lookup in a file to find its local network. A better, though still unsatisfactory, scheme used by the routing process is to interrogate the system data structures to locate those directly connected networks. A mechanism to acquire this information from the system would be a useful addition.

[5] One must, however, be concerned about loops. That is, if a host is connected to multiple networks, it will receive status information from itself. This can lead to an endless, wasteful, exchange of information.

[6] An example of such a system call is the *gethostname*(2) call which returns the host's official name.

# 5. Advanced Topics

A number of facilities have yet to be discussed. For most users of the IPC the mechanisms already described will suffice in constructing distributed applications. However, others will find need to utilize some of the features which we consider in this section.

## 5.1. Out of Band Data

The stream socket abstraction includes the notion of ***out of band*** data. Out of band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out of band data is delivered to the user independently of normal data along with the SIGURG signal. In addition to the information passed, a logical mark is placed in the data stream to indicate the point at which the out of band data was sent. The remote login and remote shell applications use this facility to propagate signals from between client and server processes. When a signal is expected to flush any pending output from the remote process(es), all data up to the mark in the data stream is discarded.

The stream abstraction defines that the out of band data facilities must support the reliable delivery of at least one out of band message at a time. This message may contain at least one byte of data, and at least one message may be pending delivery to the user at any one time. For communications protocols which support only in-band signaling (that is, the urgent data is delivered in sequence with the normal data) the system extracts the data from the normal data stream and stores it separately. This allows users to choose between receiving the urgent data in order and receiving it out of sequence without having to buffer all the intervening data.

To send an out of band message the MSG_OOB flag is supplied to a ***send*** or ***sendto*** calls, while to receive out of band data MSG_OOB should be indicated when performing a ***recvfrom*** or ***recv*** call. To find out if the read pointer is currently pointing at the mark in the data stream, the SIOCATMARK ioctl is provided:

```
ioctl(s, SIOCATMARK, &yes);
```

If ***yes*** is a 1 on return, the next read will return data after the mark. Otherwise (assuming out of band data has arrived), the next read will provide data sent by the client prior to transmission of the out of band signal. The routine used in the remote login process to flush output on receipt of an interrupt or quit signal is shown in Figure 5.

```
oob()
{
        int out = 1+1;
        char waste[BUFSIZ], mark;

        signal(SIGURG, oob);
        /* flush local terminal input and output */
        ioctl(1, TIOCFLUSH, (char *)&out);
        for (;;) {
                if (ioctl(rem, SIOCATMARK, &mark) < 0) {
                        perror("ioctl");
                        break;
                }
                if (mark)
                        break;
                (void) read(rem, waste, sizeof (waste));
        }
        recv(rem, &mark, 1, MSG_OOB);
        ...
}
```

Figure 5: Flushing terminal I/O on receipt of out of band data

## 5.2. Signals and Process Groups

Due to the existence of the SIGURG and SIGIO signals each socket has an associated process group (just as is done for terminals). This process group is initialized to the process group of its creator, but may be redefined at a later time with the SIOCSPGRP ioctl:

```
ioctl(s, SIOCSPGRP, &pgrp);
```

A similar ioctl, SIOCGPGRP, is available for determining the current process group of a socket.

## 5.3. Pseudo Terminals

Many programs will not function properly without a terminal for standard input and output. Since a socket is not a terminal, it is often necessary to have a process communicating over the network do so through a ***pseudo terminal***. A pseudo terminal is actually a pair of devices, master and slave, which allow a process to serve as an active agent in communication between processes and users. Data written on the slave side of a pseudo terminal is supplied as input to a process reading from the master side. Data written on the master side is given the slave as input. In this way, the process manipulating the master side of the pseudo terminal has control over the information read and written on the slave side. The remote login server uses pseudo terminals for remote login sessions. A user logging in to a machine across the network is provided a shell with a slave pseudo terminal as standard input, output, and error. The server process then handles the communication between the programs invoked by the remote shell and the user's local client process. When a user sends an interrupt or quit signal to a process executing on a remote machine, the client login program traps the signal, sends an out of band message to the server process who then uses the signal number, sent as the data value in the out of band message, to perform a *killpg*(2) on the appropriate process group.

## 5.4. Internet Address Binding

Binding addresses to sockets in the Internet domain can be fairly complex. Communicating processes are bound by an ***association***. An association is composed of local and foreign addresses, and local and foreign ports. Port numbers are allocated out of separate spaces, one for each Internet protocol. Associations are always unique. That is, there may never be duplicate <protocol, local address, local port, foreign address, foreign port> tuples.

The bind system call allows a process to specify half of an association, <local address, local port>, while the connect and accept primitives are used to complete a socket's association. Since the association is created in two steps the association uniqueness requirement indicated above could be violated unless care is taken. Further, it is unrealistic to expect user programs to always know proper values to use for the local address and local port since a host may reside on multiple networks and the set of allocated port numbers is not directly accessible to a user.

To simplify local address binding the notion of a ***wildcard*** address has been provided. When an address is specified as INADDR_ANY (a manifest constant defined in <netinet/in.h>), the system interprets the address as meaning, any valid address. For example, to bind a specific port number to a socket, but leave the local address unspecified, the following code might be used:

```
#include <sys/types.h>
#include <netinet/in.h>
        ...
struct sockaddr_in sin;
        ...
s = socket(AF_INET, SOCK_STREAM, 0);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = MYPORT;
bind(s, (char *)&sin, sizeof (sin));
```

Sockets with wildcarded local addresses may receive messages directed to the specified port number, and addressed to any of the possible addresses assigned a host. For example, if a host is on networks 46 and 10 and a socket is bound as above, then an accept call is performed, the process will be able to accept connection requests which arrive either from network 46 or network 10.

In a similar fashion, a local port may be left unspecified (specified as zero), in which case the system will select an appropriate port number for it. For example:

```
sin.sin_addr.s_addr = MYADDRESS;
sin.sin_port = 0;
bind(s, (char *)&sin, sizeof (sin));
```

The system selects the port number based on two criteria. The first is that ports numbered 0 through IPPORT_RESERVED−1 are reserved for privileged users (that is, the super user). The second is that the port number is not currently bound to some other socket. In order to find a free port number in the privileged range the following code is used by the remote shell server:

```
struct sockaddr_in sin;
        ...
lport = IPPORT_RESERVED - 1;
sin.sin_addr.s_addr = INADDR_ANY;
        ...
for (;;) {
        sin.sin_port = htons((u_short)lport);
        if (bind(s, (caddr_t)&sin, sizeof (sin)) >= 0)
                break;
        if (errno != EADDRINUSE && errno != EADDRNOTAVAIL) {
                perror("socket");
                break;
        }
        lport--;
        if (lport == IPPORT_RESERVED/2) {
                fprintf(stderr, "socket: All ports in use\n");
                break;
        }
}
```

The restriction on allocating ports was done to allow processes executing in a secure environment to perform authentication based on the originating address and port number.

In certain cases the algorithm used by the system in selecting port numbers is unsuitable for an application. This is due to associations being created in a two step process. For example, the Internet file transfer protocol, FTP, specifies that data connections must always originate from the same local port. However, duplicate associations are avoided by connecting to different foreign ports. In this situation the system would disallow binding the same local address and port number to a socket if a previous data connection's socket were around. To override the default port selection algorithm then an option call must be performed prior to address binding:

```
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *)0, 0);
bind(s, (char *)&sin, sizeof (sin));
```

With the above call, local addresses may be bound which are already in use. This does not violate the uniqueness requirement as the system still checks at connect time to be sure any other sockets with the same local address and port do not have the same foreign address and port (if an association already exists, the error EADDRINUSE is returned).

Local address binding by the system is currently done somewhat haphazardly when a host is on multiple networks. Logically, one would expect the system to bind the local address associated with the network through which a peer was communicating. For instance, if the local host is connected to networks 46 and 10 and the foreign host is on network 32, and traffic from network 32 were arriving via network 10, the local address to be bound would be the host's address on network 10, not network 46. This unfortunately, is not always the case. For reasons too complicated to discuss here, the local address bound may be appear to be chosen at random. This property of local address binding will normally be invisible to users unless the foreign host does not understand how to reach the address selected.[7]

[7] For example, if network 46 were unknown to the host on network 32, and the local address were bound to that located on network 46, then even though a route between the two hosts existed through network 10, a connection would fail.

