The product described in this manual and products for use with it are subject to continuous development and improvement. All information of a technical nature and particulars of the product and its use (including the information and particulars in this manual) are given by Acorn Computers Limited in good faith. However, Acorn Computers Limited cannot accept any liability for any loss or damage arising from the use of any information or particulars in this manual.

If you have any comments on this Guide, please complete and return the form at the back of the Guide to the address given there. All other correspondence should be addressed to:

Customer Support and Services
Acorn Computers Limited
Fulbourn Road
Cherry Hinton
Cambridge CB1 4JN

Published July 1989
ISBN X XXXX XXX X
Published by Acorn Computers Technical Publications Department
Part Number 0483,765
Issue 1

Contents

About this manual

Readership of this manual

This manual describes those areas of the RISC iX operating system that are of interest to the programmer writing or porting software for the RISC iX system. The manual contains information not readily available off-the-shelf. It consists of separate chapters, each chapter dealing with a part of the system and existing as a manual in its own right.

The manual is split into two volumes. Volume 1 contains programming information about languages and the kernel. This volume contains programming information about the Network file system (NFS).

Chapter summaries for volume 2

Network services guide – gives an overview of the Network File System and gives examples of how it works.

Remote Procedure Call programming guide – discusses how to write network applications using Remote Procedure Calls.

The rpcgen protocol compiler – describes how to use rpcgen to simplify the writing of RPC applications.

External Data Representation – technical notes on the implementation of the External Data Representation standard.

External Data Representation standard – the specification of the XDR standard for the description and encoding of data.

Remote Procedure Calls protocol specification – the specification of the message protocol used in implementing the Remote Procedure Call (RPC) package.

NFS version 2 protocol specification – describes the Network File System protocol.

Yellow Pages protocol specification – describes the protocol specification for the Yellow pages distributed lookup service.

Inter-Process Communication primer – this provides an introduction to the Inter-Process Communication (IPC) facilities available with the RISC iX operating system.

For general UNIX programming information refer to the Berkeley 4.3 UNIX documentation set. It contains the following manuals:

- *User's Reference Manual* (URM)
- *User's Supplementary Documentation* (USD)
- *Programmer's Reference Manual* (PRM)
- *Programmer's Supplementary Documents*, Volume 1 (PS1)
- *Programmer's Supplementary Documents*, Volume 2 (PS2)
- *System Manager's Manual (SMM)*

These are available from the EUUG, Owles Hall, Buntingford, Herts, SG9 9PL.

Chapter 1 – Network services guide

1 The Network File System

1.1 Introduction

This chapter gives an overview of the Network File System (NFS), which allows users to mount directories across the network, and then to treat remote files as if they were local. Advanced users may want to skip the first few sections, and go straight to examples of how it works. Beginners may not be interested in the next chapter, 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 both customers and vendors to interface NFS with 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.

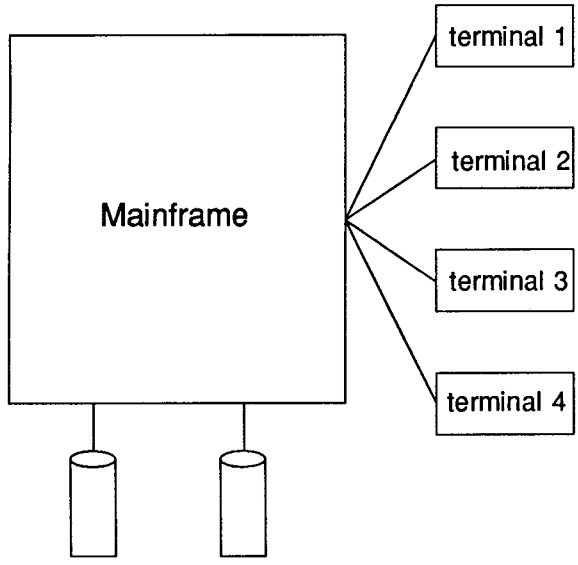
Those who work with unconnected personal computers know the inconveniences resulting from data fragmentation. Even in a network environment, sharing programs and data is sometimes difficult. Files either have to be copied to each machine where they were needed, or users have to log in to the remote machine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

To solve this problem, Sun designed a distributed filesystem that permits client systems to access 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.

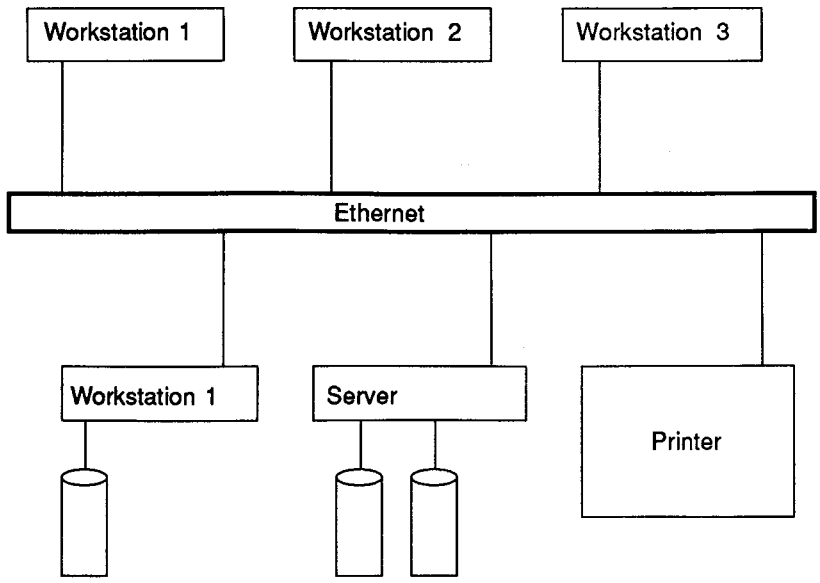
The NFS was not designed by extending the UNIX operating system onto the network. Instead, the NFS was designed to fit into Sun's network services architecture. Thus, NFS is not a distributed operating system, but rather, an interface to allow a variety of machines and operating systems to play the role 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.

1.2 Computing environments

The traditional computing environment looks like this:



The major problem with this environment is competition for CPU cycles. The workstation environment solves that problem, but requires more disc drives. A network environment looks like this:



Sun's goal with NFS was to make all discs 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.

1.3 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 generalised 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 standardised 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 centralised 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.

1.4 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 disc 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 disc, 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 `/usr/machine`, 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`.

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 is that `rcp` 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.

NFS and ND

Sun's Network disc (ND) is a device driver that makes a raw disc available using a simple protocol. The ND client builds its own filesystem, given the disc. disc space on the server machine is partitioned, and discless 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 disc 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 disc 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 disc and not a filesystem, there can be no cacheing on the server side. The NFS permits cacheing, with concomitant performance improvements.

1.5 Examples of how it works

This section gives three examples of how to use the NFS.

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`. Mount the directory containing the manuals as follows:

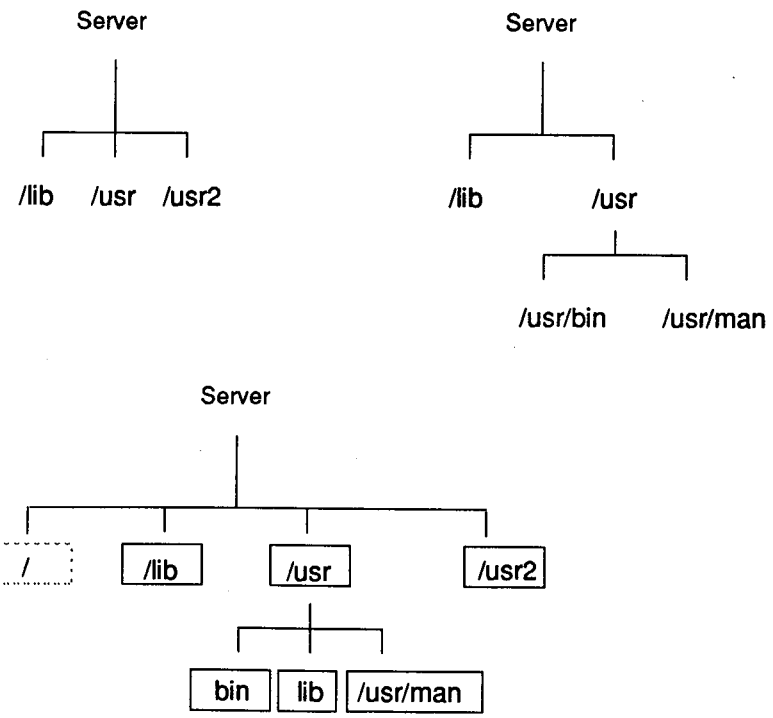
```
client# /etc/mount docserv:/usr/man /usr/man
```

Mounting a remote filesystem

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
/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:/usr/server	326215	245993	47600	84%	/usr2
docserv:/usr/man	346111	216894	94605	70%	/usr/man

You can remote mount not only filesystems, but also directory hierarchies inside filesystems. In this example, `/usr/man` is not a real mount point – it is a subdirectory of the `/usr` filesystem. Here is a diagram of the three machines involved here. Boxes represent remote filesystems, and dotted boxes represent ND partitions.



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 discless node, or has local disc. 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 a source code control system for concurrency control.

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 disc 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.

2 Architecture of NFS

2.1 Design goals

This chapter discusses the design and implementation of the NFS.

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 disc, and reading or writing a file on a disc in the next building. Information on the network is truly distributed.

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.

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.

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 centralised 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 personalisation.

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.

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.

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 discs, and clients with no discs, may yield better performance at lower cost than having many machines with small, inexpensive discs. 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 disc 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 lost if cached information is thrown away. In the case of write-behind, both the client and server attempt to flush critical information to disc 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 disc (and thus did not permit sharing). This goal has been achieved.

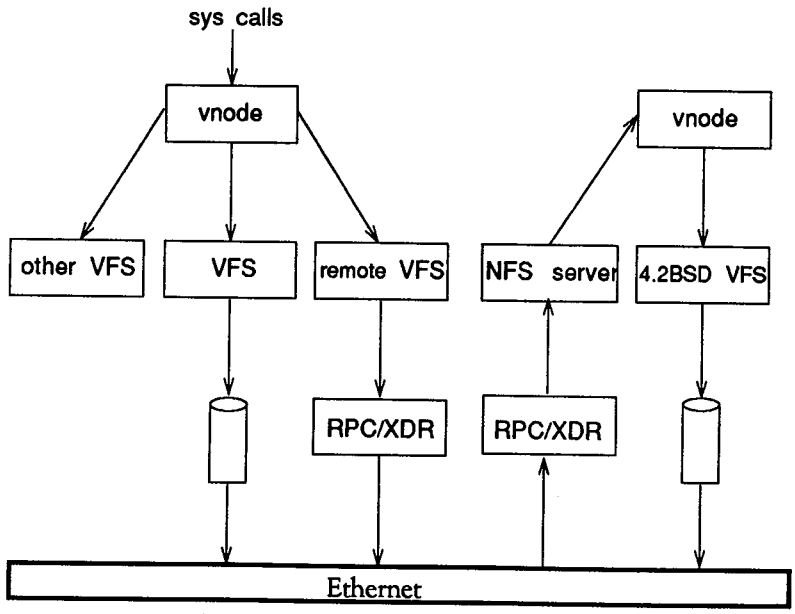
2.2 The NFS Implementation

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 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:

- *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.
- *Filesystem Location:* Since there is no differentiation between a local and a remote VFS, the location of filesystem data is transparent.
- *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.
- *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.
- *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 discs can arrange to share over the NFS without having to appeal to a system administrator, or configure a different system on their workstation.

2.3 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 `unlinks` 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 with 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 initialisation. The corresponding `umount` command 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 synchronisation; however, these will not be made part of the base NFS definition.

3 The Yellow Pages database

This chapter explains Sun's network database mechanism, called the yellow pages. Although this material is not intended for system administrators, it is heavily slanted in that direction.

Sun provides several network services, such as the Network File System (NFS), discussed in the previous sections. 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.

3.1 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.

3.2 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 `hostsmap` 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.

3.3 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`. This information is used internally by the YP.

3.4 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 disc), the NFS (Network File System), and the YP.

ND ND is a method of providing virtual disc, used by discless 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. A process can request information out of the YP database, obviating the need to have such information on every machine. 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. If a remote machine running a YP server process crashes, client processes can obtain YP services from another machine. This is so that YP services are almost always available.

3.5 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. A given server may even be master with regard to one map, and slave with regard to another. This can get confusing quickly. It is recommended that a single server be master for all 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 a database.

4 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.

4.1 The YP network service

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(2)` 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.

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`.

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 ypxfr(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.

Clients

Remember that a client machine (which is not a server) does not access local copies of /etc files, but rather makes an RPC call to a YP server each time it needs information from a YP database. The ypbind(8) daemon remembers the name of a server. When a client boots, ypbind broadcasts asking for the name of the YP server. Similarly, ypbind broadcasts asking for the name of a new YP server if the old server crashes. 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, two new commands ypcat(1) and ypmatch(1) have 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 one of the following commands.

```
% ypcat passwd | grep username
% ypmatch username passwd
```

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

4.2 Default YP files

By default, Sun workstations have seven files from /etcin the YP: /etc/passwd, /etc/group, /etc/hosts, /etc/networks, /etc/services, /etc/protocols, and /etc/ethers. 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.

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.bynamefile. Similarly, gethostbyaddr(3) retrieves the entry from the hosts.byaddrfile. Of course if the YP is not running (which is caused by commenting ypbindout of the /etc/rcfile), then gethostbynamewill read the /etc/hosts files, just as it always has.

Maps sometimes have nicknames. 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. The command ypcat -x furnishes a list of expanded nicknames.

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 and then call yppush. See ypmake(8) for details.

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 always went directly to the YP as does gethostent, 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.

Others

Of the other five 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/services, /etc/protocols, /etc/ethers, and /etc/netgroup are treated like /etc/hosts: for these files, the library routines go directly to the YP, without consulting the local files.

Changing your passwd

To change data in the YP, the system administrator must log into the master machine, and edit databases there; ypwhich 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.

Chapter 2 – 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.

NOTE: Before attempting to write a network application, or to convert an existing non-network application to run over the network, you should be familiar with the material in this chapter. However, for most applications, you can circumvent the need to cope with the kinds of details presented here by using the rpcgen protocol compiler, which is described in detail in the next chapter, the rpcgen Programming Guide. The Generating XDR Routines section of that chapter contains the complete source for a working RPC service—a remote directory listing service which uses rpcgen to generate XDR routines as well as client and server stubs.

What are remote procedure calls? Simply put, they are the high-level communications paradigm used in the operating system. RPC presumes the existence of low-level networking mechanisms (such as TCP/IP and UDP/IP), and upon them it implements a logical client to server communications system designed specifically for the support of network applications. With RPC, the client makes a procedure call 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.

1.1 Layers of RPC

The RPC interface can be seen as being divided into three layers. (For a complete specification of the routines in the remote procedure call Library, see the `rpc` manual page).

The Highest Layer: The highest layer is totally transparent to the operating system, machine and network upon which is is run. It's probably best to think of this level as a way of *using* RPC, rather than as a *part of* RPC proper. Programmers who write RPC routines should (almost) always make this layer available to others by way of a simple C front end to that entirely hides the networking.

To illustrate, at this level a program can simply make a call to `rnusers` a C routine which returns the number of users on a remote machine. The user is not explicitly aware of using RPC – they simply call a procedure, just as they would call `malloc`

The Middle Layer: The middle layer is really 'RPC proper'. Here, the user doesn't need to consider details about sockets, the UNIX system, or other low-level implementation mechanisms. They simple make remote procedure calls to routines on other machines. The selling point here is simplicity. It's this layer that allows RPC to pass the 'hello world' test – simple things should be simple. The middle-layer routines are used for most applications.

RPC calls are made with the system routines `registerrpc` `callrpc` and `svc_run`. The first two of these are the most fundamental: `registerrpc` obtains a unique system-wide procedure-identification number, and `callrpc` actually executes a remote procedure call. At the middle level, a call to `rnusers` is implemented by way of these two routines.

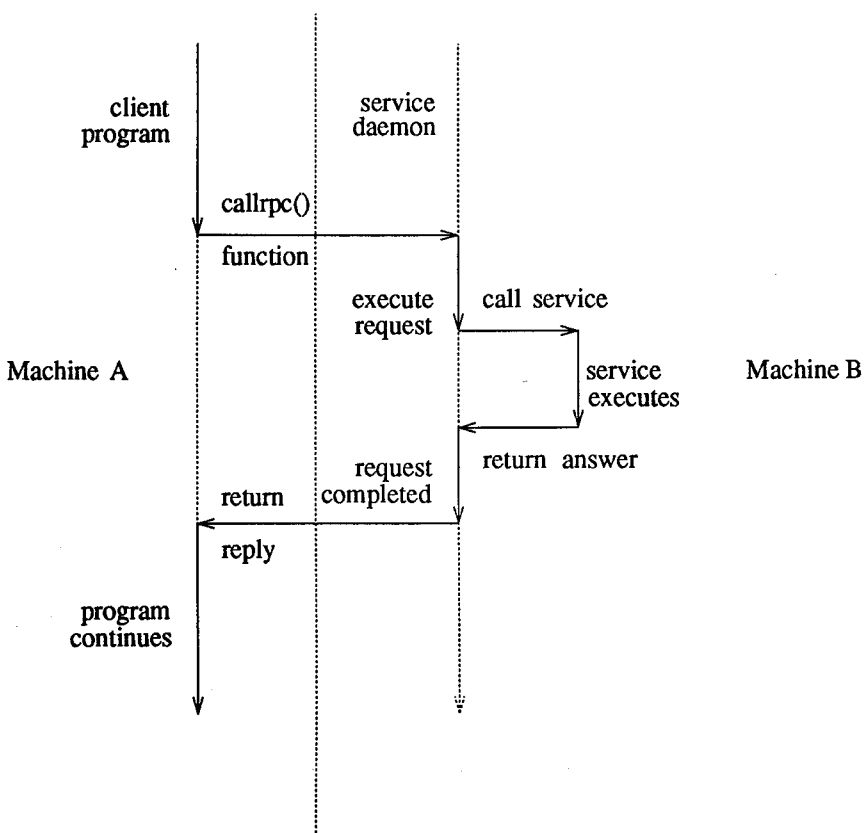
The middle layer is unfortunately rarely used in serious programming due to its inflexibility (simplicity). It does not allow timeout specifications or the choice of transport. It allows no UNIX process control or flexibility in case of errors. It doesn't support multiple kinds of call authentication. The programmer rarely needs all these kinds of control, but one or two of them is often necessary.

The Lowest Layer: The lowest layer does allow these details to be controlled by the programmer, and for that reason it is often necessary. Programs written at this level are also most efficient, but this is rarely a real issue – since RPC clients and servers rarely generate heavy network loads.

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.

1.2 The RPC paradigm

Here is a diagram of the RPC paradigm



2 Higher layers of RPC

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 RPC library routine `rnusers` as illustrated below:

```
#include <stdio.h>
main(argc, argv)
             int argc;
             char **argv;
{
             int 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 in the RPC services library `librpcsvc.a` Thus, the program above should be compiled with

```
% cc program.c -lrpcsvc
```

Here are some of the RPC service library routines available to the C programmer:

Routine	Description
`rnusers`	Return number of users on remote machine
`rusers`	Return information about users on remote machine
`havedisk`	Determine if remote machine has disc
`rstats`	Get performance data from remote kernel
`rwall`	Write to specified remote machines
`yppasswd`	Update user password in Yellow Pages

Other RPC services – for example `ether mount rquota` and `spray` – are not available to the C programmer as library routines. They do, however, have RPC program numbers so they can be invoked with `callrpc` which will be discussed in the next section. Most of them also have compilable `rpcgen` protocol description files. (The `rpcgen` protocol compiler radically simplifies the process of developing network applications. See the `rpcgen` Programming Guide chapter for detailed information about `rpcgen` and `rpcgen` protocol description files).

2.2 Intermediate layer

The simplest interface, which explicitly makes RPC calls, uses the functions `callrpc` and `registerrpc`. Using this method, the number of remote users can be arrived at as follows:

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

main(argc, argv)
        int argc;
        char **argv;
{
        unsigned long nusers;
        int stat;
        if (argc < 2) {
            fprintf(stderr, "usage: nusers hostname\n");
            exit(-1);
        }
        if (stat = callrpc(argv[1],
          RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
          xdr_void, 0, xdr_u_long, &nusers) != 0) {
            clnt_perrno(stat);
              exit(1);
        }
        printf("%d users on %s\n", nusers, argv[1]);
        exit(0);
}
```

Each RPC procedure is uniquely defined by a program number, version number, and procedure number. The program number specifies 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, just as you look up the name of a memory allocator when you want to allocate memory.

The simplest way of making remote procedure calls is with the RPC library routine `callrpc`. It has eight parameters. The first is the name of the remote server machine. The next three parameters are the program, version, and procedure numbers – together they identify the procedure to be called. The fifth and sixth parameters are an XDR filter and an argument to be encoded and passed to the remote procedure. The final two parameters are a filter for decoding the results returned by the remote procedure and a pointer to the place where the procedure's results are to be stored. Multiple arguments and results are handled by embedding them in structures. If `callrpc` completes successfully, it returns zero; else it returns a nonzero value. The return codes (of type *enum* cast into an integer) are found in `<rpc/clnt.h>`.

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 <rpc/rpc.h>
#include <utmp.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 registers a C procedure as corresponding to a given 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 local procedure that implements the remote procedure; and `xdr_void` and `xdr_u_long` are the XDR filters for the remote procedure's arguments and results, respectively. (Multiple arguments or multiple results are passed as structures).

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.

After registering the local procedure, the server program's main procedure calls `svc_run` the RPC library's remote procedure dispatcher. It is this function that calls the remote procedures in response to RPC call messages. Note that the dispatcher takes care of decoding remote procedure arguments and encoding results, using the XDR filters specified when the remote procedure was registered.

2.3 Assigning program numbers

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

```
0x0 - 0x1fffffff            Defined by Sun
0x20000000 - 0x3fffffff     Defined by user
0x40000000 - 0x5fffffff     Transient
0x60000000 - 0x7fffffff     Reserved
0x80000000 - 0x9fffffff     Reserved
0xa0000000 - 0xbfffffff     Reserved
0xc0000000 - 0xdfffffff     Reserved
0xe0000000 - 0xffffffff     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.

To register a protocol specification, send a request by network mail to *rpc@sun* or write to:

RPC Administrator
Sun Microsystems
2550 Garcia Ave.
Mountain View, CA 94043

Please include a compilable `rpcgen '.x'` file describing your protocol. You will be given a unique program number in return.

The RPC program numbers and protocol specifications of standard Sun RPC services can be found in the include files in `/usr/include/rpcsvc` These services, however, constitute only a small subset of those which have been registered. The complete list of registered programs, as of the time when this manual was written, is:

RPC Number	Program	Description
100000	PMAPPROG	portmapper
100001	RSTATPROG	remote stats
100002	RUSERSPROG	remote users
100003	NFSPROG	nfs
100004	YPPROG	Yellow Pages
100005	MOUNTPROG	demon
100006	DBXPROG	remote dbx
100007	YPBINDPROG	yp binder
100008	WALLPROG	shutdown msg
100009	YPPASSWDPROG	yppasswd server
100010	ETHERSTATPROG	ether stats
100011	RQUOTAPROG	disc quotas
100012	SPRAYPROG	spray packets
100013	IBM3270PROG	3270 mapper
100014	IBMRJEPROG	RJE mapper
100015	SELNSVCPROG	selection service
100016	RDATABASEPROG	remote database access
100017	REXECPROG	remote execution
100018	ALICEPROG	Alice Office Automation
100019	SCHEDPROG	scheduling service
100020	LOCKPROG	local lock manager
100021	NETLOCKPROG	network lock manager
100022	X25PROG	x.25 inr protocol
100023	STATMON1PROG	status monitor 1
100024	STATMON2PROG	status monitor 2
100025	SELNLIBPROG	selection library
100026	BOOTPARAMPROG	boot parameters service
100027	MAZEPROG	mazewars game
100028	YPUPDATEPROG	yp update
100029	KEYSERVEPROG	key server
100030	SECURECMDPROG	secure login

```
100031      NETFWDIPROG              nfs net forwarder init
100032      NETFWDTPROG              nfs net forwarder trans
100033      SUNLINKMAP_PRO           sunlink MAP
100034      NETMONPROG               network monitor
100035      DBASEPROG                lightweight database
100036      PWDAUTHPROG              password authorization
100037      TFSPROG                  translucent file svc
100038      NSEPROG                  nse server
100039      NSE_ACTIVATE_PROG        nse activate daemon
150001      PCNFSDPROG               pc passwd authorization
200000      PYRAMIDLOCKINGPROG       Pyramid-locking
200001      PYRAMIDSYS5              Pyramid-sys5
200002      CADDS_IMAGE              CV cadds_image
```

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_wrapstring()
xdr_char()     xdr_u_char()
```

Note that the routine `xdr_string` exists, but cannot be used with `callrpc` and `registerrpc` which only pass two parameters to their XDR routines. `xdr_wrapstring` has only two parameters, and is thus OK. It calls `xdr_string`.

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

```
struct simple {
        int a;
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* section of this manual, only few implementation examples are given here.

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

```
xdr_array()       xdr_bytes()      xdr_reference()
xdr_vector()      xdr_union()      xdr_pointer()
xdr_string()      xdr_opaque()
```

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, arrp)
        XDR *xdrsp;
        struct varintarr *arrp;
 {
        return (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, one can use `xdr_vector` which serialises fixed-length arrays.

```
int intarr[SIZE];

xdr_intarr(xdrsp, intarr)
        XDR *xdrsp;
        int intarr[];
 {
        int i;
        return (xdr_vector(xdrsp, intarr, SIZE, sizeof(int),
       xdr_int));
 }
```

XDR always converts quantities to 4-byte multiples when deserialising. 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; `xdr_bytes` has four parameters, similar to 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 serialising it gets the string length from `strlen` and on deserialising 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;
 {

        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 Lowest layer 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.

There are several occasions when you may need to use lower layers of RPC. First, you may need to use TCP, since the higher layer uses UDP, which restricts RPC calls to 8K bytes of data. Using TCP permits calls to send long streams of data. For an example, see the *TCP* section below. Second, you may want to allocate and free memory while serialising or deserialising with XDR routines. There is no call at the higher level to let you free memory explicitly. For more explanation, see the *Memory Allocation with XDR* section below. Third, you may need to perform authentication on either the client or server side, by supplying credentials or verifying them. See the explanation in the *Authentication* section below.

3.1 More on the server side

The server for the `nusers` program shown below does the same thing as the one using `registerrpc` above, but is written using a lower layer of the RPC package:

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

main()
{
        SVCXPRT *transp;
        int nuser();

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

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

        switch (rqstp->rq_proc) {
        case NULLPROC:
            if (!svc_sendreply(transp, xdr_void, 0))
                fprintf(stderr, "can't reply to RPC call\n");
        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, "can't reply to RPC call\n");
            return;
        default:
            svcerr_noproc(transp);
            return;
            }
    }
```

First, the server gets a transport handle, which is used for receiving and replying to RPC messages. `registerrpc` uses `svcudp_create` to get a UDP handle. If you require a more reliable protocol, call `svctcp_create` instead. If the argument to `svcudp_create` is `RPC_ANYSOCK` the RPC library creates a socket on which to receive and reply to 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 `clntcp_create` (the low-level client routine) must match.

If the user specifies the `RPC_ANYSOCK` argument, the RPC library routines will open sockets. Otherwise they will expect the user to do so. The routines `svcudp_create` and `clntudp_create` will cause the RPC library routines to `bind` their socket if it is not bound already.

A service may choose to register its port number with the local portmapper service. This is done is done by specifying a non-zero protocol number in `svc_register` Incidently, a client can discover the server's port number by consulting the portmapper on their server's machine. This can be done automatically by specifying a zero port number in `clntudp_create` or `clntcp_create`.

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 `RUSERSPROG` 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 results. 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 serialises 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 receives 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, "can'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);
```

Space has already been allocated in `chararr`. 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(transp, xdr_chararr2, &arrptr);
```

Note that, after being used, the character array can be freed with `svc_freeargs`. `svc_freeargs` will not attempt to free any memory if the variable indicating it is NULL. For example, in the the routine `xdr_finalexample` given earlier, if `finalp->string` was NULL, then it would not be freed. The same is true for `finalp->simplep`.

To summarise, each XDR routine is responsible for serialising, deserialising, and freeing memory. When an XDR routine is called from `callrpc` the serialising part is used. When called from `svc_getargs` the deserialiser 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. See the *eXternal Data Representation:* chapter for examples of more sophisticated XDR routines that determine which of the three modes they are in and adjust their behaviour accordingly.

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 <utmp.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 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, "can't get addr for %s\n",argv[1]);
                exit(-1);
        }
        pertry_timeout.tv_sec = 3;
        pertry_timeout.tv_usec = 0;
        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) {
                clnt_pcreateerror("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);
        close(sock);
}
```

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 serialising the argument, a pointer to the argument, the XDR routine for deserialising 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 (Transmission Control Protocol), you would use `clnttcp_create`.

The parameters to `clntudp_create` are 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`. This makes it possible, in cases where there are multiple client handles using the same socket, to destroy one handle without closing 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, &sock,
                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`:

```
transp = svctcp_create(RPC_ANYSOCK, 0, 0);
```

The last two arguments to `svctcp_create` are send and receive sizes respectively. If '0' is specified for either of these, the system chooses a reasonable default.

4 Other RPC features

4.1 Select on the server side

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

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 file descriptor, the `svc_run` call won't work. The code for `svc_run` is as follows:

```
void
svc_run()
{
        fd_set readfds;
        int dtbsz = getdtablesize();

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

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

You can bypass `svc_run` and call `svc_getreqset` 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. Note that `svc_fds` is a bit mask of all the file descriptors that RPC is using for services. It can change everytime that *any* RPC library routine is called, because descriptors are constantly being opened and closed, for example for TCP connections.

4.2 Broadcast RPC

The `portmapper` is a daemon that converts RPC program numbers into DARPA protocol port numbers; see the *portmap* man page. You can't do broadcast RPC without the portmapper. Here are the main differences between broadcast RPC and normal RPC calls:

- Normal RPC expects one answer, whereas broadcast RPC expects many answers (one or more answer from each responding machine).
- Broadcast RPC can only be supported by packet-oriented (connectionless) transport protocols like UPD/IP.
- 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.
- 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.

Broadcast RPC synopsis

```
#include <rpc/pmap_clnt.h>
 . . .
enum clnt_stat  clnt_stat;
        . . .
clnt_stat = clnt_broadcast(prognum, versnum, procnum,
  inproc, in, outproc, out, eachresult)
        u_long    prognum;        /* program number */
        u_long    versnum;        /* version number */
        u_long    procnum;        /* procedure number */
        xdrproc_t inproc;         /* xdr routine for args */
        caddr_t   in;             /* pointer to args */
        xdrproc_t outproc;        /* xdr routine for results */
        caddr_t   out;            /* pointer to results */
        bool_t    (*eachresult)();/* call with each result gotten */
```

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

```
bool_t done;
        . . .
 done = eachresult(resultsp, raddr)
        caddr_t resultsp;
        struct sockaddr_in *raddr;  /* Addr of responding machine */
```

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`

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:

- each RPC call in the pipeline requires no response from the server, and the server does not send a response message;
- 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 <suntool/windows.h>

void windowdispatch();

main()
{
        SVCXPRT *transp;

        transp = svctcp_create(RPC_ANYSOCK, 0, 0);
        if (transp == NULL){
                fprintf(stderr, "can't create an RPC server\n");
                exit(1);
        }
        pmap_unset(WINDOWPROG, WINDOWVERS);
        if (!svc_register(transp, WINDOWPROG, WINDOWVERS,
          windowdispatch, IPPROTO_TCP)) {
                fprintf(stderr, "can'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))
                   printf(stderr, "can't reply to RPC call\n");
                return;
        case RENDERSTRING:
                if (!svc_getargs(transp, xdr_wrapstring, &s)) {
                    printf(stderr, "can't decode arguments\n");
                        /*
                         * Tell caller he screwed up
                         */
                        svcerr_decode(transp);
                        break;
                }
                /*
                 * Call here to render the string s
                 */
                if (!svc_sendreply(transp, xdr_void, NULL))
                  fprintf(stderr, "can't reply to RPC call\n");
                break;
        case RENDERSTRING_BATCHED:

                if (!svc_getargs(transp, xdr_wrapstring, &s)) {
                   fprintf(stderr, "can't decode arguments\n");
```

```
                        /*
                         * We are silent in the face of
                         * protocol errors
                         */
                        break;
                }
                /*
                 * Call here to render string s, but send 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:

- the result's XDR routine must be zero NULL
- 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 <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>
#include <suntool/windows.h>

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

        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 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.