## 5.5. Broadcasting and Datagram Sockets

By using a datagram socket it is possible to send broadcast packets on many networks supported by the system (the network itself must support the notion of broadcasting; the system provides no broadcast simulation in software). Broadcast messages can place a high load on a network since they force every host on the network to service them.

To send a broadcast message, an Internet datagram socket should be created:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

and at least a port number should be bound to the socket:

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = MYPORT;
bind(s, (char *)&sin, sizeof (sin));
```

Then the message should be addressed as:

```
dst.sin_family = AF_INET;
inet_makeaddr(net, INADDR_ANY);
dst.sin_port = DESTPORT;
```

and, finally, a sendto call may be used:

```
sendto(s, buf, buflen, 0, &dst, sizeof (dst));
```

Received broadcast messages contain the senders address and port (datagram sockets are anchored before a message is allowed to go out).

There are a couple of minor problems in the above example. One is created because INADDR_ANY has two meanings:

1. Fill in my own address, and,

2. Broadcast.

Unfortunately, broadcast must at some time in the future be changed to −1 instead of 0, so that broadcast will no longer be The second problem is how do you get your net number? You could use the SIOCGICONF ioctl call, or you could get your own address and do a ***inet_netof*** on that. INADDR_ANY.

## 5.6. Signals

Two new signals have been added to the system which may be used in conjunction with the IPC facilities. The SIGURG signal is associated with the existence of an urgent condition. The SIGIO signal is used with interrupt driven I/O (not presently implemented). SIGURG is currently supplied a process when out of band data is present at a socket. If multiple sockets have out of band data awaiting delivery, a select call may be used to determine those sockets with such data.

An old signal which is useful when constructing server processes is SIGCHLD. This signal is delivered to a process when any children processes have changed state. Normally servers use the signal to ***reap*** child processes after exiting. For example, the remote login server loop shown in Figure 2 may be augmented as follows:

```
int reaper();
        ...
signal(SIGCHLD, reaper);
listen(f, 10);
for (;;) {
        int g, len = sizeof (from);

        g = accept(f, &from, &len, 0);
        if (g < 0) {
                if (errno != EINTR)
                        perror("rlogind: accept");
                continue;
        }
        ...
}
        ...
#include <wait.h>
reaper()
{
        union wait status;

        while (wait3(&status, WNOHANG, 0) > 0)
                ;
}
```

If the parent server process fails to reap its children, a large number of *zombie* processes may be created.

# Network Implementation

# Contents

# Network Implementation

## 1. Introduction

This report describes the internal structure of the networking facilities of the Sun Workstation version of the UNIX† operating system. These facilities are derived from the networking facilities added at U.C. Berkeley in the Berkeley 4.2 release of the system. The system provides a uniform user interface to networking, and a structure that permits system implementors to add new facilities. The internal structure is not visible to the user, rather it is intended to aid implementors of communication protocols and network services by providing a framework that promotes code sharing and minimizes implementation effort.

The reader is expected to be familiar with the C programming language and system interface, as described in the ***System Interface Overview*** at the beginning of the Sun ***System Interface Manual***. Basic understanding of network communication concepts is assumed; where required any additional ideas are introduced.

The remainder of this document provides a description of the system internals, avoiding, when possible, those portions utilized only by the interprocess communication facilities.

## 2. Overview

If we consider the International Standards Organization's (ISO) Open System Interconnection (OSI) model of network communication [ISO81] [Zimmermann80], the networking facilities described here correspond to a portion of the session layer (layer 3) and all of the transport and network layers (layers 2 and 1, respectively).

The network layer provides possibly imperfect data transport services with minimal addressing structure. Addressing at this level is normally host to host, with implicit or explicit routing optionally supported by the communicating agents.

At the transport layer the notions of reliable transfer, data sequencing, flow control, and service addressing are normally included. Reliability is usually managed by explicit acknowledgement of data delivered. Failure to acknowledge a transfer results in retransmission of the data. Sequencing may be handled by tagging each message handed to the network layer by a ***sequence number*** and maintaining state at the endpoints of communication to utilize received sequence numbers in reordering data which arrives out of order.

The session layer facilities may provide forms of addressing which are mapped into formats required by the transport layer, service authentication and client authentication, etc. Various systems also provide services such as data encryption and address and protocol translation.

The following sections begin by describing some of the common data structures and utility routines, then examine the internal layering. The contents of each layer and its interface are considered. Certain of the interfaces are protocol implementation specific. For these cases

† UNIX is a trademark of Bell Laboratories.

examples have been drawn from the Internet [Cerf78] protocol family. Later sections cover routing issues, the design of the raw socket interface and other miscellaneous topics.

## 3. Goals

The networking system was designed with the goal of supporting multiple ***protocol families*** and addressing styles. This required information to be "hidden" in common data structures which could be manipulated by all the pieces of the system, but which required interpretation only by the protocols which "controlled" it. The system described here attempts to minimize the use of shared data structures to those kept by a suite of protocols (a ***protocol family***), and those used for rendezvous between "synchronous" and "asynchronous" portions of the system (for example, queues of data packets are filled at interrupt time and emptied based on user requests).

A major goal of the system was to provide a framework within which new protocols and hardware could easily be supported. To this end, a great deal of effort has been extended to create utility routines which hide many of the more complex and/or hardware dependent chores of networking. Later sections describe the utility routines and the underlying data structures they manipulate.

## 4. Internal Address Representation

Common to all portions of the system are two data structures. These structures are used to represent addresses and various data objects. Addresses, internally are described by the ***sockaddr*** structure,

```
struct sockaddr {
        short   sa_family;      /* data format identifier */
        char    sa_data[14];    /* address */
};
```

All addresses belong to one or more ***address families*** which define their format and interpretation. The ***sa_family*** field indicates which address family the address belongs to, the ***sa_data*** field contains the actual data value. The size of the data field, 14 bytes, was selected based on a study of current address formats

## 5. Memory Management

A single mechanism is used for data storage: memory buffers, or ***mbuf***s. An mbuf is a structure of the form:

```
struct mbuf {
        struct  mbuf *m_next;   /* next buffer in chain */
        u_long  m_off;          /* offset of data */
        short   m_len;          /* amount of data in this mbuf */
        short   m_type;         /* mbuf type (accounting) */
        u_char  m_dat[MLEN];    /* data storage */
        struct  mbuf *m_act;    /* link in higher-level mbuf list */
};
```

The ***m_next*** field is used to chain mbufs together on linked lists, while the ***m_act*** field allows lists of mbufs to be accumulated. By convention, the mbufs common to a single object (for example, a packet) are chained together with the *m_next* field, while groups of objects are linked via the *m_act* field (possibly when in a queue).

Each mbuf has a small data area for storing information, *m_dat.* The *m_len* field indicates the amount of data, while the *m_off* field is an offset to the beginning of the data from the base of the mbuf. Thus, for example, the macro *mtod*, which converts a pointer to an mbuf to a pointer to the data stored in the mbuf, has the form