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 where client is */
    int      aup_uid;        /* client's UNIX effective uid */
    int      aup_gid;        /* client's current group id */
    u_int    aup_len;        /* element length of aup_gids */
    int      *aup_gids;      /* array of groups user is in */
}
```

These fields are set by `authunix_create_default` by invoking the appropriate system calls. Since the RPC user created this new style of authentication, the user is responsible for destroying it with:

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

This should be done in all cases, to conserve memory.

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 vers num */
    u_long    rq_proc;          /* desired procedure number */
    struct opaque_auth rq_cred; /* raw credentials from wire */
    caddr_t   rq_clntcred;      /* credentials (read only) */
}
```

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

- 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.
- 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) fields 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, transp)
        struct svc_req *rqstp;
        SVCXPRT *transp;
{
        struct authunix_parms *unix_cred;
        int uid;
        unsigned long nusers;

        /*
         * we don't care about authentication for null proc
         */
        if (rqstp->rq_proc == NULLPROC) {
                if (!svc_sendreply(transp, xdr_void, 0)) {
                  fprintf(stderr, "can'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 caller is allowed to call this
                 * proc
                 */
                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, "can'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 the service creation routine should be called in the following form:

```
transp = svcudp_create(0); /* For UDP */
transp = svctcp_create(0,0,0); /* For listener TCP sockets */
transp = svcfd_create(0,0,0);  /* For connected TCP sockets */
```

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

```
svc_register(transp, PROGNUM, VERSNUM, service, 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/inetd.conf` for RPC services is in one of the following two forms:

```
p_name/version dgram  rpc/udp wait/nowait user server args
p_name/version stream rpc/tcp wait/nowait user server args
```

where *p_name* is the symbolic name of the program as it appears in *rpc server* is the C code implementing the server, and `program` and `version` are the program and version numbers of the service. For more information, see `inetd.conf`.

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

```
rstatd/1-2 dgram rpc/udp wait root /usr/etc/rpc.rstatd
```

5 More Examples

5.1 Versions

By convention, the first version number of program PROG is PROGVERS_ORIG and the most recent version is PROGVERS. Suppose there is a new version of the *user* program that returns an *unsigned* 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, "can't register RUSER service\n");
        exit(1);
}
if (!svc_register(transp, RUSERSPROG, RUSERSVERS_SHORT,
  nuser, IPPROTO_TCP)) {
        fprintf(stderr, "can't register RUSER service\n");
        exit(1);
}
```

Both versions can be handled by the same C procedure:

```
nuser(rqstp, transp)
        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, "can'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;
                switch (rqstp->rq_vers) {
                case RUSERSVERS_ORIG:

               if (!svc_sendreply(transp, xdr_u_long, &nusers))
              {
                   fprintf(stderr, "can't reply to RPC call\n");
                        }
                        break;
                case RUSERSVERS_SHORT:
           if (!svc_sendreply(transp, xdr_u_short, &nusers2))
{
                   fprintf(stderr, "can't reply to RPC call\n");
                        }
                        break;
                }
        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 serialisation than on deserialisation.

```
/*
 * 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[BUFSIZ], *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), BUFSIZ,
                          fp)) == 0 && ferror(fp)) {
                             fprintf(stderr, "can't fread\n");
                                exit(1);
                        }
                }
                p = buf;
                if (!xdr_bytes(xdrs, &p, &size, BUFSIZ))
                        return 0;
                if (size == 0)
                        return 1;
                if (xdrs->x_op == XDR_DECODE) {
                        if (fwrite(buf, sizeof(char), size,
                          fp) != size) {
                            fprintf(stderr, "can'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 xdr_rcp();
        int err;

        if (argc < 2) {
            fprintf(stderr, "usage: %s servername\n", argv[0]);
                exit(-1);
        }
        if ((err = callrpctcp(argv[1], RCPPROG, RCPPROC,
          RCPVERS, xdr_rcp, stdin, xdr_void, 0) != 0)) {
                clnt_perrno(err);
                fprintf(stderr, "can'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, "can't 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;
     int rcp_service(), xdr_rcp();

        if ((transp = svctcp_create(RPC_ANYSOCK,
          BUFSIZ, BUFSIZ)) == 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 proto, vers, *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 error
```

```
         */
        bind(s, &addr, len);
        if (getsockname(s, &addr, &len)< 0) {
                perror("getsockname");
                return (0);
        }
        while (!pmap_set(prognum++, vers, proto,
                ntohs(addr.sin_port))) continue;
        return (prognum-1);
}
```

NOTE: The call to `ntohs` is necessary to ensure that the port number in addr.sin_port which is in network byte order, is passed in host byte order (as pmap_set expects). See the `byteorder` man page for more details on the conversion of network addresses from network to host byte order.

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 `EXAMPLEPROG` 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()
{
        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);
        }
        /* protocol is 0 - gettransient() does registering
         */
        (void)svc_register(xprt, x, 1, callback, 0);
        ans = callrpc(hostname, EXAMPLEPROG, EXAMPLEVERS,
              EXAMPLEPROC_CALLBACK, xdr_int, &x, xdr_void, 0);
        if ((enum clnt_stat) ans != RPC_SUCCESS) {
                fprintf(stderr, "call: ");
                clnt_perrno(ans);
                fprintf(stderr, "\n");
        }
        svc_run();
        fprintf(stderr, "Error: svc_run shouldn't return\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)) {
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)) {
                         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()
{
        gethostname(hostname, sizeof(hostname));
        registerrpc(EXAMPLEPROG, EXAMPLEVERS,
          EXAMPLEPROC_CALLBACK, getnewprog, xdr_int, xdr_void);
        fprintf(stderr, "server going into svc_run\n");
        signal(SIGALRM, docallback);
        alarm(10);
        svc_run();
        fprintf(stderr, "Error: svc_run shouldn't return\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");
        }
}
```

Chapter 3 – The rpcgen programming guide

1 The rpcgen protocol compiler

1.1 The rpcgen protocol compiler

The details of programming applications to use Remote Procedure Calls can be overwhelming. Perhaps most daunting is the writing of the XDR routines necessary to convert procedure arguments and results into their network format and vice-versa.

Fortunately, rpcgen exists to help programmers write RPC applications simply and directly. rpcgen does most of the dirty work, allowing programmers to debug the main features of their application, instead of requiring them to spend most of their time debugging their network interface code.

rpcgen is a compiler. It accepts a remote program interface definition written in a language, called RPC Language, which is similar to C. It produces a C language output which includes stub versions of the client routines, a server skeleton, XDR filter routines for both parameters and results, and a header file that contains common definitions. The client stubs interface with the RPC library and effectively hide the network from their callers. The server stub similarly hides the network from the server procedures that are to be invoked by remote clients. rpcgen output files can be compiled and linked in the usual way. The developer writes server procedures – in any language that observes Sun calling conventions – and links them with the server skeleton produced by rpcgen to get an executable server program. To use a remote program, a programmer writes an ordinary main program that makes local procedure calls to the client stubs produced by rpcgen. Linking this program with rpcgen stubs creates an executable program. (At present the main program must be written in C.) rpcgen options can be used to suppress stub generation and to specify the transport to be used by the server stub.

Like all compilers, rpcgen reduces development time that would otherwise be spent coding and debugging low-level routines. All compilers, including rpcgen do this at a small cost in efficiency and flexibility. However, many compilers allow escape hatches for programmers to mix low-level code with high-level code. rpcgen is no exception. In speed-critical applications, hand-written routines can be linked with the rpcgen output without any difficulty. Also, one may proceed by using rpcgen output as a starting point, and rewriting it as necessary.

1.2 Converting local procedures into remote procedures

Assume an application that runs on a single machine, one which we want to convert to run over the network. Here we will demonstrate such a conversion by way of a simple example – a program that prints a message to the console:

```
/*
 * printmsg.c: print a message on the console
 */
#include <stdio.h>

main(argc, argv) int argc;

        char *argv[];
{
        char *message;
        if (argc < 2) {
             fprintf(stderr, "usage: %s <message>\n", argv[0]);
```

```
                exit(1);
        }
        message = argv[1];
        if (!printmessage(message)) {
           fprintf(stderr, "%s: couldn't print your message\n",
                        argv[0]);
                exit(1);
        }
        printf("Message delivered!0);
}
/*
 * Print a message to the console.
 * Return a boolean indicating whether the message was actually printed.
 */
printmessage(msg)
        char *msg;
{
        FILE *f;
        f = fopen("/dev/console", "w");
        if (f == NULL) {
                return (0);
        }
        fprintf(f, "%s\n", msg);
        fclose(f);
        return(1);
}
```

And then, of course:

```
example%  cc printmsg.c -o printmsg
example%  printmsg "Hello, there."
Message delivered!
example%.
```

If `printmessage` was turned into a remote procedure, then it could be called from anywhere in the network. Ideally, one would just like to stick a keyword like *remote* in front of a procedure to turn it into a remote procedure. Unfortunately, we have to live within the constraints of the C language, since it existed long before RPC did. But even without language support, it's not very difficult to make a procedure remote.

In general, it's necessary to figure out what the types are for all procedure inputs and outputs. In this case, we have a procedure `printmessage` which takes a string as input, and returns an integer as output. Knowing this, we can write a protocol specification in RPC language that describes the remote version of `printmessage` Here it is:

```
/*
 * msg.x: Remote message printing protocol
 */

program MESSAGEPROG {
        version MESSAGEVERS {
                int PRINTMESSAGE(string) = 1;
        } = 1;
} = 99;
```

Remote procedures are part of remote programs, so we actually declared an entire remote program here which contains the single procedure `PRINTMESSAGE` This procedure was declared to be in version 1 of the remote program. No null procedure (procedure 0) is necessary because `rpcgen` generates it automatically.

Notice that everything is declared with all capital letters. This is not required, but is a good convention to follow.

Notice also that the argument type is 'string' and not 'char *'. This is because a 'char *' in C is ambiguous. Programmers usually intend it to mean a null-terminated string of characters, but it could also represent a pointer to a single character or a

pointer to an array of characters. In RPC language, a null-terminated string is unambiguously called a 'string'.

There are just two more things to write. First, there is the remote procedure itself. Here's the definition of a remote procedure to implement the PRINTMESSAGE procedure we declared above:

```
/*
 * msg_proc.c: implementation of the remote procedure "printmessage"
 */

#include <stdio.h>
#include <rpc/rpc.h>     /* always needed  */
#include "msg.h"         /* need this too: msg.h will be generated by rpcgen
                         */
/*
 * Remote verson of "printmessage"
 */
int *
printmessage_1(msg)
        char **msg;
{
        static int result;  /* must be static! */
        FILE *f;

        f = fopen("/dev/console", "w");
        if (f == NULL) {
                result = 0;
                return (&result);
        }
        fprintf(f, "%s\n", *msg);
        fclose(f);
        result = 1;
        return (&result);
}
```

Notice here that the declaration of the remote procedure `printmessage_1` differs from that of the local procedure `printmessage` in three ways:

1 It takes a pointer to a string instead of a string itself. This is true of all remote procedures: they always take pointers to their arguments rather than the arguments themselves.

2 It returns a pointer to an integer instead of an integer itself. This is also generally true of remote procedures: they always return a pointer to their results.

3 It has an '_1' appended to its name. In general, all remote procedures called by rpcgen are named by the following rule: the name in the program definition (here PRINTMESSAGE is converted to all lower-case letters, an underbar ('_') is appended to it, and finally the version number (here 1) is appended.

The last thing to do is declare the main client program that will call the remote procedure. Here it is:

```
/*
 * rprintmsg.c: remote version of "printmsg.c"
 */
#include <stdio.h>
#include <rpc/rpc.h>     /* always needed  */
#include "msg.h"         /* need this too: msg.h will be generated by rpcgen */

main(argc, argv)
        int argc;
        char *argv[];
{
        CLIENT *cl;
        int *result;
        char *server;
        char *message;
```

```
        if (argc < 3) {
          fprintf(stderr, "usage: %s host message\n", argv[0]);
                exit(1);
        }

        /*
         * Save values of command line arguments
         */
        server = argv[1];
        message = argv[2];

/*
 * Create client "handle" used for calling MESSAGEPROG on the
 * server designated on the command line.We tell the RPC package
 * to use the "tcp" protocol when contacting the server.
 */
        cl = clnt_create(server, MESSAGEPROG, MESSAGEVERS, "tcp");
        if (cl == NULL) {
                /*
                 * Couldn't establish connection with server.
                 * Print error message and die.
                 */
                clnt_pcreateerror(server);
                exit(1);
        }

/*
 * Call the remote procedure "printmessage" on the server
 */
        result = printmessage_1(&message, cl);
        if (result == NULL) {
                /*
                 * An error occurred while calling the server.
                 * Print error message and die.
                 */
                clnt_perror(cl, server);
                exit(1);
        }

/*
 * Okay, we successfully called the remote procedure.
 */
        if (*result == 0) {
                /*
                 * Server was unable to print our message.
                 * Print error message and die.
                 */
        fprintf(stderr, "%s: %s couldn't print your message\n",
                        argv[0], server);
                exit(1);
        }

/*
 * The message got printed on the server's console
 */
        printf("Message delivered to %s!\n", server);
}
```

There are two things to note here:

1 First a client 'handle' is created using the RPC library routine `clnt_create`. This client handle will be passed to the stub routines which call the remote procedure.

2 The remote procedure `printmessage_1` is called exactly the same way as it is declared in `msg_proc.c` except for the inserted client handle as the first argument.

Here's how to put all of the pieces together:

```
example% rpcgen msg.x
example% cc rprintmsg.c msg_clnt.c -o rprintmsg
example% cc msg_proc.c msg_svc.c -o msg_server
```

Two programs were compiled here: the client program `printmsg` and the server program `msg_server` Before doing this though, `rpcgen` was used to fill in the missing pieces.

Here is what `rpcgen` did with the input file `msg.x`

1 It created a header file called *msg.h* that contained `#define`'s for `MESSAGEPROG MESSAGEVERS` and `PRINTMESSAGE` for use in the other modules.

2 It created client 'stub' routines in the `msg_clnt.c` file. In this case there is only one, the `printmessage_1` that was referred to from the *`printmsg`* client program. The name of the output file for client stub routines is always formed in this way: if the name of the input file is `FOO.x` the client stubs output file is called `FOO_clnt.c`.

3 It created the server program which calls `printmessage_1` in `msg_proc.c`. This server program is named `msg_svc.c`. The rule for naming the server output file is similar to the previous one: for an input file called `FOO.x` the output server file is named `FOO_svc.c`.

Now we're ready to have some fun. First, copy the server to a remote machine and run it. For this example, the machine is called *moon*. Server processes are run in the background, because they never exit.

```
moon% msg_server &
```

Then on our local machine (*sun*) we can print a message on *moons* console:

```
sun% printmsg moon "Hello, moon."
```

The message will get printed to *moons* console. You can print a message on anybody's console (including your own) with this program if you are able to copy the server to their machine and run it.

1.3 Generating XDR Routines

The previous example only demonstrated the automatic generation of client and server RPC code. `rpcgen` may also be used to generate XDR routines, that is, the routines necessary to convert local data structures into network format and vice-versa. This example presents a complete RPC service – a remote directory listing service, which uses `rpcgen` not only to generate stub routines, but also to generate the XDR routines. Here is the protocol description file:

```
/*
 * dir.x: Remote directory listing protocol
 */
const MAXNAMELEN = 255;
 /* maximum length of a directory entry */
typedef string nametype<MAXNAMELEN>;
 /* a directory entry */
typedef struct namenode *namelist;
 /* a link in the listing */
/*
 * A node in the directory listing
 */
struct namenode {
        nametype name;          /* name of directory entry */
        namelist next;          /* next entry */
};

/*
 * The result of a READDIR operation.
 */
union readdir_res switch (int errno) {
case 0:
        namelist list; /* no error: return directory listing */
```

```
default:
        void;       /* error occurred: nothing else to return */
};

/*
 * The directory program definition
 */
program DIRPROG {
        version DIRVERS {
                readdir_res
                READDIR(nametype) = 1;
        } = 1;
} = 76;
```

Running `rpcgen` on `dir.x` creates four output files. Three are the same as before: header file, client stub routines and server skeleton. The fourth are the XDR routines necessary for converting the data types we declared into XDR format and vice-versa. These are output in the file `dir_xdr.c`.