```
        #define mtod(x,t)          ((t)((int)(x) + (x)->m_off))
```

(note the *t* parameter, a C type cast, is used to cast the resultant pointer for proper assignment).

In addition to storing data directly in the mbuf's data area, data of page size may be also be stored in a separate area of memory. The mbuf utility routines maintain a pool of pages for this purpose and manipulate a private page map for such pages. The virtual addresses of these data pages precede those of mbufs, so when pages of data are separated from an mbuf, the mbuf data offset is a negative value. An array of reference counts on pages is also maintained so that copies of pages may be made without core to core copying (copies are created simply by duplicating the relevant page table entries in the data page map and incrementing the associated reference counts for the pages). Separate data pages are currently used only when copying data from a user process into the kernel, and when bringing data in at the hardware level. Routines which manipulate mbufs are not normally aware if data is stored directly in the mbuf data array, or if it is kept in separate pages.

The following utility routines are available for manipulating mbuf chains:

`m = m_copy(m0, off, len);`

The *m_copy* routine create a copy of all, or part, of a list of the mbufs in *m0. Len* bytes of data, starting *off* bytes from the front of the chain, are copied. Where possible, reference counts on pages are used instead of core to core copies. The original mbuf chain must have at least *off* + *len* bytes of data. If *len* is specified as M_COPYALL, all the data present, offset as before, is copied.

`m_cat(m, n);`

The mbuf chain, *n*, is appended to the end of *m*. Where possible, compaction is performed.

`m_adj(m, diff);`

The mbuf chain, *m* is adjusted in size by *diff* bytes. If *diff* is non-negative, *diff* bytes are shaved off the front of the mbuf chain. If *diff* is negative, the alteration is performed from back to front. No space is reclaimed in this operation, alterations are accomplished by changing the *m_len* and *m_off* fields of mbufs.

`m = m_pullup(m0, size);`

After a successful call to *m_pullup*, the mbuf at the head of the returned list, *m*, is guaranteed to have at least *size* bytes of data in contiguous memory (allowing access via a pointer, obtained using the *mtod* macro). If the original data was less than *size* bytes long, *len* was greater than the size of an mbuf data area (112 bytes), or required resources were unavailable, *m* is 0 and the original mbuf chain is deallocated.

This routine is particularly useful when verifying packet header lengths on reception. For example, if a packet is received and only 8 of the necessary 16 bytes required for a valid packet header are present at the head of the list of mbufs representing the packet, the remaining 8 bytes may be "pulled up" with a single *m_pullup* call. If the call fails the invalid packet will have been discarded.

By insuring mbufs always reside on 128 byte boundaries it is possible to always locate the mbuf associated with a data area by masking off the low bits of the virtual address. This allows modules to store data structures in mbufs and pass them around without concern for locating the original mbuf when it comes time to free the structure. The *dtom* macro is used to convert a pointer into an mbuf's data area to a pointer to the mbuf,

```
#define dtom(x) ((struct mbuf *)((int)x & ~(MSIZE-1)))
```

Mbufs are used for dynamically allocated data structures such as sockets, as well as memory allocated for packets. Statistics are maintained on mbuf usage and can be viewed by users using the *netstat*(8) program.

# 6. Internal Layering

The internal structure of the network system is divided into three layers. These layers correspond to the services provided by the socket abstraction, those provided by the communication protocols, and those provided by the hardware interfaces. The communication protocols are normally layered into two or more individual cooperating layers, though they are collectively viewed in the system as one layer providing services supportive of the appropriate socket abstraction.

The following sections describe the properties of each layer in the system and the interfaces each must conform to.

## 6.1. Socket Layer

The socket layer deals with the interprocess communications facilities provided by the system. A socket is a bidirectional endpoint of communication which is "typed" by the semantics of communication it supports. The system calls described in the *System Interface Overview* are used to manipulate sockets.

A socket consists of the following data structure:

```
struct socket {
        short   so_type;                /* generic type */
        short   so_options;             /* from socket call */
        short   so_linger;              /* time to linger while closing */
        short   so_state;               /* internal state flags */
        caddr_t so_pcb;                 /* protocol control block */
        struct  protosw *so_proto;      /* protocol handle */
        struct  socket *so_head;        /* back pointer to accept socket */
        struct  socket *so_q0;          /* queue of partial connections */
        short   so_q0len;               /* partials on so_q0 */
        struct  socket *so_q;           /* queue of incoming connections */
        short   so_qlen;                /* number of connections on so_q */
        short   so_qlimit;              /* max number queued connections */
        struct  sockbuf so_snd;         /* send queue */
        struct  sockbuf so_rcv;         /* receive queue */
        short   so_timeo;               /* connection timeout */
        u_short so_error;               /* error affecting connection */
        short   so_oobmark;             /* chars to oob mark */
        short   so_pgrp;                /* pgrp for signals */
};
```

Each socket contains two data queues, *so_rcv* and *so_snd*, and a pointer to routines which provide supporting services. The type of the socket, ***so_type*** is defined at socket creation time and used in selecting those services which are appropriate to support it. The supporting protocol is selected at socket creation time and recorded in the socket data structure for later use. Protocols are defined by a table of procedures, the ***protosw*** structure, which will be described in detail later. A pointer to a protocol specific data structure, the "protocol control block" is also present in the socket structure. Protocols control this data structure and it normally includes a back pointer to the parent socket structure(s) to allow easy lookup when returning information to a user (for example, placing an error number in the ***so_error*** field). The other entries in the socket structure are used in queueing connection requests, validating user requests, storing socket characteristics (for example, options supplied at the time a socket is created), and maintaining a socket's state.

Processes "rendezvous at a socket" in many instances. For instance, when a process wishes to extract data from a socket's receive queue and it is empty, or lacks sufficient data to satisfy the request, the process blocks, supplying the address of the receive queue as an "wait channel' to be used in notification. When data arrives for the process and is placed in the socket's queue, the blocked process is identified by the fact it is waiting "on the queue".

### *6.1.1. Socket State*

A socket's state is defined from the following:

```
#define SS_NOFDREF         0x001  /* no file table ref any more */
#define SS_ISCONNECTED     0x002  /* socket connected to a peer */
#define SS_ISCONNECTING    0x004  /* in process of connecting to peer */
#define SS_ISDISCONNECTING 0x008  /* in process of disconnecting */
#define SS_CANTSENDMORE    0x010  /* can't send more data to peer */
#define SS_CANTRCVMORE     0x020  /* can't receive more data from peer */
#define SS_CONNAWAITING    0x040  /* connections awaiting acceptance */
#define SS_RCVATMARK       0x080  /* at mark on input */
#define SS_PRIV            0x100  /* privileged */
#define SS_NBIO            0x200  /* non-blocking ops */
#define SS_ASYNC           0x400  /* async i/o notify */
```

The state of a socket is manipulated both by the protocols and the user (through system calls). When a socket is created the state is defined based on the type of input/output the user wishes to perform. "Non-blocking" I/O implies a process should never be blocked to await resources. Instead, any call which would block returns prematurely with the error EWOULDBLOCK (the service request may be partially fulfilled, for example, a request for more data than is present).

If a process requested "asynchronous" notification of events related to the socket the SIGIO signal is posted to the process. An event is a change in the socket's state, examples of such occurances are: space becoming available in the send queue, new data available in the receive queue, connection establishment or disestablishment, etc.

A socket may be marked "priviledged" if it was created by the super-user. Only priviledged sockets may send broadcast packets, or bind addresses in priviledged portions of an address space.

### *6.1.2. Socket Data Queues*

A socket's data queue contains a pointer to the data stored in the queue and other entries related to the management of the data. The following structure defines a data queue:

```
      struct sockbuf {
              short    sb_cc;           /* actual chars in buffer */
              short    sb_hiwat;        /* max actual char count */
              short    sb_mbcnt;        /* chars of mbufs used */
              short    sb_mbmax;        /* max chars of mbufs to use */
              short    sb_lowat;        /* low water mark */
              short    sb_timeo;        /* timeout */
              struct   mbuf *sb_mb;     /* the mbuf chain */
              struct   proc *sb_sel;    /* process selecting read/write */
              short    sb_flags;        /* flags, see below */
      };
```

Data is stored in a queue as a chain of mbufs. The actual count of characters as well as high and low water marks are used by the protocols in controlling the flow of data. The socket routines cooperate in implementing the flow control policy by blocking a process when it requests to send data and the high water mark has been reached, or when it requests to receive data and less than the low water mark is present (assuming non-blocking I/O has not been specified).

When a socket is created, the supporting protocol "reserves" space for the send and receive queues of the socket. The actual storage associated with a socket queue may fluctuate during a socket's lifetime, but is assumed this reservation will always allow a protocol to acquire enough memory to satisfy the high water marks.

The timeout and select values are manipulated by the socket routines in implementing various portions of the interprocess communications facilities and will not be described here.

A socket queue has a number of flags used in synchronizing access to the data and in acquiring resources;