Here is the implementation of the READDIR procedure:

```
/*
 * dir_proc.c: remote readdir implementation
 */
#include <rpc/rpc.h>
#include <sys/dir.h>
#include "dir.h"

extern int errno;
extern char *malloc();
extern char *strdup();

readdir_res *
readdir_1(dirname)
        nametype *dirname;
{
        DIR *dirp;
        struct direct *d;
        namelist nl;
        namelist *nlp;
        static readdir_res res; /* must be static! */

        /*
         * Open directory
         */
        dirp = opendir(*dirname);
        if (dirp == NULL) {
                res.errno = errno;
                return (&res);
        }

        /*
         * Free previous result
         */
        xdr_free(xdr_readdir_res, &res);

        /*
         * Collect directory entries
         */
        nlp = &res.readdir_res_u.list;
        while (d = readdir(dirp)) {
                nl = *nlp = (namenode *) malloc(sizeof(namenode));
                nl->name = strdup(d->d_name);
                nlp = &nl->next;
        }
        *nlp = NULL;
        /*
         * Return the result
         */
        res.errno = 0;
        closedir(dirp);
        return (&res);
}
```

Finally, there is the client side program to call the server:

```
/*
 * rls.c: Remote directory listing client
 */
#include <stdio.h>
#include <rpc/rpc.h>    /* always need this */
#include "dir.h"
/* need this too: will be generated by rpcgen */

extern int errno;

main(argc, argv)
        int argc;
        char *argv[];
{
        CLIENT *cl;
        char *server;
        char *dir;
        readdir_res *result;
        namelist nl;


        if (argc != 3) {
        fprintf(stderr, "usage: %s host directory\n", argv[0]);
                exit(1);
        }

        /*
         * Remember what our command line arguments refer to
         */
        server = argv[1];
        dir = argv[2];

/*
 * Create client "handle" used for calling MESSAGEPROG on the
 * server designated on the command line.We tell the RPC package
 * to use the "tcp" protocol when contacting the server.
 */
        cl = clnt_create(server, DIRPROG, DIRVERS, "tcp");
        if (cl == NULL) {
                /*
                 * Couldn't establish connection with server.
                 * Print error message and die.
                 */
                clnt_pcreateerror(server);
                exit(1);
        }

        /*
         * Call the remote procedure readdir on the server
         */
        result = readdir_1(&dir, cl);
        if (result == NULL) {
                /*
                 * An error occurred while calling the server.
                 * Print error message and die.
                 */
                clnt_perror(cl, server);
                exit(1);
        }

        /*
         * Okay, we successfully called the remote procedure.
         */
        if (result->errno != 0) {
                /*
                 * A remote system error occurred.
                 * Print error message and die.
                 */
                errno = result->errno;
                perror(dir);
                exit(1);
        }
```

```
        /*
         * Successfully got a directory listing.
         * Print it out.
         */
        for (nl = result->readdir_res_u.list; nl != NULL;
          nl = nl->next) {
                printf("%s0, nl->name);
        }
}
```

Compile everything, and run.

```
sun%  rpcgen dir.x
sun%  cc rls.c dir_clnt.c dir_xdr.c -o rls
sun%  cc dir_svc.c dir_proc.c dir_xdr.c -o dir_svc

sun%  dir_svc &

moon%  rls sun /usr/pub
.
..
ascii
eqnchar
greek
kbd
marg8
tabclr
tabs
tabs4
moon%
```

A final note about rpcgen. The client program and the server procedure can be tested together as a single program by simply linking them with each other rather than with the client and server stubs. The procedure calls will be executed as ordinary local procedure calls and the program can be debugged with a local debugger such as dbx. When the program is working, the client program can be linked to the client stub produced by rpcgen and the server procedures can be linked to the server stub produced by rpcgen.

NOTE: If you do this, you may want to comment out calls to RPC library routines, and have client-side routines call server routines directly.

1.4 The C-Preprocessor

The C-preprocessor is run on all input files before they are compiled, so all the preprocessor directives are legal within a ".x" file. Four symbols may be defined, depending upon which output file is getting generated. The symbols are:

Symbol	Usage
RPC_HDR	for header-file output
RPC_XDR	for XDR routine output
RPC_SVC	for server-skeleton output
RPC_CLNT	for client stub output

Also, rpcgen does a little preprocessing of its own. Any line that begins with a percent sign is passed directly into the output file, without any interpretation of the line. Here is a simple example that demonstrates the preprocessing features.

```
/*
 * time.x: Remote time protocol
 */
program TIMEPROG {
        version TIMEVERS {
                unsigned int TIMEGET(void) = 1;
        } = 1;
} = 44;

#ifdef RPC_SVC
%int *
```

```
%timeget_1()
%{
%       static int thetime;
%
%       thetime = time(0);
%       return (&thetime);
%}
#endif
```

The '%' feature is not generally recommended, as there is no guarantee that the compiler will stick the output where you intended.

1.5 The RPC language

RPC language is an extension of XDR language. The sole extension is the addition of the *program* type. For a complete description of the XDR language syntax, see the *eXternal Data Representation Standard: Protocol Specification* chapter. For a description of the RPC extensions to the XDR language, see the *Remote Procedure Calls: Protocol Specification* chapter.

However, XDR language is so close to C that if you know C, you know most of it already. We describe here the syntax of the RPC language, showing a few examples along the way. We also show how the various RPC and XDR type definitions get compiled into C type definitions in the output header file.

Definitions

An RPC language file consists of a series of definitions.

```
definition-list:
definition ";"
definition ";" definition-list
```

It recognizes five types of definitions.

```
definition:
enum-definition
struct-definition
union-definition
typedef-definition
const-definition
program-definition
```

Structures

An XDR struct is declared almost exactly like its C counterpart. It looks like the following:

```
struct-definition:
    "struct" struct-ident "{"
        declaration-list
    "}"

declaration-list:
    declaration ";"
    declaration ";" declaration-list
```

As an example, here is an XDR structure to define a two-dimensional coordinate, and the C structure that it gets compiled into in the output header file.

```
struct coord {              struct coord {
    int x;        -->           int x;
    int y;                      int y;
};                          };
                            typedef struct coord coord;
```

The output is identical to the input, except for the added `typedef` at the end of the output. This allows one to use 'coord' instead of 'struct coord' when declaring items.

Unions

XDR unions are discriminated unions, and look quite different from C unions. They are more analogous to Pascal variant records than they are to C unions.

```
union-definition:
    "union" union-ident "switch" "(" declaration ")" "{"
        case-list
    "}"

case-list:
    "case" value ":" declaration ";"
    "default" ":" declaration ";"
    "case" value ":" declaration ";" case-list
```

Here is an example of a type that might be returned as the result of a 'read data' operation. If there is no error, return a block of data. Otherwise, don't return anything.

```
union read_result switch (int errno) {
case 0:
    opaque data[1024];
default:
    void;
};
```

It gets compiled into the following:

```
struct read_result {
    int errno;
    union {
        char data[1024];
    } read_result_u;
};
typedef struct read_result read_result;
```

Notice that the union component of the output struct has the name as the type name, except for the trailing '_u'.

Enumerations

XDR enumerations have the same syntax as C enumerations.

```
enum-definition:
    "enum" enum-ident "{"
        enum-value-list
    "}"

enum-value-list:
    enum-value
    enum-value "," enum-value-list

enum-value:
    enum-value-ident
    enum-value-ident "=" value
```

Here is a short example of an XDR enum, and the C enum that it gets compiled into.

```
enum colortype {        enum colortype {
    RED = 0,                RED = 0,
    GREEN = 1,    -->       GREEN = 1,
    BLUE = 2                BLUE = 2,
};                      };
                        typedef enum colortype colortype;
```

Typedef

XDR typedefs have the same syntax as C typedefs.

```
typedef-definition:
    "typedef" declaration
```

Here is an example that defines a `fname_type` used for declaring file name strings that have a maximum length of 255 characters.

```
typedef string fname_type<255>; --> typedef char *fname_type;
```

Constants

XDR constants symbolic constants that may be used wherever a integer constant is used, for example, in array size specifications

```
const-definition:
    "const" const-ident "=" integer
```

For example, the following defines a constant DOZEN equal to 12.

```
const DOZEN = 12;  -->  #define DOZEN 12
```

Programs

RPC programs are declared using the following syntax:

```
program-definition:
    "program" program-ident "{"
        version-list
    "}" "=" value

version-list:
    version ";"
    version ";" version-list

version:
    "version" version-ident "{"
        procedure-list
    "}" "=" value

procedure-list:
    procedure ";"
    procedure ";" procedure-list

procedure:
   type-ident procedure-ident "(" type-ident ")" "=" value
```

For example, here is the time protocol, revisited:

```
/*
 * time.x: Get or set the time.Time is represented as number
 * of seconds
 * since 0:00, January 1, 1970.
 */

program TIMEPROG {
    version TIMEVERS {
        unsigned int TIMEGET(void) = 1;
        void TIMESET(unsigned) = 2;
    } = 1;
} = 44;
```

This file compiles into #defines in the output header file:

```
#define TIMEPROG 44
#define TIMEVERS 1
#define TIMEGET 1
#define TIMESET 2
```

Declarations

In XDR, there are only four kinds of declarations.

```
declaration:
     simple-declaration
     fixed-array-declaration
     variable-array-declaration
     pointer-declaration
```

Simple declarations are just like simple C declarations.

```
simple-declaration:
    type-ident variable-ident
```

Example:

```
colortype color;    --> colortype color;
```

Fixed-length array declarations are just like C array declarations:

```
fixed-array-declaration:
    type-ident variable-ident "[" value "]"
```

Example:

```
colortype palette[8];    --> colortype palette[8];
```

Variable-length array declarations have no explicit syntax in C, so XDR invents its own using angle-brackets.

```
variable-array-declaration:
    type-ident variable-ident "<" value ">"
    type-ident variable-ident "<" ">"
```

The maximum size is specified between the angle brackets. The size may be omitted, indicating that the array may be of any size.

```
int heights<12>;    /* at most 12 items */
int widths<>;       /* any number of items */
```

Since variable-length arrays have no explicit syntax in C, these declarations are actually compiled into 'struct's. For example, the 'heights' declaration gets compiled into the following struct:

```
struct {
    u_int heights_len;  /* # of items in array */
    int *heights_val;   /* pointer to array */
} heights;
```

Note that the number of items in the array is stored in the '_len' component and the pointer to the array is stored in the '_val' component. The first part of each of these component's names is the same as the name of the declared XDR variable.

4) Pointer Declarations are made in XDR exactly as they are in C. You can't really send pointers over the network, but you can use XDR pointers for sending recursive data types such as lists and trees. The type is actually called 'optional-data', not 'pointer', in XDR language.

```
pointer-declaration:
    type-ident "*" variable-ident
```

Example:

```
listitem *next;  -->  listitem *next;
```

Special cases

There are a few exceptions to the rules described above.

Booleans

C has no built-in boolean type. However, the RPC library does a boolean type called `bool_t` that is either *TRUE* or *FALSE*. Things declared as type in XDR language are compiled into `bool_t` in the output header file.

Example:

```
bool married;  -->  bool_t married;
```

Strings

C has no built-in string type, but instead uses the null-terminated 'char *' convention. In XDR language, strings are declared using the 'string' keyword, and compiled into 'char *'s in the output header file. The maximum size contained in the angle brackets specifies the maximum number of characters allowed in the strings (not counting the

NULL character). The maximum size may be left off, indicating a string of arbitrary length.

Examples:

```
string name<32>;    -->  char *name;
   string longname<>;  -->  char *longname;
```

Opaque Data

Opaque data is used in RPC and XDR to describe untyped data, that is, just sequences of arbitrary bytes. It may be declared either as a fixed or variable length array.

```
Examples:
      opaque diskblock[512];  -->  char diskblock[512];
   opaque filedata<1024>;  -->  struct {
                                   u_int filedata_len;
                                   char *filedata_val;
                                } filedata;
```

Voids

In a void declaration, the variable is not named. The declaration is just 'void' and nothing else. Void declarations can only occur in two places: union definitions and program definitions (as the argument or result of a remote procedure).

Chapter 4 – External Data Representation protocol specification

1 Introduction

This chapter contains technical notes on Sun's implementation of the eXternal Data Representation (XDR) standard, a set of library routines that allow a C programmer to describe arbitrary data structures in a machine-independent fashion. For a formal specification of the XDR standard, see the *eXternal Data Representation Standard.* XDR 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 contains a short tutorial overview of the XDR library routines, a guide to accessing currently available XDR streams, and information on defining new streams and data types. XDR was designed to work across different languages, operating systems, and machine architectures. Most users (particularly RPC users) will need only the information in sections 1, 2 and 3 of this document. Programmers wishing to implement RPC and XDR on new machines will need the information in the rest of this document, and especially the *eXternal Data Representation Standard.* NOTE: `rpcgen` can be used to write XDR routines in cases where no RPC calls are being made.

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
```

1.1 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 behaviour 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 4.2BSD 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`:

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

NOTE: Integers are 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. And pointers, which are very convenient to use, have no meaning outside the machine where they are defined.

A canonical standard

XDR's approach to standardising data representations is *canonical.* That is, XDR defines a single byte order (Big Endian), a single floating-point representation (IEEE), and so on. Any program running on any machine can use XDR to create portable data by translating its local representation to the XDR standard representations; similarly, any program running on any machine can read portable data by translating the XDR standard representaions to its local equivalents. The single standard completely decouples programs that create or send portable data from those that use or receive portable data. The advent of a new machine or a new language has no effect on the community of existing portable data creators and users. A new machine joins this community be being 'taught' how to convert the standard representations and its local representations; the local representations of other machines are irrelevant. Conversely, to existing programs running on other machines, the local representations of the new machine are also irrelevant; such programs can immediately read portable data produced by the new machine because such data conforms to the canonical standards that they already understand.

There are strong precedents for XDR's canonical approach. For example, TCP/IP, UDP/IP, XNS, Ethernet, and, indeed, all protocols below layer five of the ISO model, are canonical protocols. The advantage of any canonical approach is simplicity; in the case of XDR, a single set of conversion routines is written once and is never touched again. The canonical approach has a disadvantage, but it is unimportant in real-world data transfer applications. Suppose two Little-Endian machines are transferring integers according to the XDR standard. The sending machine converts the integers from Little-Endian byte order to XDR (Big-Endian) byte order; the receiving machine performs the reverse conversion. Because both machines observe the same byte order, their conversions are unnecessary. The point, however, is not necessity, but cost as compared to the alternative.

The time spent converting to and from a canonical representation is insignificant, especially in networking applications. Most of the time required to prepare a data structure for transfer is not spent in conversion but in traversing the elements of the data structure. To transmit a tree, for example, each leaf must be visited and each element in a leaf record must be copied to a buffer and aligned there; storage for

the leaf may have to be deallocated as well. Similarly, to receive a tree, storage must be allocated for each leaf, data must be moved from the buffer to the leaf and properly aligned, and pointers must be constructed to link the leaves together. Every machine pays the cost of traversing and copying data structures whether or not conversion is required. In networking applications, communications overhead – the time required to move the data down through the sender's protocol layers, across the network and up through the receiver's protocol layers – dwarfs conversion overhead.

1.2 The XDR library

The XDR library not only solves data portability problems, it also allows you to write and read arbitrary C constructs in a consistent, specified, well-documented manner. Thus, it can make 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 deserialising in the `reader` program.

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

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, xp)
        XDR *xdrs;
        xxx *xp;
{
}
```

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 serialise or deserialise 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 serialised data can also be deserialised. 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, the user can obtain the direction of the XDR operation. See the *XDR Operation Directions* section of this chapter 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 used for generic enums */
```

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.

2 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>.

2.1 Number filters

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

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

Specifically, the eight primitives are:

```
bool_t xdr_char(xdrs, cp)
        XDR *xdrs;
        char *cp;

bool_t xdr_u_char(xdrs, ucp)
        XDR *xdrs;
        unsigned char *ucp;

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.

2.2 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.

2.3 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 `TRUE (1)` or `FALSE (0)`.

```
#define bool_int
#define FALSE0
#define TRUE1

#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 routine returns *FALSE* if the number of characters exceeds `maxlength` and *TRUE* if it doesn't.

2.4 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 */
```

2.5 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 deserialising 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`.

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 behaviour of `xdr_string` is similar to the behaviour 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 the string does not exceed *maxlength* the bytes are serialised.

The effect of deserialising 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.

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,

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 deserialising.

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, elementsiz, xdr_element)
    XDR *xdrs;
    char **ap;
    u_int *lp;
    u_int maxlength;
```

```
    u_int elementsiz;
    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 deserialised, 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 serialised; `*lp` is set to the array length when the array is deserialised. The parameter `maxlength` is the maximum number of elements that the array is allowed to have; `elementsiz` 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 serialise, deserialise, 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 the `gethostname` man page; (b) the user's UID: see the `geteuid` man page; and (c) the group numbers to which the user belongs: see the `getgroups` man page. 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 < 256 chars */
#define NGRPS 20    /* user can't be in > 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:

```
truct 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 cannot be > 1000 chars */
#define NARGC 100    /* commands cannot have > 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.

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.

Fixed size arrays

The XDR library provides a primitive, *xdr_vector* for fixed-length arrays.

```
#define NLEN 255    /* machine names must be < 256 chars */
#define NGRPS 20    /* user belongs to exactly 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);
    if (!xdr_vector(xdrs, nup->nu_gids, NGRPS, sizeof(int),
        xdr_int)) {
            return(FALSE);
    }
    return(TRUE);
}
```

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;   /* the union's discriminant */
    union {
        int ival;
        char *pval;
        struct gnumbers gn;
    } uval;
};
```

The following constructs and XDR procedure (de)serialise 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 above in the *The XDR Library* section. `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 may legally 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.

Pointers

In C it is often convenient to put pointers to another structure within a structure. The primitive `xdr_reference` makes it easy to serialise, deserialise, 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.

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);
}
```

Pointers 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 the UNIX system, a core dump.

`xdr_pointer` correctly handles *NULL* pointers. For more information about its use, see *Linked Lists*.

2.6 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.

2.7 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.

3 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.

3.1 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 part of rpc */

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.

3.2 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.

3.3 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 BSD connection interface.

```
#include <rpc/rpc.h>    /* xdr streams part of rpc */
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 behaviour 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 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.