```
      #define SB_LOCK 0x01     /* lock on data queue (so_rcv only) */
      #define SB_WANT 0x02     /* someone is waiting to lock */
      #define SB_WAIT 0x04     /* someone is waiting for data/space */
      #define SB_SEL  0x08     /* buffer is selected */
      #define SB_COLL 0x10     /* collision selecting */
```

The last two flags are manipulated by the system in implementing the select mechanism.

### *6.1.3. Socket Connection Queueing*

In dealing with connection oriented sockets (for example, SOCK_STREAM) the two sides are considered distinct. One side is termed ***active***, and generates connection requests. The other side is called ***passive*** and accepts connection requests.

From the passive side, a socket is created with the option SO_ACCEPTCONN specified, creating two queues of sockets: ***so_q0*** for connections in progress and ***so_q*** for connections already made and awaiting user acceptance. As a protocol is preparing incoming connections, it creates a socket structure queued on ***so_q0*** by calling the routine ***sonewconn***(). When the connection is established, the socket structure is then transfered to ***so_q***, making it available for an accept.

If an SO_ACCEPTCONN socket is closed with sockets on either *so_q0* or *so_q*, these sockets are dropped.

## 6.2. Protocol Layer(s)

Protocols are described by a set of entry points and certain socket visible characteristics, some of which are used in deciding which socket type(s) they may support.

An entry in the "protocol switch" table exists for each protocol module configured into the system. It has the following form:

```
struct protosw {
        short   pr_type;                    /* socket type used for */
        short   pr_family;                  /* protocol family */
        short   pr_protocol;                /* protocol number */
        short   pr_flags;                   /* socket visible attributes */
    /* protocol-protocol hooks */
        int     (*pr_input)();              /* input to protocol (from below) */
        int     (*pr_output)();             /* output to protocol (from above) */
        int     (*pr_ctlinput)();           /* control input (from below) */
        int     (*pr_ctloutput)();          /* control output (from above) */
    /* user-protocol hook */
        int     (*pr_usrreq)();             /* user request */
    /* utility hooks */
        int     (*pr_init)();               /* initialization routine */
        int     (*pr_fasttimo)();           /* fast timeout (200ms) */
        int     (*pr_slowtimo)();           /* slow timeout (500ms) */
        int     (*pr_drain)();              /* flush any excess space possible */
};
```

A protocol is called through the *pr_init* entry before any other. Thereafter it is called every 200 milliseconds through the *pr_fasttimo* entry and every 500 milliseconds through the *pr_slowtimo* for timer based actions. The system will call the *pr_drain* entry if it is low on space and this should throw away any non-critical data.

Protocols pass data between themselves as chains of mbufs using the *pr_input* and *pr_output* routines. *Pr_input* passes data up (towards the user) and *pr_output* passes it down (towards the network); control information passes up and down on *pr_ctlinput* and *pr_ctloutput*. The protocol is responsible for the space occupied by any the arguments to these entries and must dispose of it.

The *pr_userreq* routine interfaces protocols to the socket code and is described below.

The *pr_flags* field is constructed from the following values:

```
#define PR_ATOMIC       0x01    /* exchange atomic messages only */
#define PR_ADDR         0x02    /* addresses given with messages */
#define PR_CONNREQUIRED 0x04    /* connection required by protocol */
#define PR_WANTRCVD     0x08    /* want PRU_RCVD calls */
#define PR_RIGHTS       0x10    /* passes capabilities */
```

Protocols which are connection-based specify the PR_CONNREQUIRED flag so that the socket routines will never attempt to send data before a connection has been established. If the PR_WANTRCVD flag is set, the socket routines will notfiy the protocol when the user has removed data from the socket's receive queue. This allows the protocol to implement acknowledgement on user receipt, and also update windowing information based on the amount of space available in the receive queue. The PR_ADDR field indicates any data placed in the socket's receive queue will be preceded by the address of the sender. The PR_ATOMIC flag specifies each *user* request to send data must be performed in a single *protocol* send request; it is the protocol's responsibility to maintain record boundaries on data to be sent. The PR_RIGHTS flag indicates the protocol supports the passing of capabilities; this is currently used only the protocols in the UNIX protocol family.

When a socket is created, the socket routines scan the protocol table looking for an appropriate protocol to support the type of socket being created. The *pr_type* field contains one of the possible socket types (for example, SOCK_STREAM), while the *pr_family* field indicates which protocol family the protocol belongs to. The *pr_protocol* field contains the protocol number of the protocol, normally a well known value.

## 6.3. Network-Interface Layer

Each network-interface configured into a system defines a path through which packets may be sent and received. Normally a hardware device is associated with this interface, though there is no requirement for this (for example, all systems have a software "loopback" interface used for debugging and performance analysis). In addition to manipulating the hardware device, an interface module is responsible for encapsulation and deencapsulation of any low level header information required to deliver a message to it's destination. The selection of which interface to use in delivering packets is a routing decision carried out at a higher level than the network-interface layer. Each interface normally identifies itself at boot time to the routing module so that it may be selected for packet delivery.

An interface is defined by the following structure,

```
    struct ifnet {
            char    *if_name;               /* name, for example, ``en'' or ``lo'' */
            short   if_unit;                /* sub-unit for lower level driver */
            short   if_mtu;                 /* maximum transmission unit */
            int     if_net;                 /* network number of interface */
            short   if_flags;               /* up/down, broadcast, etc. */
            short   if_timer;               /* time 'til if_watchdog called */
            int     if_host[2];             /* local net host number */
            struct  sockaddr if_addr;       /* address of interface */
            union {
                    struct  sockaddr ifu_broadaddr;
                    struct  sockaddr ifu_dstaddr;
            } if_ifu;
            struct  ifqueue if_snd;         /* output queue */
            int     (*if_init)();           /* init routine */
            int     (*if_output)();         /* output routine */
            int     (*if_ioctl)();          /* ioctl routine */
            int     (*if_reset)();          /* bus reset routine */
            int     (*if_watchdog)();       /* timer routine */
            int     if_ipackets;            /* packets received on interface */
            int     if_ierrors;             /* input errors on interface */
            int     if_opackets;            /* packets sent on interface */
            int     if_oerrors;             /* output errors on interface */
            int     if_collisions;          /* collisions on csma interfaces */
            struct  ifnet *if_next;
    };
```

Each interface has a send queue and routines used for initialization, *if_init*, and output, *if_output*. If the interface resides on a system bus, the routine *if_reset* will be called after a bus reset has been performed. An interface may also specify a timer routine, *if_watchdog*, which should be called every *if_timer* seconds (if non-zero).

The state of an interface and certain characteristics are stored in the *if_flags* field. The following values are possible:

```
    #define IFF_UP            0x1     /* interface is up */
    #define IFF_BROADCAST     0x2     /* broadcast address valid */
    #define IFF_DEBUG         0x4     /* turn on debugging */
    #define IFF_ROUTE         0x8     /* routing entry installed */
    #define IFF_POINTOPOINT   0x10    /* interface is point-to-point link */
    #define IFF_NOTRAILERS    0x20    /* avoid use of trailers */
    #define IFF_RUNNING       0x40    /* resources allocated */
```

If the interface is connected to a network which supports transmission of *broadcast* packets, the IFF_BROADCAST flag will be set and the *if_broadaddr* field will contain the address to be used in sending or accepting a broadcast packet. If the interface is associated with a point to point hardware link (for example, a DEC DMR-11), the IFF_POINTOPOINT flag will be set and *if_dstaddr* will contain the address of the host on the other side of the connection. These addresses and the local address of the interface, *if_addr*, are used in filtering incoming packets. The interface sets IFF_RUNNING after it has allocated system resources and posted an initial read on the device it manages. This state bit is used to avoid multiple allocation requests when an interface's address is changed. The IFF_NOTRAILERS flag indicates the interface should refrain from using a *trailer* encapsulation on outgoing packets.[1]

[1] *Trailer* protocols are normally disabled on the Sun Workstation.

The information stored in an *ifnet* structure for point to point communication devices is not currently used by the system internally. Rather, it is used by the user level routing process in determining host network connections and in initially devising routes (refer to chapter 10 for more information).

Various statistics are also stored in the interface structure. These may be viewed by users using the *netstat*(1) program.

The interface address and flags may be set with the SIOCSIFADDR and SIOCSIFFLAGS ioctls. SIOCSIFADDR is used to initially define each interface's address; SIOGSIFFLAGS can be used to mark an interface down and perform site-specific configuration.

## 7. Socket/Protocol Interface

The interface between the socket routines and the communication protocols is through the *pr_usrreq* routine defined in the protocol switch table. The following requests to a protocol module are possible:

```
#define PRU_ATTACH      0       /* attach protocol */
#define PRU_DETACH      1       /* detach protocol */
#define PRU_BIND        2       /* bind socket to address */
#define PRU_LISTEN      3       /* listen for connection */
#define PRU_CONNECT     4       /* establish connection to peer */
#define PRU_ACCEPT      5       /* accept connection from peer */
#define PRU_DISCONNECT  6       /* disconnect from peer */
#define PRU_SHUTDOWN    7       /* won't send any more data */
#define PRU_RCVD        8       /* have taken data; more room now */
#define PRU_SEND        9       /* send this data */
#define PRU_ABORT       10      /* abort (fast DISCONNECT, DETATCH) */
#define PRU_CONTROL     11      /* control operations on protocol */
#define PRU_SENSE       12      /* return status into m */
#define PRU_RCVOOB      13      /* retrieve out of band data */
#define PRU_SENDOOB     14      /* send out of band data */
#define PRU_SOCKADDR    15      /* fetch socket's address */
#define PRU_PEERADDR    16      /* fetch peer's address */
#define PRU_CONNECT2    17      /* connect two sockets */
/* begin for protocols internal use */
#define PRU_FASTTIMO    18      /* 200ms timeout */
#define PRU_SLOWTIMO    19      /* 500ms timeout */
#define PRU_PROTORCV    20      /* receive from below */
#define PRU_PROTOSEND   21      /* send to below */
```

A call on the user request routine is of the form,

```
error = (*protosw[].pr_usrreq)(up, req, m, addr, rights);
        int error;
        struct socket *up;
        int req;
        struct mbuf *m, *rights;
        caddr_t addr;
```

The mbuf chain, *m*, and the address are optional parameters. The *rights* parameter is an optional pointer to an mbuf chain containing user specified capabilities (see the *sendmsg* and *recvmsg* system calls). The protocol is responsible for disposal of both mbuf chains. A non-zero return value gives a UNIX error number which should be passed to higher level software. The following paragraphs describe each of the requests possible.

PRU_ATTACH
: When a protocol is bound to a socket (with the ***socket*** system call) the protocol module is called with this request. It is the responsibility of the protocol module to allocate any resources necessary. The "attach" request will always precede any of the other requests, and should not occur more than once.

PRU_DETACH
: This is the antithesis of the attach request, and is used at the time a socket is deleted. The protocol module may deallocate any resources assigned to the socket.

PRU_BIND
: When a socket is initially created it has no address bound to it. This request indicates an address should be bound to an existing socket. The protocol module must verify the requested address is valid and available for use.

PRU_LISTEN
: The "listen" request indicates the user wishes to listen for incoming connection requests on the associated socket. The protocol module should perform any state changes needed to carry out this request (if possible). A "listen" request always precedes a request to accept a connection.

PRU_CONNECT
: The "connect" request indicates the user wants to a establish an association. The ***addr*** parameter supplied describes the peer to be connected to. The effect of a connect request may vary depending on the protocol. Virtual circuit protocols, such as TCP [Postel80b], use this request to initiate establishment of a TCP connection. Datagram protocols, such as UDP [Postel79], simply record the peer's address in a private data structure and use it to tag all outgoing packets. There are no restrictions on how many times a connect request may be used after an attach. If a protocol supports the notion of ***multi-casting***, it is possible to use multiple connects to establish a multi-cast group. Alternatively, an association may be broken by a PRU_DISCONNECT request, and a new association created with a subsequent connect request; all without destroying and creating a new socket.

PRU_ACCEPT
: Following a successful PRU_LISTEN request and the arrival of one or more connections, this request is made to indicate the user has accepted the first connection on the queue of pending connections. The protocol module should fill in the supplied address buffer with the address of the connected party.

PRU_DISCONNECT
: Eliminate an association created with a PRU_CONNECT request.

PRU_SHUTDOWN
: This call is used to indicate no more data will be sent and/or received (the ***addr*** parameter indicates the direction of the shutdown, as encoded in the ***soshutdown*** system call). The protocol may, at its discretion, deallocate any data structures related to the shutdown.

PRU_RCVD
: This request is made only if the protocol entry in the protocol switch table includes the PR_WANTRCVD flag. When a user removes data from the receive queue this request will be sent to the protocol module. It may be used to trigger acknowledgements, refresh windowing information, initiate data transfer, etc.

PRU_SEND

Each user request to send data is translated into one or more PRU_SEND requests (a protocol may indicate a single user send request must be translated into a single PRU_SEND request by specifying the PR_ATOMIC flag in its protocol description). The data to be sent is presented to the protocol as a list of mbufs and an address is, optionally, supplied in the *addr* parameter. The protocol is responsible for preserving the data in the socket's send queue if it is not able to send it immediately, or if it may need it at some later time (for example, for retransmission).

PRU_ABORT

This request indicates an abnormal termination of service. The protocol should delete any existing association(s).

PRU_CONTROL

The "control" request is generated when a user performs a UNIX *ioctl* system call on a socket (and the ioctl is not intercepted by the socket routines). It allows protocol-specific operations to be provided outside the scope of the common socket interface. The *addr* parameter contains a pointer to a static kernel data area where relevant information may be obtained or returned. The *m* parameter contains the actual *ioctl* request code (note the non-standard calling convention).

PRU_SENSE

The "sense" request is generated when the user makes an *fstat* system call on a socket; it requests status of the associated socket. There currently is no common format for the status returned. Information which might be returned includes per-connection statistics, protocol state, resources currently in use by the connection, the optimal transfer size for the connection (based on windowing information and maximum packet size). The *addr* parameter contains a pointer to a static kernel data area where the status buffer should be placed.

PRU_RCVOOB

Any "out-of-band" data presently available is to be returned. An mbuf is passed in to the protocol module and the protocol should either place data in the mbuf or attach new mbufs to the one supplied if there is insufficient space in the single mbuf.

PRU_SENDOOB

Like PRU_SEND, but for out-of-band data.

PRU_SOCKADDR

The local address of the socket is returned, if any is currently bound to the it. The address format (protocol specific) is returned in the *addr* parameter.

PRU_PEERADDR

The address of the peer to which the socket is connected is returned. The socket must be in a SS_ISCONNECTED state for this request to be made to the protocol. The address format (protocol specific) is returned in the *addr* parameter.

PRU_CONNECT2

The protocol module is supplied two sockets and requested to establish a connection between the two without binding any addresses, if possible. This call is used in implementing the *socketpair*(2) system call.

The following requests are used internally by the protocol modules and are never generated by the socket routines. In certain instances, they are handed to the *pr_usrreq* routine solely for convenience in tracing a protocol's operation (for example, PRU_SLOWTIMO).

PRU_FASTTIMO

A “fast timeout” has occured. This request is made when a timeout occurs in the protocol’s ***pr_fastimo*** routine. The ***addr*** parameter indicates which timer expired.

PRU_SLOWTIMO

A “slow timeout” has occured. This request is made when a timeout occurs in the protocol’s ***pr_slowtimo*** routine. The ***addr*** parameter indicates which timer expired.

PRU_PROTORCV

This request is used in the protocol-protocol interface, not by the routines. It requests reception of data destined for the protocol and not the user. No protocols currently use this facility.

PRU_PROTOSEND

This request allows a protocol to send data destined for another protocol module, not a user. The details of how data is marked “addressed to protocol” instead of “addressed to user” are left to the protocol modules. No protocols currently use this facility.

## 8. Protocol/Protocol Interface

The interface between protocol modules is through the ***pr_usrreq***, ***pr_input***, ***pr_output***, ***pr_ctlinput***, and ***pr_ctloutput*** routines. The calling conventions for all but the ***pr_usrreq*** routine are expected to be specific to the protocol modules and are not guaranteed to be consistent across protocol families. We will examine the conventions used for some of the Internet protocols in this section as an example.

### 8.1. pr_output

The Internet protocol UDP uses the convention,