4 XDR implementation

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

4.1 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 added param */
    struct xdr_ops {
        bool_t  (*x_getlong)(); /* get long from stream */
        bool_t  (*x_putlong)(); /* put long to stream */
        bool_t  (*x_getbytes)(); /* get bytes from stream */
        bool_t  (*x_putbytes)(); /* put bytes to stream */
        u_int   (*x_getpostn)(); /* return stream offset */
        bool_t  (*x_setpostn)(); /* reposition offset */
        caddr_t (*x_inline)();   /* ptr to buffered data */
        VOID    (*x_destroy)();  /* free private area */
    } *x_ops;
    caddr_t     x_public;       /* users' data */
    caddr_t     x_private;      /* pointer to private data */
    caddr_t     x_base;         /* private 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 the previous section. 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 – those who wish to obtain maximum throughput. 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.

```
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. 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.

5 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. The *eXternal Data Representation Standard* chapter of this *Networking Programming* manual describes this language in complete detail.

5.1 Linked lists

The last example in the *Pointers* section presented a C data structure and its associated XDR routines for a individual'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:

```
struct gnumbers_node {
        struct gnumbers gn_numbers;
        struct gnumbers_node *gn_next;
};
typedef struct gnumbers_node *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 `gn_next` field is used to indicate whether or not the object has terminated. Unfortunately, if the object continues, the `gn_next` field is also the address of where it continues. The link addresses carry no useful information when the object is serialised. LP The XDR data description of this linked list is described by the recursive declaration of `gnumbers_list`:

```
Bstruct gnumbers {
        int g_assets;
        int g_liabilities;
};
struct gnumbers_node {
        gnumbers gn_numbers;
        gnumbers_list gn_next;
};
union gnumbers_list switch (bool more_data) {
case TRUE:
        gnumbers_node node;
case FALSE:
```

```
        void;
};
```

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`. Note that the C declaration has no boolean explicitly declared in it (though the `gn_next` field implicitly carries the information), while the XDR data description has no pointer explicitly declared in it.

Hints for writing the XDR routines for a `gnumbers_list` follow easily from the XDR description above. Note how the primitive `xdr_pointer` is used to implement the XDR union above.

```
bool_t
xdr_gnumbers_node(xdrs, gn)
        XDR *xdrs;
        gnumbers_node *gn;
{
        return(xdr_gnumbers(xdrs, &gn->gn_numbers) &&
                xdr_gnumbers_list(xdrs, &gp->gn_next));
}
bool_t
xdr_gnumbers_list(xdrs, gnp)
        XDR *xdrs;
        gnumbers_list *gnp;
{
        return(xdr_pointer(xdrs, gnp,
                sizeof(struct gnumbers_node),
                xdr_gnumbers_node));
}
```

The unfortunate side effect of XDR'ing a list with these routines is that the C stack grows linearly with respect to the number of node in the list. This is due to the recursion. The following routine collapses the above two mutually recursive into a single, non-recursive one.

```
bool_t
xdr_gnumbers_list(xdrs, gnp)
        XDR *xdrs;
        gnumbers_list *gnp;
{
        bool_t more_data;
        gnumbers_list *nextp;
        for (;;) {
                more_data = (*gnp != NULL);
                if (!xdr_bool(xdrs, &more_data)) {
                        return(FALSE);
                }
                if (! more_data) {
                        break;
                }
                if (xdrs->x_op == XDR_FREE) {
                        nextp = &(*gnp)->gn_next;
                }
                if (!xdr_reference(xdrs, gnp,
                        sizeof(struct gnumbers_node), xdr_gnumbers)) {

                return(FALSE);
                }
                gnp = (xdrs->x_op == XDR_FREE) ?
                        nextp : &(*gnp)->gn_next;
        }
        *gnp = NULL;
        return(TRUE);
}
```

The first task is to find out whether there is more data or not, so that this boolean information can be serialised. Notice that this statement is unnecessary in the

XDR_DECODE case, since the value of more_data is not known until we deserialise it in the next statement.

The next statement XDR's the more_data field of the XDR union. Then if there is truly no more data, we set this last pointer to *NULL* to indicate the end of the list, and return *TRUE* because we are done. Note that setting the pointer to *NULL* is only important in the XDR_DECODE case, since it is already *NULL* in the XDR_ENCODE and XDR_FREE cases.

Next, if the direction is XDR_FREE the value of nextp is set to indicate the location of the next pointer in the list. We do this now because we need to dereference gnp to find the location of the next item in the list, and after the next statement the pointer *gnp* will be freed up and no longer valid. We can't do this for all directions though, because in the XDR_DECODE direction the value of gnp won't be set until the next statement.

Next, we XDR the data in the node using the primitive xdr_reference xdr_reference is like xdr_pointer which we used before, but it does not send over the boolean indicating whether there is more data. We use it instead of xdr_pointer because we have already XDR'd this information ourselves. Notice that the XDR routine passed is not the same type as an element in the list. The routine passed is xdr_gnumbers for XDR'ing gnumbers, but each element in the list is actually of type gnumbers_node. We don't pass xdr_gnumbers_node because it is recursive, and instead use xdr_gnumbers which XDR's all of the non-recursive part. Note that this trick will work only if the gn_numbers field is the first item in each element, so that their addresses are identical when passed to xdr_reference.

Finally, we update gnp to point to the next item in the list. If the direction is XDR_FREE we set it to the previously saved value, otherwise we can dereference gnp to get the proper value. Though harder to understand than the recursive version, this non-recursive routine will never cause the C stack to blow up. It will also run more efficiently since a lot of procedure call overhead has been removed. Most lists are small though (in the hundreds of items or less) and the recursive version should be sufficient for them.

Chapter 5 – External Data Representation standard

1 XDR standard

5.1 Introduction

This chapter specifies a protocol that Sun Microsystems, Inc., and others are using. It has been designated RFC1014 by the ARPA Network Information Center.

XDR is a standard for the description and encoding of data. It is useful for transferring data between different computer architectures, and has been used to communicate data between such diverse machines as the Sun Workstation, VAX, IBM-PC, and Cray. XDR fits into the ISO presentation layer, and is roughly analogous in purpose to X.409, ISO Abstract Syntax Notation. The major difference between these two is that XDR uses implicit typing, while X.409 uses explicit typing.

XDR uses a language to describe data formats. The language can only be used only to describe data; it is not a programming language. This language allows one to describe intricate data formats in a concise manner. The alternative of using graphical representations (itself an informal language) quickly becomes incomprehensible when faced with complexity. The XDR language itself is similar to the C language [1], just as Courier [4] is similar to Mesa. Protocols such as Sun RPC (Remote Procedure Call) and the NFS (Network File System) use XDR to describe the format of their data.

The XDR standard makes the following assumption: that bytes (or octets) are portable, where a byte is defined to be eight bits of data. A given hardware device should encode the bytes onto the various media in such a way that other hardware devices may decode the bytes without loss of meaning. For example, the Ethernet standard suggests that bytes be encoded in 'little-endian' style [2], or least significant bit first.

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. The bytes are read or written to some byte stream such that byte m always precedes byte m+1. If the n bytes needed to contain the data are not a multiple of four, then the n bytes are followed by enough (0 to 3) residual zero bytes, r, to make the total byte count a multiple of 4.

We include the familiar graphic box notation for illustration and comparison. In most illustrations, each box (delimited by a plus sign at the four corners and vertical bars and dashes) depicts a byte. Ellipsis (...) between boxes show zero or more additional bytes

```
        A Block
+--------+--------+...+--------+--------+... +--------+
|byte 0  |byte 1  |...|byte n-1|  0     |... +--------+
|<-----------n bytes---------->|<------r bytes------->|
|<-----------n+r (where (n+r) mod 4 = 0)>------------>|
```

Each of the sections that follow describes a data type defined in the XDR standard, shows how it is declared in the language, and includes a graphic illustration of its encoding.

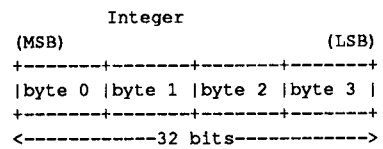
5.2 XDR data types

For each data type in the language we show a general paradigm declaration. Note that angle brackets (< and >) denote variable length sequences of data and square brackets ([and]) denote fixed-length sequences of data. 'n', 'm' and 'r' denote integers. For the full language specification and more formal definitions of terms such as 'identifier' and 'declaration', refer to *The XDR Language Specification* below.

For some data types, more specific examples are included. A more extensive example of a data description is in *An Example of an XDR Data Description* below.

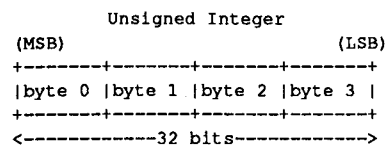
Integer

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. Integers are declared as follows:



Unsigned integer

An XDR unsigned integer is a 32-bit datum that encodes a non-negative 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. An unsigned integer is declared as follows:



Enumeration

Enumerations have the same representation as signed integers. Enumerations are handy for describing subsets of the integers. Enumerated data is declared as follows:

```
enum { name-identifier = constant, ... } identifier;
```

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

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

It is an error to encode as an enum any other integer than those that have been given assignments in the enum declaration.

Boolean

Booleans are important enough and occur frequently enough to warrant their own explicit type in the standard. Booleans are declared as follows:

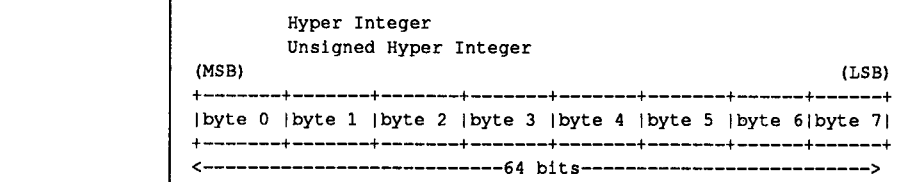
```
bool identifier;
```

This is equivalent to:

```
enum { FALSE = 0, TRUE = 1 } identifier;
```

Hyper integer and unsigned hyper integer

The standard also defines 64-bit (8-byte) numbers called hyper integer and unsigned hyper integer. Their representations are the obvious extensions of integer and unsigned integer defined above. They are represented in two's complement notation. The most and least significant bytes are 0 and 7, respectively. Their declarations:



Floating point

The standard defines the floating-point data type 'float' (32 bits or 4 bytes). The encoding used is the IEEE standard for normalized single-precision floating-point numbers [3]. The following three fields describe the single-precision floating-point number:

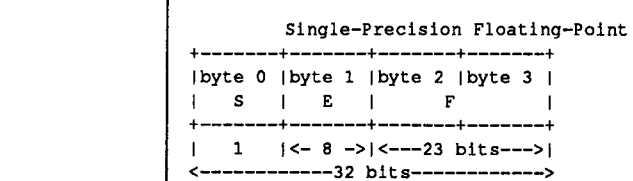
S: The sign of the number. Values 0 and 1 represent positive and negative, respectively. One bit.

E: The exponent of the number, base 2. Eight bits are devoted to this field. The exponent is biased by 127.

F: The fractional part of the number's mantissa, base 2. 23 bits are devoted to this field.

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. Note that these numbers refer to the mathematical positions of the bits, and NOT to their actual physical locations (which vary from medium to medium).

The IEEE specifications should be consulted concerning the encoding for signed zero, signed infinity (overflow), and denormalized numbers (underflow) [3]. According to IEEE specifications, the 'NaN' (not a number) is system dependent and should not be used externally.

Double precision floating point

The standard defines the encoding for the double-precision floating- point data type 'double' (64 bits or eight bytes). The encoding used is the IEEE standard for normalised double-precision floating-point numbers [3]. The standard encodes the following three fields, which describe the double-precision floating-point number:

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

E: The exponent of the number, base 2. 11 bits are devoted to this field. The exponent is biased by 1023.

F: The fractional part of the number's mantissa, base 2. 52 bits are devoted to this field.

Therefore, the floating-point number is described by:

(-1)**S * 2**(E-Bias) * 1.F

It is declared as follows:

```
        Double-Precision Floating-Point
+------+------+------+------+------+------+------+------+
|byte 0|byte 1|byte 2|byte 3|byte 4|byte 5|byte 6|byte 7|
    S  |        E    |           F                       |
+------+------+------+------+------+------+------+------+
   1   |<----11----->|<------------52 bits------------->|
<----------------------64 bits-------------------------->
```

Just as the most and least significant bytes of a number are 0 and 3, the most and least significant bits of a double-precision floating-point number are 0 and 63. The beginning bit (and most significant bit) offsets of S, E , and F are 0, 1, and 12, respectively. Note that these numbers refer to the mathematical positions of the bits, and NOT to their actual physical locations (which vary from medium to medium).

The IEEE specifications should be consulted concerning the encoding for signed zero, signed infinity (overflow), and denormalised numbers (underflow) [3]. According to IEEE specifications, the 'NaN' (not a number) is system dependent and should not be used externally.

Fixed-length opaque data

At times, fixed-length uninterpreted data needs to be passed among machines. This data is called 'opaque' and is declared as follows:

```
opaque identifier[n];
```

where the constant 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 (0 to 3) residual zero bytes, r, to make the total byte count of the opaque object a multiple of four.

```
Fixed-Length Opaque
   0        1      ...

+--------+--------+...+--------+--------+...+--------+
| byte 0 | byte 1 |...|byte n-1|   0    |...|   0    |
+--------+--------+...+--------+--------+...+--------+
|<-----------n bytes---------->|<------r bytes------>|
|<-----------n+r (where (n+r) mod 4 = 0)------------>|
```

Variable-length opaque data

The standard also provides for variable-length (counted) opaque data, defined as a sequence of n (numbered 0 through n-1) arbitrary bytes to be the number n encoded as an unsigned integer (as described below), and followed by the n bytes of the sequence.

Byte m of the sequence always precedes byte m+1 of the sequence, and byte 0 of the sequence always follows the sequence's length (count). Enough (0 to 3) residual zero bytes, r, to make the total byte count a multiple of four. Variable-length opaque data is declared in the following way:

```
opaque identifier<m>;
```

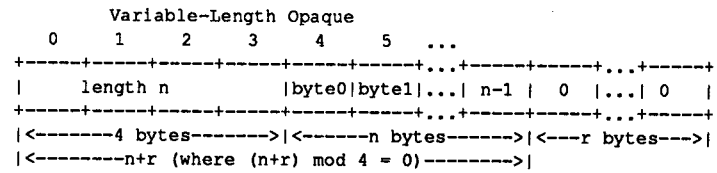
or

```
opaque identifier<>;
```

The constant m denotes an upper bound of the number of bytes that the sequence may contain. If m is not specified, as in the second declaration, it is assumed to be (2**32) –1, the maximum length. The constant m would normally be found in a protocol specification. For example, a filing protocol may state that the maximum data transfer size is 8192 bytes, as follows:

```
opaque filedata<8192>;
```

This can be illustrated as follows:



It is an error to encode a length greater than the maximum described in the specification.

String

The standard defines a string of n (numbered 0 through n-1) ASCII bytes to be the number n encoded as an unsigned integer (as described above), and followed by the n bytes of the string. Byte m of the string always precedes byte m+1 of the string, and byte 0 of the string always follows the string's length. If n is not a multiple of four, then the n bytes are followed by enough (0 to 3) residual zero bytes, r, to make the total byte count a multiple of four. Counted byte strings are declared as follows:

```
string object<m>;
```

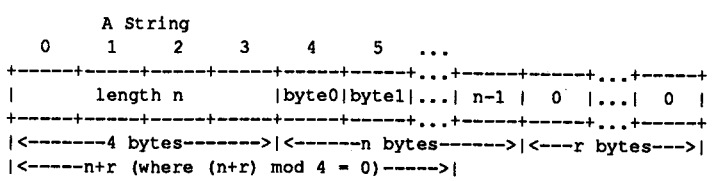
or

```
string object<>;
```

The constant m denotes an upper bound of the number of bytes that a string may contain. If m is not specified, as in the second declaration, it is assumed to be (2**32) –1, the maximum length. The constant m 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, as follows:

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

Which can be illustrated as:



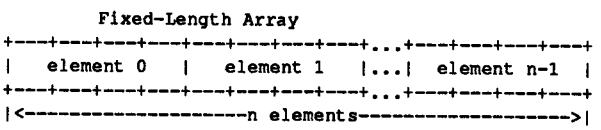
It is an error to encode a length greater than the maximum described in the specification.

Fixed-length array

Declarations for fixed-length arrays of homogeneous elements are in the following form:

```
type-name identifier[n];
```

Fixed-length 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. Each element's size is a multiple of four bytes. Though all elements are of the same type, the elements may have different sizes. For example, in a fixed-length array of strings, all elements are of type 'string', yet each element will vary in its length.