```
error = udp_output(inp, m);
        int error;
        struct inpcb *inp;
        struct mbuf *m;
```

where the ***inp***, “***in***ternet ***p***rotocol ***c***ontrol ***b***lock”, passed between modules conveys per connection state information, and the mbuf chain contains the data to be sent. UDP performs consistency checks, appends its header, calculates a checksum, etc. before passing the packet on to the IP module:

```
error = ip_output(m, opt, ro, allowbroadcast);
        int error;
        struct mbuf *m, *opt;
        struct route *ro;
        int allowbroadcast;
```

The call to IP’s output routine is more complicated than that for UDP, as befits the additional work the IP module must do. The ***m*** parameter is the data to be sent, and the ***opt*** parameter is an optional list of IP options which should be placed in the IP packet header. The ***ro*** parameter is is used in making routing decisions (and passing them back to the caller). The final parameter, ***allowbroadcast*** is a flag indicating if the user is allowed to transmit a broadcast packet. This may be inconsequential if the underlying hardware does not support the notion of broadcasting.

All output routines return 0 on success and a UNIX error number if a failure occured which could be immediately detected (no buffer space available, no route to destination, etc.).

## 8.2. pr_input

Both UDP and TCP use the following calling convention,

```
(void) (*protosw[].pr_input)(m);
        struct mbuf *m;
```

Each mbuf list passed is a single packet to be processed by the protocol module.

The IP input routine is a software interrupt level routine, and so is not called with any parameters. It instead communicates with network interfaces through a queue, *ipintrq*, which is identical in structure to the queues used by the network interfaces for storing packets awaiting transmission.

## 8.3. pr_ctlinput

This routine is used to convey "control" information to a protocol module (i.e. information which might be passed to the user, but is not data). This routine, and the *pr_ctloutput* routine, have not been extensively developed, and thus suffer from a "clumsiness" that can only be improved as more demands are placed on it.

The common calling convention for this routine is,

```
(void) (*protosw[].pr_ctlinput)(req, info);
        int req;
        caddr_t info;
```

The *req* parameter is one of the following,

```
#define PRC_IFDOWN            0  /* interface transition */
#define PRC_ROUTEDEAD         1  /* select new route if possible */
#define PRC_QUENCH            4  /* some said to slow down */
#define PRC_HOSTDEAD          6  /* normally from IMP */
#define PRC_HOSTUNREACH       7  /* ditto */
#define PRC_UNREACH_NET       8  /* no route to network */
#define PRC_UNREACH_HOST      9  /* no route to host */
#define PRC_UNREACH_PROTOCOL 10  /* dst says bad protocol */
#define PRC_UNREACH_PORT     11  /* bad port # */
#define PRC_MSGSIZE          12  /* message size forced drop */
#define PRC_REDIRECT_NET     13  /* net routing redirect */
#define PRC_REDIRECT_HOST    14  /* host routing redirect */
#define PRC_TIMXCEED_INTRANS 17  /* packet lifetime expired in transit */
#define PRC_TIMXCEED_REASS   18  /* lifetime expired on reass q */
#define PRC_PARAMPROB        19  /* header incorrect */
```

while the *info* parameter is a "catchall" value which is request dependent. Many of the requests have obviously been derived from ICMP (the Internet Control Message Protocol), and from error messages defined in the 1822 host/IMP convention [BBN78]. Mapping tables exist to convert control requests to UNIX error codes which are delivered to a user.

### 8.4. pr_ctloutput

This routine is not currently used by any protocol modules.

## 9. Protocol/Network-Interface Interface

The lowest layer in the set of protocols which comprise a protocol family must interface itself to one or more network interfaces in order to transmit and receive packets. It is assumed that any routing decisions have been made before handing a packet to a network interface, in fact this is absolutely necessary in order to locate any interface at all (unless, of course, one uses a single "hardwired" interface). There are two cases to be concerned with, transmission of a packet, and receipt of a packet; each will be considered separately.

### 9.1. Packet Transmission

Assuming a protocol has a handle on an interface, *ifp*, a (struct ifnet *), it transmits a fully formatted packet with the following call,

```
error = (*ifp->if_output)(ifp, m, dst)
        int error;
        struct ifnet *ifp;
        struct mbuf *m;
        struct sockaddr *dst;
```

The output routine for the network interface transmits the packet *m* to the *dst* address, or returns an error indication (a UNIX error number). In reality transmission may not be immediate, or successful; normally the output routine simply queues the packet on its send queue and primes an interrupt driven routine to actually transmit the packet. For unreliable mediums, such as the Ethernet, "successful" transmission simply means the packet has been placed on the cable without a collision. On the other hand, an 1822 interface guarantees proper delivery or an error indication for each message transmitted. The model employed in the networking system attaches no promises of delivery to the packets handed to a network interface, and thus corresponds more closely to the Ethernet. Errors returned by the output routine are normally trivial in nature (no buffer space, address format not handled, etc.).

### 9.2. Packet Reception

Each protocol family must have one or more "lowest level" protocols. These protocols deal with internetwork addressing and are responsible for the delivery of incoming packets to the proper protocol processing modules. In the PUP model [Boggs78] these protocols are termed Level 1 protocols, in the ISO model, network layer protocols. In our system each such protocol module has an input packet queue assigned to it. Incoming packets received by a network interface are queued up for the protocol module and a software interrupt is posted to initiate processing.

Three macros are available for queueing and dequeueing packets,

IF_ENQUEUE(ifq, m)
    This places the packet *m* at the tail of the queue *ifq*.

IF_DEQUEUE(ifq, m)

This places a pointer to the packet at the head of queue *ifq* in *m*. A zero value will be returned in *m* if the queue is empty.

IF_PREPEND(ifq, m)

This places the packet *m* at the head of the queue *ifq*.

Each queue has a maximum length associated with it as a simple form of congestion control. The macro IF_QFULL(ifq) returns 1 if the queue is filled, in which case the macro IF_DROP(ifq) should be used to bump a count of the number of packets dropped and the offending packet dropped. For example, the following code fragment is commonly found in a network interface's input routine,

```
if (IF_QFULL(inq)) {
        IF_DROP(inq);
        m_freem(m);
} else
        IF_ENQUEUE(inq, m);
```

# 10. Gateways and Routing Issues

The system has been designed with the expectation that it will be used in an internetwork environment. The "canonical" environment was envisioned to be a collection of local area networks connected at one or more points through hosts with multiple network interfaces (one on each local area network), and possibly a connection to a long haul network (for example, the ARPANET). In such an environment, issues of gatewaying and packet routing become very important. Certain of these issues, such as congestion control, have been handled in a simplistic manner or specifically not addressed. Instead, where possible, the network system attempts to provide simple mechanisms upon which more involved policies may be implemented. As some of these problems become better understood, the solutions developed will be incorporated into the system.

This section will describe the facilities provided for packet routing. The simplistic mechanisms provided for congestion control are described in chapter 12.

## 10.1. Routing Tables

The network system maintains a set of routing tables for selecting a network interface to use in delivering a packet to its destination. These tables are of the form:

```
struct rtentry {
        u_long  rt_hash;                /* hash key for lookups */
        struct  sockaddr rt_dst;        /* destination net or host */
        struct  sockaddr rt_gateway;    /* forwarding agent */
        short   rt_flags;               /* see below */
        short   rt_refcnt;              /* no. of references to structure */
        u_long  rt_use;                 /* packets sent using route */
        struct  ifnet *rt_ifp;          /* interface to give packet to */
};
```

The routing information is organized in two separate tables, one for routes to a host and one for routes to a network. The distinction between hosts and networks is necessary so that a single mechanism may be used for both broadcast and multi-drop type networks, and also for networks built from point-to-point links (e.g DECnet [DEC80]).

Each table is organized as a hashed set of linked lists. Two 32-bit hash values are calculated by routines defined for each address family; one based on the destination being a host, and one assuming the target is the network portion of the address. Each hash value is used to locate a hash chain to search (by taking the value modulo the hash table size) and the entire 32-bit value is then used as a key in scanning the list of routes. Lookups are applied first to the routing table for hosts, then to the routing table for networks. If both lookups fail, a final lookup is made for a "wildcard" route (by convention, network 0). By doing this, routes to a specific host on a network may be present as well as routes to the network. This also allows a "fall back" network route to be defined to an "smart" gateway which may then perform more intelligent routing.

Each routing table entry contains a destination (who's at the other end of the route), a gateway to send the packet to, and various flags which indicate the route's status and type (host or network). A count of the number of packets sent using the route is kept for use in deciding between multiple routes to the same destination (see below), and a count of "held references" to the dynamically allocated structure is maintained to insure memory reclamation occurs only when the route is not in use. Finally a pointer to the a network interface is kept; packets sent using the route should be handed to this interface.

Routes are typed in two ways: either as host or network, and as "direct" or "indirect". The host/network distinction determines how to compare the ***rt_dst*** field during lookup. If the route is to a network, only a packet's destination network is compared to the ***rt_dst*** entry stored in the table. If the route is to a host, the addresses must match bit for bit.

The distinction between "direct" and "indirect" routes indicates whether the destination is directly connected to the source. This is needed when performing local network encapsulation. If a packet is destined for a peer at a host or network which is not directly connected to the source, the internetwork packet header will indicate the address of the eventual destination, while the local network header will indicate the address of the intervening gateway. Should the destination be directly connected, these addresses are likely to be identical, or a mapping between the two exists. The RTF_GATEWAY flag indicates the route is to an "indirect" gateway agent and the local network header should be filled in from the ***rt_gateway*** field instead of ***rt_dst***, or from the internetwork destination address.

It is assumed multiple routes to the same destination will not be present unless they are deemed ***equal*** in cost (the current routing policy process never installs multiple routes to the same destination). However, should multiple routes to the same destination exist, a request for a route will return the "least used" route based on the total number of packets sent along this route. This can result in a "ping-pong" effect (alternate packets taking alternate routes), unless protocols "hold onto" routes until they no longer find them useful; either because the destination has changed, or because the route is lossy.

Routing redirect control messages are used to dynamically modify existing routing table entries as well as dynamically create new routing table entries. On hosts where exhaustive routing information is too expensive to maintain (for example, work stations), the combination of wildcard routing entries and routing redirect messages can be used to provide a simple routing management scheme without the use of a higher level policy process. Statistics are kept by the routing table routines on the use of routing redirect messages and their affect on the routing tables. These statistics may be viewed using ***netstat***(1).

Status information other than routing redirect control messages may be used in the future, but at present they are ignored. Likewise, more intelligent "metrics" may be used to describe routes in the future, possibly based on bandwidth and monetary costs.

## 10.2. Routing Table Interface

A protocol accesses the routing tables through three routines, one to allocate a route, one to free a route, and one to process a routing redirect control message. The routine *rtalloc* performs route allocation; it is called with a pointer to the following structure,

```
struct route {
        struct  rtentry *ro_rt;
        struct  sockaddr ro_dst;
};
```

The route returned is assumed "held" by the caller until disposed of with an *rtfree* call. Protocols which implement virtual circuits, such as TCP, hold onto routes for the duration of the circuit's lifetime, while connection-less protocols, such as UDP, currently allocate and free routes on each transmission.

The routine *rtredirect* is called to process a routing redirect control message. It is called with a destination address and the new gateway to that destination. If a non-wildcard route exists to the destination, the gateway entry in the route is modified to point at the new gateway supplied. Otherwise, a new routing table entry is inserted reflecting the information supplied. Routes to interfaces and routes to gateways which are not directly accesible from the host are ignored.

## 10.3. User-Level Routing Policies

Routing policies implemented in user processes manipulate the kernel routing tables through two *ioctl* calls. The commands SIOCADDRT and SIOCDELRT add and delete routing entries, respectively; the tables are read through the /dev/kmem device. The decision to place policy decisions in a user process implies routing table updates may lag a bit behind the identification of new routes, or the failure of existing routes, but this period of instability is normally very small with proper implementation of the routing process. Advisory information, such as ICMP error messages and IMP diagnostic messages, may be read from raw sockets (described in the next section).

One routing policy process has already been implemented. The system standard "routing daemon" uses a variant of the Xerox NS Routing Information Protocol [Xerox82] to maintain up to date routing tables in our local environment. Interaction with other existing routing protocols, such as the Internet GGP (Gateway-Gateway Protocol), may be accomplished using a similar process.

# 11. Raw Sockets

A raw socket is a mechanism which allows users direct access to a lower level protocol. Raw sockets are intended for knowledgeable processes which wish to take advantage of some protocol feature not directly accessible through the normal interface, or for the development of new protocols built atop existing lower level protocols. For example, a new version of TCP might be developed at the user level by utilizing a raw IP socket for delivery of packets. The raw IP socket interface attempts to provide an identical interface to the one a protocol would have if it were resident in the kernel.

The raw socket support is built around a generic raw socket interface, and (possibly) augmented by protocol-specific processing routines. This section will describe the core of the raw socket interface.

## 11.1. Control Blocks

Every raw socket has a protocol control block of the following form,

```
struct rawcb {
        struct  rawcb *rcb_next;          /* doubly linked list */
        struct  rawcb *rcb_prev;
        struct  socket *rcb_socket;       /* back pointer to socket */
        struct  sockaddr rcb_faddr;       /* destination address */
        struct  sockaddr rcb_laddr;       /* socket's address */
        caddr_t rcb_pcb;                  /* protocol specific stuff */
        short   rcb_flags;
};
```

All the control blocks are kept on a doubly linked list for performing lookups during packet dispatch. Associations may be recorded in the control block and used by the output routine in preparing packets for transmission. The addresses are also used to filter packets on input; this will be described in more detail shortly. If any protocol specific information is required, it may be attached to the control block using the *rcb_pcb* field.

A raw socket interface is datagram oriented. That is, each send or receive on the socket requires a destination address. This address may be supplied by the user or stored in the control block and automatically installed in the outgoing packet by the output routine. Since it is not possible to determine whether an address is present or not in the control block, two flags, RAW_LADDR and RAW_FADDR, indicate if a local and foreign address are present. Another flag, RAW_DONTROUTE, indicates if routing should be performed on outgoing packets. If it is, a route is expected to be allocated for each "new" destination address. That is, the first time a packet is transmitted a route is determined, and thereafter each time the destination address stored in *rcb_route* differs from *rcb_faddr*, or *rcb_route.ro_rt* is zero, the old route is discarded and a new one allocated.

## 11.2. Input Processing

Input packets are "assigned" to raw sockets based on a simple pattern matching scheme. Each network interface or protocol gives packets to the raw input routine with the call:

```
raw_input(m, proto, src, dst)
        struct mbuf *m;
        struct sockproto *proto, struct sockaddr *src, *dst;
```

The data packet then has a generic header prepended to it of the form