Variable-length array

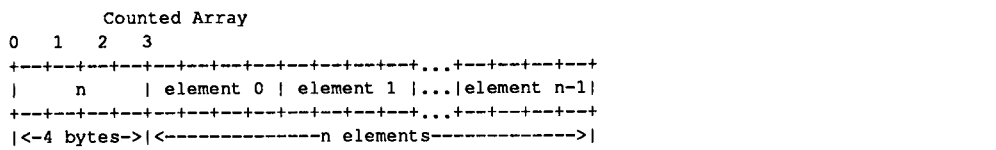
Counted arrays provide the ability to encode variable-length arrays of homogeneous elements. The array is encoded as the element count n (an unsigned integer) followed by the encoding of each of the array's elements, starting with element 0 and progressing through element n- 1. The declaration for variable-length arrays follows this form:

```
type-name identifier<m>;
```

or

```
type-name identifier<>;
```

The constant m specifies the maximum acceptable element count of an array; if m is not specified, as in the second declaration, it is assumed to be (2**32) –1.



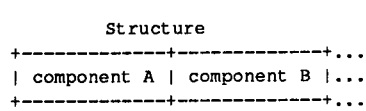
It is an error to encode a value of n that is greater than the maximum described in the specification.

Structure

Structures are declared as follows:

```
struct {
        component-declaration-A;
        component-declaration-B;
        ...
} identifier;
```

The components of the structure are encoded in the order of their declaration in the structure. Each component's size is a multiple of four bytes, though the components may be different sizes.



Discriminated union

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 discriminant is either `int`, `unsigned int`, or an enumerated type, such as `bool`. The component types are called 'arms' of the union, and are preceded by the value of the discriminant which implies their encoding. Discriminated unions are declared as follows:

```
union switch (discriminant-declaration) {
        case discriminant-value-A:
        arm-declaration-A;
        case discriminant-value-B:
        arm-declaration-B;
        ...
        default: default-declaration;
} identifier;
```

Each 'case' keyword is followed by a legal value of the discriminant. The default arm is optional. If it is not specified, then a valid encoding of the union cannot take on unspecified discriminant values. The size of the implied arm is always a multiple of four bytes.

The discriminated union is encoded as its discriminant followed by the encoding of the implied arm.

```
        Discriminated Union
  0   1   2   3
+---+---+---+---+---+---+---+---+
| discriminant  |implied arm    |
+---+---+---+---+---+---+---+---+
|<---4 bytes--->|
```

Void

An XDR void is a 0-byte quantity. Voids are useful for describing operations that take no data as input or no data as output. They are also useful in unions, where some arms may contain data and others do not. The declaration is simply as follows:

```
void;
```

Voids are illustrated as follows:

```
        Void
  ++
  ||
  ++
--><-- 0 bytes
```

Constant

The data declaration for a constant follows this form:

```
const name-identifier = n;
```

'const' is used to define a symbolic name for a constant; it does not declare any data. The symbolic constant may be used anywhere a regular constant may be used. For example, the following defines a symbolic constant DOZEN, equal to 12.

```
const DOZEN = 12;
```

Typedef

'typedef' does not declare any data either, but serves to define new identifiers for declaring data. The syntax is:

```
typedef declaration;
```

The new type name is actually the variable name in the declaration part of the typedef. For example, the following defines a new type called 'eggbox' using an existing type called 'egg':

```
typedef egg eggbox[DOZEN];
```

Variables declared using the new type name have the same type as the new type name would have in the typedef, if it was considered a variable. For example, the following two declarations are equivalent in declaring the variable 'fresheggs':

```
eggbox  fresheggs;
egg     fresheggs[DOZEN];
```

When a typedef involves a struct, enum, or union definition, there is another (preferred) syntax that may be used to define the same type. In general, a typedef of the following form:

```
typedef <<struct, union, or enum definition>> identifier;
```

may be converted to the alternative form by removing the 'typedef' part and placing the identifier after the 'struct', 'union', or 'enum' keyword, instead of at the end. For example, here are the two ways to define the type 'bool':

```
typedef enum {              /* using typedef */
        FALSE = 0,
        TRUE = 1
        } bool;

        enum bool {         /* preferred alternative */
        FALSE = 0,
        TRUE = 1
        };
```

The reason this syntax is preferred is one does not have to wait until the end of a declaration to figure out the name of the new type.

Optional-data

Optional-data is one kind of union that occurs so frequently that we give it a special syntax of its own for declaring it. It is declared as follows:

```
type-name *identifier;
```

This is equivalent to the following union:

```
union switch (bool opted) {
        case TRUE:
        type-name element;
        case FALSE:
        void;
} identifier;
```

It is also equivalent to the following variable-length array declaration, since the boolean 'opted' can be interpreted as the length of the array:

```
type-name identifier<1>;
```

Optional-data is not so interesting in itself, but it is very useful for describing recursive data-structures such as linked-lists and trees. For example, the following defines a type 'stringlist' that encodes lists of arbitrary length strings:

```
struct *stringlist {
        string item<>;
        stringlist next;
};
```

It could have been equivalently declared as the following union:

```
union stringlist switch (bool opted) {
        case TRUE:
            struct {
                string item<>;
                stringlist next;
            } element;
        case FALSE:
            void;
};
```

or as a variable-length array:

```
struct stringlist<1> {
        string item<>;
        stringlist next;
};
```

Both of these declarations obscure the intention of the stringlist type, so the optional-data declaration is preferred over both of them. The optional-data type also has a close correlation to how recursive data structures are represented in high-level languages such as Pascal or C by use of pointers. In fact, the syntax is the same as that of the C language for pointers.

Areas for future enhancement

The XDR standard lacks representations for bit fields and bitmaps, since the standard is based on bytes. Also missing are packed (or binary-coded) decimals.

The intent of the XDR standard was not to describe every kind of data that people have ever sent or will ever want to send from machine to machine. Rather, it only describes the most commonly used data-types of high-level languages such as Pascal or C so that applications written in these languages will be able to communicate easily over some medium.

One could imagine extensions to XDR that would let it describe almost any existing protocol, such as TCP. The minimum necessary for this are support for different block sizes and byte-orders. The XDR discussed here could then be considered the 4-byte big-endian member of a larger XDR family.

5.3 Discussion

Why a language for describing data?

There are many advantages in using a data-description language such as XDR versus using diagrams. Languages are more formal than diagrams and lead to less ambiguous descriptions of data. Languages are also easier to understand and allow one to think of other issues instead of the low-level details of bit-encoding. Also, there is a close analogy between the types of XDR and a high-level language such as C or Pascal. This makes the implementation of XDR encoding and decoding modules an easier task. Finally, the language specification itself is an ASCII string that can be passed from machine to machine to perform on-the-fly data interpretation.

Why only one byte-order for an XDR unit?

Supporting two byte-orderings requires a higher level protocol for determining in which byte-order the data is encoded. Since XDR is not a protocol, this can't be done. The advantage of this, though, is that data in XDR format can be written to a magnetic tape, for example, and any machine will be able to interpret it, since no higher level protocol is necessary for determining the byte-order.

Why does XDR use big-endian byte -order?

Yes, it is unfair, but having only one byte-order means you have to be unfair to somebody. Many architectures, such as the Motorola 68000 and IBM 370, support the big-endian byte-order.

Why is the XDR unit four bytes wide?

There is a tradeoff in choosing the XDR unit size. Choosing a small size such as two makes the encoded data small, but causes alignment problems for machines that aren't aligned on these boundaries. A large size such as eight means the data will be aligned on virtually every machine, but causes the encoded data to grow too big. Four was chosen as a compromise. Four is big enough to support most architectures efficiently, except for rare machines such as the eight-byte aligned Cray. Four is also small enough to keep the encoded data restricted to a reasonable size.

Why must variable-length data be padded with zeros?

It is desirable that the same data encode into the same thing on all machines, so that encoded data can be meaningfully compared or checksummed. Forcing the padded bytes to be zero ensures this.

Why is there no explicit data-typing?

Data-typing has a relatively high cost for what small advantages it may have. One cost is the expansion of data due to the inserted type fields. Another is the added cost of interpreting these type fields and acting accordingly. And most protocols already know what type they expect, so data-typing supplies only redundant information. However, one can still get the benefits of data-typing using XDR. One way is to encode two things: first a string which is the XDR data description of the encoded data, and then the encoded data itself. Another way is to assign a value to all the types in XDR, and then define a universal type which takes this value as its discriminant and for each value, describes the corresponding data type.

5.4 The XDR language specification

This specification uses an extended Backus-Naur Form notation for describing the XDR language.

Notational conventions

Here is a brief description of the notation:

- The characters | ( ) [ ] " , and * are special.
- Terminal symbols are strings of any characters surrounded by double quotes.
- Non-terminal symbols are strings of non-special characters.
- Alternative items are separated by a vertical bar ('|').
- Optional items are enclosed in brackets.
- Items are grouped together by enclosing them in parentheses.
- A * following an item means 0 or more occurrences of that item.

For example, consider the following pattern:

"a " "very" (", " " very")* [" cold " "and"] " rainy " ("day" | "night")

An infinite number of strings match this pattern. A few of them are:

"a very rainy day"
"a very, very rainy day"
"a very cold and rainy day"
"a very, very, very cold and rainy night"

Lexical notes

- Comments begin with '/*' and terminate with '*/'.
- White space serves to separate items and is otherwise ignored.
- An identifier is a letter followed by an optional sequence of letters, digits or underbar ('_'). The case of identifiers is not ignored.
- A constant is a sequence of one or more decimal digits, optionally preceded by a minus-sign ('-').

Syntax information

```
declaration:
        type-specifier identifier
        | type-specifier identifier "[" value "]"
        | type-specifier identifier "<" [ value ] ">"
        | "opaque" identifier "[" value "]"
        | "opaque" identifier "<" [ value ] ">"
        | "string" identifier "<" [ value ] ">"
        | type-specifier "*" identifier
        | "void"

value:
        constant
        | identifier

type-specifier:
          [ "unsigned" ] "int"
        | [ "unsigned" ] "hyper"
        | "float"
        | "double"
        | "bool"
        | enum-type-spec
        | struct-type-spec
        | union-type-spec
        | identifier

enum-type-spec:
        "enum" enum-body

enum-body:
        "{"
        ( identifier "=" value )
        ( "," identifier "=" value )*
        "}"
```

```
struct-type-spec:
        "struct" struct-body

struct-body:
        "{"
        ( declaration ";" )
        ( declaration ";" )*
        "}"

union-type-spec:
        "union" union-body

union-body:
        "switch" "(" declaration ")" "{"
        ( "case" value ":" declaration ";" )
        ( "case" value ":" declaration ";" )*
        [ "default" ":" declaration ";" ]
        "}"

constant-def:
        "const" identifier "=" constant ";"

type-def:
        "typedef" declaration ";"
        | "enum" identifier enum-body ";"
        | "struct" identifier struct-body ";"
        | "union" identifier union-body ";"

definition:
        type-def
        | constant-def

specification:
        definition *
```

Syntax notes

1. The following are keywords and cannot be used as identifiers: `"bool"`, `"case"`, `"const"`, `"default"`, `"double"`, `"enum"`, `"float"`, `"hyper"`, `"opaque"`, `"string"`, `"struct"`, `"switch"`, `"typedef"`, `"union"`, `"unsigned"` and `"void"`.

2. Only unsigned constants may be used as size specifications for arrays. If an identifier is used, it must have been declared previously as an unsigned constant in a 'const' definition.

3. Constant and type identifiers within the scope of a specification are in the same name space and must be declared uniquely within this scope.

4. Similarly, variable names must be unique within the scope of struct and union declarations. Nested struct and union declarations create new scopes.

5. The discriminant of a union must be of a type that evaluates to an integer. That is, 'int', 'unsigned int', 'bool', an enumerated type or any typedefed type that evaluates to one of these is legal. Also, the case values must be one of the legal values of the discriminant. Finally, a case value may not be specified more than once within the scope of a union declaration.

An example of an XDR data description

Here is a short XDR data description of a thing called a 'file', which might be used to transfer files from one machine to another.

```
const MAXUSERNAME = 32;     /* max length of a user name */
const MAXFILELEN = 65535;   /* max length of a file      */
const MAXNAMELEN = 255;     /* max length of a file name */
/*
 * Types of files:
 */

enum filekind {
        TEXT = 0,       /* ascii data */
        DATA = 1,       /* raw data   */
        EXEC = 2        /* executable */
};

/*
 * File information, per kind of file:
 */

union filetype switch (filekind kind) {
        case TEXT:
                void;               /* no extra information */
        case DATA:
                string creator<MAXNAMELEN>;
                                    /* data creator         */
        case EXEC:
                string interpretor<MAXNAMELEN>;
                                    /* program interpretor  */
};

/*
 * A complete file:
 */

struct file {
        string filename<MAXNAMELEN>; /* name of file */
        filetype type;               /* info about file */
        string owner<MAXUSERNAME>;   /* owner of file   */
        opaque data<MAXFILELEN>;     /* file data       */
};
```

Suppose now that there is a user named 'john' who wants to store his lisp program 'sillyprog' that contains just the data '(quit)'. His file would be encoded as follows:

Offset	Hex Bytes	ASCII	Description
0	00 00 00 09	....	Length of filename = 9
4	73 69 6c 6c	sill	Filename characters
8	79 70 72 6f	ypro	...and more characters ...
12	67 00 00 00	g...	... and 3 zero-bytes of fill
16	00 00 00 02	....	Filekind is EXEC = 2
20	00 00 00 04	....	Length of interpretor = 4
24	6c 69 73 70	lisp	Interpretor characters
28	00 00 00 04	....	Length of owner = 4
32	6a 6f 68 6e	john	Owner characters
36	00 00 00 06	....	Length of file data = 6
40	28 71 75 69	(qui	File data bytes ...
44	74 29 00 00	t)..	... and 2 zero-bytes of fill

References

[1] Brian W. Kernighan & Dennis M. Ritchie, '*The C Programming Language*', Bell Laboratories, Murray Hill, New Jersey, 1978.

[2] Danny Cohen, '*On Holy Wars and a Plea for Peace*', IEEE Computer, October 1981.

[3] '*IEEE Standard for Binary Floating-Point Arithmetic*', ANSI/IEEE Standard 754-1985, Institute of Electrical and Electronics Engineers, August 1985.

[4] '*Courier: The Remote Procedure Call Protocol*', XEROX Corporation, XSIS 038112, December 1981.

Chapter 6 – Remote Procedure Calls protocol specification

1 Introduction

This chapter specifies a protocol that Sun Microsystems, Inc., and others are using. It has been submitted to the ARPA-Internet for consideration as an RFC. Certain details may change as a result of comments made during the review of this draft standard.

This chapter 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. See the *eXternal Data Representation Standard* for the details. Here, we assume that the reader is familiar with XDR and do not attempt to justify RPC or its uses). The paper by Birrell and Nelson [1] is recommended as an excellent background to and justification of RPC.

1.1 Terminology

This chapter discusses servers, services, programs, procedures, clients, and versions. A server is a piece of software where network services are implemented. A network 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 the *Port Mapper Program Protocol*, below, 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 IO 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, in that one thread of control logically 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. However, this model is only given as an example. The RPC protocol makes no restrictions on the concurrency model implemented, and others are possible. For example, an implementation may choose to have RPC calls be asynchronous, so that the client may do useful work while waiting for the reply from the server. Another possibility is to have the server create a task to process an incoming request, so that the server can be free to receive other requests.

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 deals only with specification and interpretation of messages.

It is important to point out that RPC does not try to implement any kind of reliability and that the application must be aware of the type of transport protocol underneath RPC. If it knows it is running on top of a reliable transport such as TCP/IP[6], then most of the work is already done for it. On the other hand, if it is running on top of an unreliable transport such as UDP/IP[7], it must implement is own retransmission and time-out policy as the RPC layer does not provide this service.

Because of transport independence, the RPC protocol does not attach specific semantics to the remote procedures or their execution. Semantics can be inferred from (but should be explicitly specified by) the underlying transport protocol. For example, consider RPC running on top of an unreliable transport such as UDP/IP. If an application retransmits RPC messages after short time-outs, the only thing it can infer if it receives no reply is that the procedure was executed zero or more times. If it does receive a reply, then it can infer that the procedure was executed at least once.

A server may wish to remember previously granted requests from a client and not regrant them in order to ensure some degree of execute-at-most-once semantics. A server can do this by taking advantage of the transaction ID that is packaged with every RPC request. The main use of this transaction is by the client RPC layer in matching replies to requests. However, a client application may choose to reuse its previous transaction ID when retransmitting a request. The server application, knowing this fact, may choose to remember this ID after granting a request and not regrant requests with the same ID in order to achieve some degree of execute-at-most-once semantics. The server is not allowed to examine this ID in any other way except as a test for equality.

On the other hand, if using a reliable transport such as TCP/IP, the application can infer from a reply message that the procedure was executed exactly once, but if it receives no reply message, it cannot assume the remote procedure was not executed. Note that even if a connection-oriented protocol like TCP is used, an application still needs time-outs and reconnection to handle server crashes.

There are other possibilities for transports besides datagram- or connection-oriented protocols. For example, a request-reply protocol such as VMTP[2] is perhaps the most natural transport for RPC.

NOTE: RPC is currently implemented on top of both 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 the *Port Mapper Program Protocol*, below).

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 itself to a service and vice-versa. Security and access control mechanisms can be built on top of the message authentication. Several different authentication protocols can be supported. A field in the RPC header indicates which protocol is being used. More information on specific authentication protocols can be found in the *Authentication Protocols*, below.

2 RPC protocol requirments

The RPC protocol must provide for the following:

- Unique specification of a procedure to be called.
- Provisions for matching response messages to request messages.
- 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:

- RPC protocol mismatches.
- Remote program protocol version mismatches.
- Protocol errors (such as misspecification of a procedure's parameters).
- Reasons why remote authentication failed.
- Any other reasons why the desired procedure was not called.

2.1 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. 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 in it the RPC version number, which is always equal to two for the version of RPC described here.

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

- The remote implementation of RPC does not speak protocol version 2. The lowest and highest supported RPC version numbers are returned.
- The remote program is not available on the remote system.
- The remote program does not support the requested version number. The lowest and highest supported remote program version numbers are returned.

- The requested procedure number does not exist. (This is usually a caller side protocol or programming error.)
- The parameters to the remote procedure appear to be garbage from the server's point of view. (Again, this is usually 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 part 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 {
    auth_flavor flavor;
    opaque body<400>;
};
```

In simple English, any `opaque_auth` structure is an `auth_flavor` enumeration followed by bytes which 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. (See *Authentication Protocols*, below, for definitions of the various 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* according to the following chart:

Program numbers	Description
00000000 - 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 sites. The second range is for applications peculiar to a particular site. This range is intended primarily for debugging new programs. When a site 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 reserved for future use, and should not be used.

2.4 Other uses of 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, or perhaps 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.

Batching

Batching allows a client to send an arbitrarily large sequence of call messages to a server; batching typically uses reliable byte stream protocols (like TCP/IP) for its 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).

Broadcast RPC

In broadcast RPC-based protocols, the client sends a broadcast packet to the network and waits for numerous replies. Broadcast RPC uses unreliable, packet-based protocols (like UDP/IP) as its transports. Servers that support broadcast protocols only respond when the request is successfully processed, and are silent in the face of errors. Broadcast RPC uses the Port Mapper RPC service to achieve its semantics. See the *Port Mapper Program Protocol*, below, for more information.

3 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.

```
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,/* RPC executed successfully       */
        PROG_UNAVAIL  = 1,/* remote hasn't exported program  */
        PROG_MISMATCH = 2,/* remote can't support version #  */
        PROC_UNAVAIL  = 3,/* program can't support procedure */
        GARBAGE_ARGS  = 4,/* procedure can't decode params   */
};


/*
* Reasons why a call message was rejected:
*/
enum reject_stat {
        RPC_MISMATCH = 0,/* RPC version number != 2          */
        AUTH_ERROR = 1   /* remote can't authenticate caller */
};

/*
* Why authentication failed:
*/
enum auth_stat {
        AUTH_BADCRED    = 1,    /* bad credentials (seal  broken)*/
        AUTH_REJECTEDCRED = 2,  /* client must begin new session */
        AUTH_BADVERF  = 3,      /* bad verifier (seal broken)    */
        AUTH_REJECTEDVERF = 4,  /* verifier expired or replayed  */
        AUTH_TOOWEAK      = 5   /* rejected for 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  or for servers detecting  retransmissions; the
* service side  cannot treat this id  as any type   of sequence
* number.
*/
```

```
struct rpc_msg {
        unsigned int xid;
        union switch (msg_type mtype) {
                case CALL:
                        call_body cbody;
                        call_body cbody;
                case REPLY:
                        reply_body rbody;
        } 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 number, and the  procedure within
* the remote program to be called.  After these  fields are two
* authentication  parameters: cred (authentication credentials)
* and verf  (authentication verifier).  The  two authentication
* parameters are   followed by  the  parameters  to  the remote
* procedure,  which  are specified  by  the  specific   program
* protocol.
*/
struct call_body {
        unsigned int rpcvers;  /* must be equal to two (2) */
        unsigned int prog;
        unsigned int vers;
        unsigned int proc;
        opaque_auth cred;
        opaque_auth verf;
        /* procedure specific parameters start here */
};

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


/*
* Reply to   an RPC request  that  was accepted  by the server:
* there could be an error even though the request was accepted.
* The first field is an authentication verifier that 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_ARGP
* arms of the union are void.   The PROG_MISMATCH arm specifies
* the lowest and highest version numbers of the  remote program
* supported by the server.
*/
struct accepted_reply {
        opaque_auth verf;
        union switch (accept_stat stat) {
                case SUCCESS:
                        opaque results[0];
                        /* procedure-specific results start here */
                case PROG_MISMATCH:
                        struct {
                                unsigned int low;
                                unsigned int high;
                        } mismatch_info;
                default:
                        /*
                        * Void.  Cases include PROG_UNAVAIL, PROC_UNAVAIL,
                        * and GARBAGE_ARGS.
                        */
```

```
                        void;
        } reply_data;
};

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

3.1 Authentication protocols

As previously stated, authentication parameters are opaque, but open-ended to the rest of the RPC protocol. This section defines some 'flavours' of authentication implemented. Other sites are free to invent new authentication types, with the same rules of flavor number assignment as there is for program number assignment.

Null authentication

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

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 the credential's opaque body encode the the following structure:

```
struct auth_unix {
        unsigned int stamp;
        string machinename<255>;
        unsigned int uid;
        unsigned int gid;
        unsigned int 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 caller's 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 discriminant of the response verifier received in the reply message from the server may be AUTH_NULL or AUTH_SHORT. In the case of AUTH_SHORT the bytes of the response verifier's string encode an opaque structure. This new opaque structure may now be passed to the server instead of the original AUTH_UNIX flavor credentials. The server keeps a cache which maps shorthand opaque structures (passed back by way of an 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 shorthand 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.

3.2 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). This RM/TCP/IP transport is used 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!)

3.3 The RPC language

Just as there was a need to describe the XDR data-types in a formal language, there is also need to describe the procedures that operate on these XDR data-types in a formal language as well. We use the RPC Language for this purpose. It is an extension to the XDR language. The following example is used to describe the essence of the language. Here is an example of the specification of a simple ping program.

```
/*
* Simple ping program
*/
program PING_PROG {
        /* Latest and greatest version */
        version PING_VERS_PINGBACK {
        void
        PINGPROC_NULL(void) = 0;

        /*
        * Ping the caller, return the round-trip time
        * (in microseconds). Returns -1 if the operation
        * timed out.
        */
        int
        PINGPROC_PINGBACK(void) = 1;
} = 2;

/*
* Original version
*/
version PING_VERS_ORIG {
        void
        PINGPROC_NULL(void) = 0;
        } = 1;
} = 1;
const PING_VERS = 2;      /* latest version */
```

The first version described is PING_VERS_PINGBACK with two procedures, PINGPROC_NULL and PINGPROC_PINGBACK. PINGPROC_NULL takes no arguments and returns no results, but it is useful for computing round-trip times from the client to the server and back again. By convention, procedure 0 of any RPC protocol should have the same semantics, and never require any kind of authentication. The second procedure is used for the client to have the server do a reverse ping operation back to the client, and it returns the amount of time (in microseconds) that the operation used. The next version, PING_VERS_ORIG is the

original version of the protocol and it does not contain PINGPROC_PINGBACK procedure. It is useful for compatibility with old client programs, and as this program matures it may be dropped from the protocol entirely.

RPC language specification

The RPC language is identical to the XDR language, except for the added definition of a `program-def` described below.

```
program-def:
        "program" identifier "{"
                version-def
                version-def *
        "}" "=" constant ";"


version-def:
        "version" identifier "{"
                procedure-def
                procedure-def *
        "}" "=" constant ";"


procedure-def:
        type-specifier identifier "(" type-specifier ")"
        "=" constant ";"
```

Syntax notes

1. The following keywords are added and cannot be used as identifiers: 'program' and 'version';
2. A version name cannot occur more than once within the scope of a program definition. Nor can a version number occur more than once within the scope of a program definition.
3. A procedure name cannot occur more than once within the scope of a version definition. Nor can a procedure number occur more than once within the scope of version definition.
4. Program identifiers are in the same name space as constant and type identifiers.
5. Only unsigned constants can be assigned to programs, versions and procedures.

4 Port Mapper program protocol

4.1 The RPC protocol

The port mapper program maps RPC program and version numbers to transport-specific 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.

The port mapper also aids in broadcast RPC. A given RPC program will usually have different port number bindings on different machines, so there is no way to directly broadcast to all of these programs. The port mapper, however, does have a fixed port number. So, to broadcast to a given program, the client actually sends its message to the port mapper located at the broadcast address. Each port mapper that picks up the broadcast then calls the local service specified by the client. When the port mapper gets the reply from the local service, it sends the reply on back to the client.

```
Port Mapper Protocol Specification (in RPC Language)

const PMAP_PORT = 111;      /* portmapper port number */

/*
* A mapping of (program, version, protocol) to port number
*/
struct mapping {
        unsigned int prog;
        unsigned int vers;
        unsigned int prot;
        unsigned int port;
};

/*
* Supported values for the "prot" field
*/
const IPPROTO_TCP = 6;      /* protocol number for TCP/IP */
const IPPROTO_UDP = 17;     /* protocol number for UDP/IP */

/*
* A list of mappings
*/
struct *pmaplist {
        mapping map;
        pmaplist next;
};


/*
* Arguments to callit
*/
struct call_args {
        unsigned int prog;
        unsigned int vers;
        unsigned int proc;
        opaque args<>;
};
```

```
/*
* Results of callit
*/
struct call_result {
        unsigned int port;
struct *pmaplist {
        mapping map;
        pmaplist next;
};


/*
* Arguments to callit
*/
struct call_args {
        unsigned int prog;
        unsigned int vers;
        unsigned int proc;
        opaque args<>;
};

/*
* Results of callit
*/
struct call_result {
        unsigned int port;
        opaque res<>;
};


/*
* Port mapper procedures
*/
program PMAP_PROG {
        version PMAP_VERS {
                void
                PMAPPROC_NULL(void)          = 0;

                bool
                PMAPPROC_SET(mapping)        = 1;

                bool
                PMAPPROC_UNSET(mapping)      = 2;

                unsigned int
                PMAPPROC_GETPORT(mapping)    = 3;

                pmaplist
                PMAPPROC_DUMP(void)          = 4;

                call_result
                PMAPPROC_CALLIT(call_args)   = 5;
        } = 2;
} = 100000;
```

Port mapper operation

The portmapper program currently supports two protocols (UDP/IP and TCP/IP). The portmapper is contacted by talking to it on assigned port number 111 (SUNRPC [8]) on either of these protocols. The following is a description of each of the portmapper procedures:

PMAPPROC_NULL:

This procedure does no work. By convention, procedure zero of any protocol takes no parameters and returns no results.

PMAPPROC_SET:

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 a boolean response whose

value is *TRUE* if the procedure successfully established the mapping and *FALSE* otherwise. The procedure refuses to establish a mapping if one already exists for the tuple '(prog, vers, prot)'.

PMAPPROC_UNSET:

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` The protocol and port number fields of the argument are ignored.

PMAPPROC_GETPORT:

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 the program has not been registered. The 'port' field of the argument is ignored.

PMAPPROC_DUMP:

This procedure enumerates all entries in the port mapper's database. The procedure takes no parameters and returns a list of program, version, protocol, and port values.

PMAPPROC_CALLIT:

This procedure allows a caller to call another remote procedure on the same machine without knowing the remote procedure's port number. It is intended 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 of the remote procedure.

- This procedure only sends a response if the procedure was successfully executed and is silent (no response) otherwise.
- The port mapper communicates with the remote program using UDP/IP only.

The procedure returns the remote program's port number, and the bytes of results are the results of the remote procedure.

References

[1] Birrell, Andrew D. & Nelson, Bruce Jay; *'Implementing Remote Procedure Calls'*; XEROX CSL-83-7, October 1983.

[2] Cheriton, D.; *'VMTP: Versatile Message Transaction Protocol'*, Preliminary Version 0.3; Stanford University, January 1987.

[3] Diffie & Hellman; *'Net Directions in Cryptography'*; IEEE Transactions on Information Theory IT-22, November 1976.

[4] Harrenstien, K.; *'Time Server'*, RFC 738; Information Sciences Institute, October 1977.

[5] National Bureau of Standards; *'Data Encryption Standard'*; Federal Information Processing Standards Publication 46, January 1977.

[6] Postel, J.; *'Transmission Control Protocol - DARPA Internet Program Protocol Specification'*, RFC 793; Information Sciences Institute, September 1981.

[7] Postel, J.; *'User Datagram Protocol'*, RFC 768; Information Sciences Institute, August 1980.

[8] Reynolds, J. & Postel, J.; *'Assigned Numbers'*, RFC 923; Information Sciences Institute, October 1984.

Chapter 7 – NFS version 2 protocol specification

1 Introduction

This chapter specifies a protocol that Sun Microsystems, Inc., and others are using. It specifies it in standard ARPA RFC form. The Sun Network File System (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). Implementations exist for a variety of machines, from personal computers to supercomputers.

The supporting mount protocol allows the server to hand out remote access privileges to a restricted set of clients. It performs the operating system-specific functions that allow, for example, to attach remote directory trees to some local file system.

1.1 Remote Procedure Call

Sun's remote procedure call specification provides a procedure-oriented interface to remote services. Each server supplies a program that is a set of procedures. NFS is one such program. The combination of host address, program number, and procedure number specifies one remote service procedure. RPC does not depend on services provided by specific protocols, so it can be used with any underlying transport protocol. See the *Remote Procedure Calls: Protocol Specification* chapter of this manual.

1.2 External Data Representation

The External Data Representation (XDR) standard provides a common way of representing a set of data types over a network. The NFS Protocol Specification is written using the RPC data description language. For more information, see the *eXternal Data Representation Standard: Protocol Specification* chapter of this manual. Sun provides implementations of XDR and RPC, but NFS does not require their use. Any software that provides equivalent functionality can be used, and if the encoding is exactly the same it can interoperate with other implementations of NFS.

1.3 Stateless servers

The NFS protocol is stateless. That is, a server does not need to maintain any extra state information about any of its clients in order to function correctly. Stateless servers have a distinct advantage over stateful servers in the event of a failure. 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, or the network temporarily went down. The client of a stateful server, on the other hand, needs to either detect a server crash and rebuild the server's state when it comes back up, or cause client operations to fail.

This may not sound like an important issue, but it affects the protocol in some unexpected ways. We feel that it is worth a bit of extra complexity in the protocol to be able to write very simple servers that do not require fancy crash recovery. On the other hand, NFS deals with objects such as files and directories that inherently have state – what good would a file be if it did not keep its contents intact? The goal is to not introduce any extra state in the protocol itself. Another way to simplify recovery is by making operations 'idempotent' whenever possible (so that they can potentially be repeated).

2 NFS protocol definition

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. This RFC describes version two of the NFS protocol. Even in the second version, there are various obsolete procedures and parameters, which will be removed in later versions. An RFC for version three of the NFS protocol is currently under preparation.

2.1 File system model

NFS assumes a file system that is hierarchical, with directories as all but the bottom-level files. Each entry in a directory (file, directory, device, etc.) has a string name. Different operating systems may have restrictions on the depth of the tree or the names used, as well as using different syntax to represent the pathname, which is the concatenation of all the components (directory and file names) in the name. A file system is a tree on a single server (usually a single disc or physical partition) with a specified 'root'. Some operating systems provide a mount operation to make all file systems appear as a single tree, while others maintain a 'forest' of file systems. Files are unstructured streams of uninterpreted bytes. Version 3 of NFS will use a slightly more general file system model.

NFS looks up one component of a pathname at a time. It may not be obvious why it does not just take the whole pathname, work down the directories, and return a file handle when it is done. There are several 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. Other issues are discussed in *NFS Implementation Issues* below.

Although files and directories are similar objects in many ways, different procedures are used to read directories and files. This provides a network standard format for representing directories. The same argument as above could have been used to justify a 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 be just too slow.

RPC information

Authentication

The NFS service uses `AUTH_UNIX` `AUTH_DES` or `AUTH_SHORT` style authentication, except in the NULL procedure where `AUTH_NONE` is also allowed.

Transport Protocols

NFS currently is supported on UDP/IP only.

Port Number

The NFS protocol currently uses the UDP port number 2049. This is not an officially assigned port, so later versions of the protocol use the "Portmapping" facility of RPC.

Sizes of XDR structures

These are the sizes, given in decimal bytes, of various XDR structures used in the protocol:

```
/* The maximum number of bytes of data in a READ or WRITE request  */
const MAXDATA = 8192;

/* The maximum number of bytes pin a pathname argument */
const MAXPATHLEN = 1024;

/* The maximum number of bytes in a file name argument */
const MAXNAMLEN = 255;

/* The size in bytes of the opaque 'cookie' passed by READDIR */
const COOKIESIZE  = 4;

/* The size in bytes of the opaque file handle */
const FHSIZE = 32;
```

Basic data types

The following XDR definitions are basic structures and types used in other structures described further on.