```
struct raw_header {
        struct  sockproto raw_proto;
        struct  sockaddr raw_dst;
        struct  sockaddr raw_src;
};
```

and it is placed in a packet queue for the "raw input protocol" module. Packets taken from this queue are copied into any raw sockets that match the header according to the following rules,

1) The protocol family of the socket and header agree.

2) If the protocol number in the socket is non-zero, then it agrees with that found in the packet header.

3) If a local address is defined for the socket, the address format of the local address is the same as the destination address's and the two addresses agree bit for bit.

4) The rules of 3) are applied to the socket's foreign address and the packet's source address.

A basic assumption is that addresses present in the control block and packet header (as constructed by the network interface and any raw input protocol module) are in a canonical form which may be "block compared".

### 11.3. Output Processing

On output the raw *pr_usrreq* routine passes the packet and raw control block to the raw protocol output routine for any processing required before it is delivered to the appropriate network interface. The output routine is normally the only code required to implement a raw socket interface.

## 12. Buffering and Congestion Control

One of the major factors in the performance of a protocol is the buffering policy used. Lack of a proper buffering policy can force packets to be dropped, cause falsified windowing information to be emitted by protocols, fragment host memory, degrade the overall host performance, etc. Due to problems such as these, most systems allocate a fixed pool of memory to the networking system and impose a policy optimized for "normal" network operation.

The networking system developed for UNIX is little different in this respect. At boot time a fixed amount of memory is allocated by the networking system. At later times more system memory may be requested as the need arises, but at no time is memory ever returned to the system. It is possible to garbage collect memory from the network, but difficult. In order to perform this garbage collection properly, some portion of the network will have to be "turned off" as data structures are updated. The interval over which this occurs must kept small compared to the average inter-packet arrival time, or too much traffic may be lost, impacting other hosts on the network, as well as increasing load on the interconnecting mediums. In our environment we have not experienced a need for such compaction, and thus have left the problem unresolved.

The mbuf structure was introduced in chapter 5. In this section a brief description will be given of the allocation mechanisms, and policies used by the protocols in performing connection level buffering.

### 12.1. Memory Management

The basic memory allocation routines place no restrictions on the amount of space which may be allocated. Any request made is filled until the system memory allocator starts refusing to allocate additional memory. When the current quota of memory is insufficient to satisfy an mbuf allocation request, the allocator requests enough new pages from the system to satisfy the current request only. All memory owned by the network is described by a private page table used in remapping pages to be logically contiguous as the need arises. In addition, an array of reference counts parallels the page table and is used when multiple copies of a page are present.

Mbufs are 128 byte structures, 16 fitting in a 2048 byte page of memory. When data is placed in mbufs, if possible, it is copied or remapped into logically contiguous pages of memory from the network page pool. Data smaller than the size of a page is copied into one or more 112 byte mbuf data areas.

## 12.2. Protocol Buffering Policies

Protocols reserve fixed amounts of buffering for send and receive queues at socket creation time. These amounts define the high and low water marks used by the socket routines in deciding when to block and unblock a process. The reservation of space does not currently result in any action by the memory management routines, though it is clear if one imposed an upper bound on the total amount of physical memory allocated to the network, reserving memory would become important.

Protocols which provide connection level flow control do this based on the amount of space in the associated socket queues. That is, send windows are calculated based on the amount of free space in the socket's receive queue, while receive windows are adjusted based on the amount of data awaiting transmission in the send queue. Care has been taken to avoid the "silly window syndrome" described in [Clark82] at both the sending and receiving ends.

## 12.3. Queue Limiting

Incoming packets from the network are always received unless memory allocation fails. However, each Level 1 protocol input queue has an upper bound on the queue's length, and any packets exceeding that bound are discarded. It is possible for a host to be overwhelmed by excessive network traffic (for instance a host acting as a gateway from a high bandwidth network to a low bandwidth network). As a "defensive" mechanism the queue limits may be adjusted to throttle network traffic load on a host. Consider a host willing to devote some percentage of its machine to handling network traffic. If the cost of handling an incoming packet can be calculated so that an acceptable "packet handling rate" can be determined, then input queue lengths may be dynamically adjusted based on a host's network load and the number of packets awaiting processing. Obviously, discarding packets is not a satisfactory solution to a problem such as this (simply dropping packets is likely to increase the load on a network); the queue lengths were incorporated mainly as a safeguard mechanism.

## 12.4. Packet Forwarding

When packets can not be forwarded because of memory limitations, the system generates a "source quench" message. In addition, any other problems encountered during packet forwarding are also reflected back to the sender in the form of ICMP packets. This helps hosts avoid unneeded retransmissions.

Broadcast packets are never forwarded due to possible dire consequences. In an early stage of network development, broadcast packets were forwarded and a "routing loop" resulted in network saturation and every host on the network crashing.

# 13. Out of Band Data

Out of band data is a facility peculiar to the stream socket abstraction defined. Little agreement appears to exist as to what its semantics should be. TCP defines the notion of "urgent data" as in-line, while the NBS protocols [Burruss81] and numerous others provide a fully independent logical transmission channel along which out of band data is to be sent. In addition, the amount of the data which may be sent as an out of band message varies from protocol to protocol; everything from 1 bit to 16 bytes or more.

A stream socket's notion of out of band data has been defined as the lowest reasonable common denominator (at least reasonable in our minds); clearly this is subject to debate. Out of band data is expected to be transmitted out of the normal sequencing and flow control constraints of the data stream. A minimum of 1 byte of out of band data and one outstanding out of band message are expected to be supported by the protocol supporting a stream socket. It is a protocols prerogative to support larger sized messages, or more than one outstanding out of band message at a time.

Out of band data is maintained by the protocol and usually not stored in the socket's send queue. The PRU_SENDOOB and PRU_RCVOOB requests to the ***pr_usrreq*** routine are used in sending and receiving data.

# Appendix A. Acknowledgements and References

The internal structure of the system is patterned after the Xerox PUP architecture [Boggs79], while in certain places the Internet protocol family has had a great deal of influence in the design. The use of software interrupts for process invocation is based on similar facilities found in the VMS operating system. Many of the ideas related to protocol modularity, memory management, and network interfaces are based on Rob Gurwitz's TCP/IP implementation for the 4.1BSD version of UNIX on the VAX [Gurwitz81].

# Appendix B. References

[Boggs79] Boggs, D. R., J. F. Shoch, E. A. Taft, and R. M. Metcalfe; *PUP: An Internetwork Architecture.* Report CSL-79-10. XEROX Palo Alto Research Center, July 1979.

[BBN78] Bolt Beranek and Newman; *Specification for the Interconnection of Host and IMP.* BBN Technical Report 1822. May 1978.

[Cerf78] Cerf, V. G.; The Catenet Model for Internetworking. Internet Working Group, IEN 48. July 1978.

[Clark82] Clark, D. D.; Window and Acknowledgement Strategy in TCP. Internet Working Group, IEN Draft Clark-2. March 1982.

[DEC80] Digital Equipment Corporation; *DECnet DIGITAL Network Architecture – General Description.* Order No. AA-K179A-TK. October 1980.

[Gurwitz81] Gurwitz, R. F.; VAX-UNIX Networking Support Project – Implementation Description. Internetwork Working Group, IEN 168. January 1981.

[ISO81] International Organization for Standardization. ***ISO Open Systems Interconnection — Basic Reference Model.*** ISO/TC 97/SC 16 N 719. August 1981.

[Joy82a] Joy, W.; Cooper, E.; Fabry, R.; Leffler, S.; and McKusick, M.; ***System Interface Overview.*** Computer Systems Research Group, Technical Report 5. University of California, Berkeley. Draft of September 1, 1982.

[Postel79] Postel, J., ed. ***DOD Standard User Datagram Protocol.*** Internet Working Group, IEN 88. May 1979.

[Postel80a] Postel, J., ed. ***DOD Standard Internet Protocol.*** Internet Working Group, IEN 128. January 1980.

[Postel80b] Postel, J., ed. ***DOD Standard Transmission Control Protocol.*** Internet Working Group, IEN 129. January 1980.

[Xerox81] Xerox Corporation. ***Internet Transport Protocols.*** Xerox System Integration Standard 028112. December 1981.

[Zimmermann80] Zimmermann, H. OSI Reference Model — The ISO Model of Architecture for Open Systems Interconnection. IEEE Transactions on Communications. Com-28(4); 425-432. April 1980.