```
stat

enum stat {
        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
};
```

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:

Some sort of hard error occurred when the operation was in progress. This could be a disc 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:

disc quota exceeded. The client's disc 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 disc.

ftype

```
enum ftype {
        NFNON = 0,
        NFREG = 1,
        NFDIR = 2,
        NFBLK = 3,
        NFCHR = 4,
        NFLNK = 5
};
```

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.

fhandle

```
typedef opaque fhandle[FHSIZE];
```

The `fhandle` is the file handle passed between the server and 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.

timeval

```
struct timeval {
        unsigned int seconds;
        unsigned int useconds;
};
```

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.

fattr

```
struct fattr {
        ftype        type;
        unsigned int mode;
        unsigned int nlink;
        unsigned int uid;
        unsigned int gid;
        unsigned int size;
        unsigned int blocksize;
        unsigned int rdev;
        unsigned int blocks;
        unsigned int fsid;
        unsigned int fileid;
        timeval      atime;
        timeval      mtime;
        timeval      ctime;
};
```

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 (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 the file takes up on disc; 'fsid' is the file system identifier for the filesystem containing 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. 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 will be fixed in future versions. The descriptions given below specify the bit positions using octal numbers.

Bit	Description
0040000	This is a directory; 'type' field should be NFDIR.
0020000	This is a character special file; 'type' field should be NFCHR.
0060000	This is a block special file; 'type' field should be NFBLK.
0100000	This is a regular file; 'type' field should be NFREG.
0120000	This is a symbolic link file; 'type' field should be NFLNK.
0140000	This is a named socket; '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.

Notes: The bits are the same as the mode bits returned by the *stat* system call in the UNIX system. The file type is specified both in the mode bits and in the file type. This will be fixed in future versions.

The 'rdev' field in the attributes structure is an operating system specific device specifier. It will be removed and generalized in the next revision of the protocol.

sattr

```
struct sattr {
        unsigned int mode;
        unsigned int uid;
        unsigned int gid;
        unsigned int size;
        timeval      atime;
        timeval      mtime;
};
```

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.

filename

```
typedef string filename<MAXNAMLEN>;
```

The type `filename` is used for passing file names or pathname components.

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.

attrstat

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

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.

diropargs

```
struct diropargs {
        fhandle  dir;
        filename name;
};
```

The `diropargs` structure is used in directory operations. The file handle 'dir' refers to the directory in which to find the file 'name'. A directory operation is one in which the directory is affected.

diropres

```
union diropres switch (stat status) {
        case NFS_OK:
        struct {
        fhandle file;
        fattr   attributes;
        } diropok;
        default:
        void;
};
```

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'.

Server procedures

The protocol definition is given as a set of procedures with arguments and results defined using the RPC language. A brief description of the function of each procedure should provide enough information to allow implementation.

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), 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.

```
/*
* Remote file service routines
*/
program NFS_PROGRAM {
version NFS_VERSION {
                void            NFSPROC_NULL(void)              = 0;
                attrstat        NFSPROC_GETATTR(fhandle)        = 1;
                attrstat        NFSPROC_SETATTR(sattrargs)      = 2;
                void            NFSPROC_ROOT(void)              = 3;
                diropres        NFSPROC_LOOKUP(diropargs)       = 4;
                readlinkres     NFSPROC_READLINK(fhandle)       = 5;
                readres         NFSPROC_READ(readargs)          = 6;
                void            NFSPROC_WRITECACHE(void)        = 7;
                attrstat        NFSPROC_WRITE(writeargs)        = 8;
                diropres        NFSPROC_CREATE(createargs)      = 9;
                stat            NFSPROC_REMOVE(diropargs)       = 10;
                stat            NFSPROC_RENAME(renameargs)      = 11;
                stat            NFSPROC_LINK(linkargs)          = 12;
                stat            NFSPROC_SYMLINK(symlinkargs)    = 13;
                diropres        NFSPROC_MKDIR(createargs)       = 14;
                stat            NFSPROC_RMDIR(diropargs)        = 15;
                readdirres      NFSPROC_READDIR(readdirargs)    = 16;
                statfsres       NFSPROC_STATFS(fhandle)         = 17;
        } = 2;
} = 100003;
```

Do nothing

```
void
NFSPROC_NULL(void) = 0;
```

This procedure does no work. It is made available in all RPC services to allow server response testing and timing.

Get file attributes

```
attrstat
NFSPROC_GETATTR (fhandle) = 1;
```

If the reply status is NFS_OK then the reply attributes contains the attributes for the file given by the input fhandle.

Set file attributes

```
struct sattrargs {
        fhandle file;
        sattr attributes;
        };

attrstat
NFSPROC_SETATTR (sattrargs) = 2;
```

The 'attributes' argument contains fields which are either -1 or are the new value for the attributes of 'file'. If the reply status is NFS_OK then the reply attributes have the attributes of the file after the 'SETATTR' operation has completed.

Note: The use of –1 to indicate an unused field in 'attributes' will be changed in the next version of the protocol.

Get filesystem root

```
void
NFSPROC_ROOT(void) = 3;
```

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 the *Mount Protocol Definition* below for details.)

Look up file name

```
diropres
NFSPROC_LOOKUP(diropargs) = 4;
```

If the reply 'status' is NFS_OK then the reply 'file' and reply 'attributes' are the file handle and attributes for the file 'name' in the directory given by 'dir' in the argument.

Read from symbolic link

```
union readlinkres switch (stat status) {
        case NFS_OK:
        path data;
        default:
        void;
};

readlinkres
NFSPROC_READLINK(fhandle) = 5;
```

If 'status' has the value NFS_OK then the reply 'data' is the data in the symbolic link given by the file referred to by the fhandle argument.

Note: Since NFS always parses pathnames on the client, the pathname in a symbolic link may mean something different (or be meaningless) on a different client or on the server if a different pathname syntax is used.

Read from file

```
struct readargs {
        fhandle file;
        unsigned offset;
        unsigned count;
        unsigned totalcount;
};

union readres switch (stat status) {
        case NFS_OK:
        fattr attributes;
        nfsdata data;
        default:
        void;
};

readres
NFSPROC_READ(readargs) = 6;
```

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'.

Note: The argument 'totalcount' is unused, and will be removed in the next protocol revision.

Write to cache

```
void
NFSPROC_WRITECACHE(void) = 7;
```

To be used in the next protocol revision.

Write to file

```
struct writeargs {
        fhandle file;
        unsigned beginoffset;
        unsigned offset;
        unsigned totalcount;
        nfsdata data;
};

attrstat
NFSPROC_WRITE(writeargs) = 8;
```

Writes 'data' beginning 'offset' bytes from the beginning of 'file'. The first byte of the file is at offset zero. If the reply 'status' is NFS_OK, then the reply 'attributes' contains the attributes of the file after the write has completed. The write operation is atomic. Data from this call to `WRITE` will not be mixed with data from another client's calls.

Note: The arguments 'beginoffset' and 'totalcount' are ignored and will be removed in the next protocol revision.

Create file

```
struct createargs {
        diropargs where;
        sattr attributes;
};

diropres
NFSPROC_CREATE(createargs) = 9;
```

The file 'name' is created in the directory given by '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.

Note: This routine should pass an exclusive create flag, meaning 'create the file only if it is not already there'.

Remove file

```
stat
NFSPROC_REMOVE(diropargs) = 10;
```

The file 'name' is removed from the directory given by 'dir'. A reply of NFS_OK means the directory entry was removed.

Note: Possibly non-idempotent operation.

Rename file

```
struct renameargs {
        diropargs from;
        diropargs to;
};

stat
NFSPROC_RENAME(renameargs) = 11;
```

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 the reply is NFS_OK the file was renamed. The RENAME operation is atomic on the server; it cannot be interrupted in the middle.

Note: possibly non-idempotent operation.

Create link to file

```
struct linkargs {
        fhandle from;
        diropargs to;
};

stat
NFSPROC_LINK(linkargs) = 12;
```

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 is NFS_OK a link was created. Any other return value indicates an error, and the link was 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' that is one greater than the value before the link.

Note: Possibly non-idempotent operation.

Create symbolic link

```
struct symlinkargs {
        diropargs from;
        path to;
        sattr attributes;
};

stat
NFSPROC_SYMLINK(symlinkargs) = 13;
```

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 is NFS_OK a link was created. Any other return value indicates an error, and the link was not created.

A symbolic link is a pointer to another file. The name given in 'to' is not interpreted by the server, only stored in the newly created file. When the client references a file that is a symbolic link, the contents of the symbolic link are normally transparently reinterpreted as a pathname to substitute. A READLINK operation returns the data to the client for interpretation.

Note: On UNIX servers the attributes are never used, since symbolic links always have mode 0777.

Create directory

```
diropres
NFSPROC_MKDIR (createargs) = 14;
```

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.

Note: possibly non-idempotent operation.

Remove directory

```
stat
NFSPROC_RMDIR(diropargs) = 15;
```

The existing empty directory 'name' in the directory given by 'dir' is removed. If the reply is NFS_OK the directory was removed.

Note: possibly non-idempotent operation.

Read from directory

```
struct readdirargs {
        fhandle dir;
        nfscookie cookie;
        unsigned count;
};

struct entry {
        unsigned fileid;
        filename name;
        nfscookie cookie;
        entry *nextentry;
};

union readdirres switch (stat status) {
        case NFS_OK:
        struct {
        entry *entries;
        bool eof;
        } readdirok;
        default:
        void;
};

readdirres
NFSPROC_READDIR (readdirargs) = 16;
```

Returns a variable number of directory entries, with a total size of up to 'count' bytes, from the directory given by 'dir'. If the returned value of 'status' is NFS_OK then it is followed by a variable number of 'entry's. Each 'entry' contains a 'fileid'

which consists of 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 *Basic Data Types* section.) The 'eof' flag has a value of TRUE if there are no more entries in the directory.

Get filesystem attributes

```
union statfsres (stat status) {
        case NFS_OK:
        struct {
        unsigned tsize;
        unsigned bsize;
        unsigned blocks;
        unsigned bfree;
        unsigned bavail;
        } info;
        default:
        void;
};

statfsres
NFSPROC_STATFS(fhandle) = 17;
```

If the reply 'status' is NFS_OK then the reply 'info' gives the attributes for the filesystem that contains file referred to by the input fhandle. 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.

Note: This call does not work well if a filesystem has variable size blocks.

3 NFS implementation issues

The NFS protocol is designed to be operating system independent, but since this version was designed in a UNIX environment, many operations have semantics similar to the operations of the UNIX file system. This section discusses some of the implementation-specific semantic issues.

3.1 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, some operating systems allow 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 such as renaming the file on remove, and only removing it on close. We believe that the server provides enough functionality to implement most file system semantics on the client.

Every NFS client can also potentially 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 follow the second remote mount would require loop detection, server lookup, and user revalidation. Instead, it was decided not to let clients cross a server's mount point. When a client does a LOOKUP on a directory on which the server has mounted a filesystem, 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.

Pathname interpretation

There are a few complications to the rule that pathnames are always parsed on the client. For example, symbolic links could have different interpretations on different clients. Another common problem for non-UNIX implementations is the special interpretation of the pathname '..' to mean the parent of a given directory. The next revision of the protocol will use an explicit flag to indicate the parent instead.

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 operating system 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. AUTH_DES (and the next revision of the NFS protocol) uses string names instead of numbers, but there are still complex problems to be solved.

Another problem arises due to the usually stateful open operation. Most operating systems check permission at open time, and then check that the file is open on each read and write request. With stateless servers, the server has no idea that the file is open and must do permission checking on each read and write call. On a local filesystem, a user can open a file and 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 regardless of the permission setting.

A similar problem has to do with paging in from a file over the network. The operating system usually checks for execute permission before opening a file for demand paging, and 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 not 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 most operating systems, a particular user (on the user ID zero) has access to all files no matter what permission and ownership they have. This 'super-user' permission may not be allowed on the server, since anyone who can become super-user on their workstation could gain access to all remote files. The UNIX server by default maps user id 0 to –2 before doing its access checking. This works except for NFS root filesystems, where super-user access cannot be avoided.

3.2 Setting RPC parameters

Various file system parameters and options should be set at mount time. The mount protocol is described in the appendix below. For example, 'Soft' mounts as well as 'Hard' mounts are usually both provided. Soft mounted file systems return errors when RPC operations fail (after a given number of optional retransmissions), while hard mounted file systems continue to retransmit forever. Clients and servers may need to keep caches of recent operations to help avoid problems with non-idempotent operations.

4 Mount protocol definition

4.1 Mount protocol definition

The mount protocol is separate from, but related to, the NFS protocol. It provides operating system specific services to get the NFS off the ground – looking up server 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 intended for advisory use only, for example, to warn possible clients when a server is going down.

Version one of the mount protocol is used with version two of the NFS protocol. The only connecting point is the `fhandle` structure, which is the same for both protocols.

RPC information

Authentication

The mount service uses `AUTH_UNIX` and `AUTH_DES` style authentication only.

Transport Protocols

The mount service is currently supported on UDP/IP only.

Port Number

Consult the server's portmapper, described in the Remote Procedure Calls: Protocol Specification, to find the port number on which the mount service is registered.

Sizes of XDR structures

These are the sizes, given in decimal bytes, of various XDR structures used in the protocol:

```
/* The maximum number of bytes in a pathname argument */
const MNTPATHLEN = 1024;

/* The maximum number of bytes in a name argument */
const MNTNAMLEN = 255;

/* The size in bytes of the opaque file handle */
const FHSIZE = 32;
```

Basic data types

This section presents the data types used by the mount protocol. In many cases they are similar to the types used in NFS.

fhandle

```
typedef opaque fhandle[FHSIZE];
```

The type `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 *Basic Data Types* in the definition of the NFS protocol, above.

fhstatus

```
union fhstatus switch (unsigned status) {
        case 0:
        fhandle directory;
        default:
        void;
};
```

The type `fhstatus` is a union. 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.

dirpath

```
typedef string dirpath<MNTPATHLEN>;
```

The type `dirpath` is a server pathname of a directory.

name

```
typedef string name<MNTNAMLEN>;
```

The type `name` is an arbitrary string used for various names.

Server procedures

The following sections define the RPC procedures supplied by a mount server.

```
/*
* Protocol description for the mount program
*/

program MOUNTPROG {
/*
* Version 1 of the mount protocol used with
* version 2 of the NFS protocol.
*/
        version MOUNTVERS {
                void        MOUNTPROC_NULL(void)     = 0;
                fhstatus    MOUNTPROC_MNT(dirpath)   = 1;
                mountlist   MOUNTPROC_DUMP(void)     = 2;
                void        MOUNTPROC_UMNT(dirpath)  = 3;
                void        MOUNTPROC_UMNTALL(void)  = 4;
                exportlist  MOUNTPROC_EXPORT(void)   = 5;
        } = 1;
}
```

Do nothing

```
void
MNTPROC_NULL(void) = 0;
```

This procedure does no work. It is made available in all RPC services to allow server response testing and timing.

Add mount entry

```
fhstatus
MNTPROC_MNT(dirpath) = 1;
```

If the reply 'status' is 0, then the 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'.

Return mount entries

```
struct *mountlist {
        name      hostname;
        dirpath   directory;
        mountlist nextentry;
};

       mountlist
       MNTPROC_DUMP(void) = 2;
```

Returns the list of remote mounted filesystems. The 'mountlist' contains one entry for each 'hostname' and 'directory' pair.

Remove mount entry

```
void
MNTPROC_UMNT(dirpath) = 3;
```

Removes the mount list entry for the input 'dirpath'.

Remove all mount entries

```
void
MNTPROC_UMNTALL(void) = 4;
```

Removes all of the mount list entries for this client.

Return export list

```
struct *groups {
        name grname;
        groups grnext;
};

struct *exportlist {
        dirpath filesys;
        groups groups;
        exportlist next;
};

exportlist
MNTPROC_EXPORT(void) = 5;
```

Returns 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 'filesys', and the group name is in the list 'groups'.

Note: The export list should contain more information about the status of the filesystem, such as a read-only flag.

Chapter 8 – 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. 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: match and enumerate. 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 match and enumerate operations in a network environment, where 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 provide 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

The 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, clients must also know the server's host internet address, and the server's rendezvous port. The server listens for service requests at ports associated with a particular transport protocol – TCP/IP 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.

2 YP database servers

2.1 Maps and map operations

Maps are named sets of key-value pairs. Keys and their values are counted binary objects, and may be ASCII information, but 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.

Map structure

Typical implementations for YP maps are files or database management systems. The design of the YP's map database is an implementation detail, and is unspecified by the protocol.

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.

Map entry enumeration

It is possible to get the first key-value pair in a map with `YPPROC_FIRST`, and the next key-value pair with `YPPROC_NEXT`. Calling "get first" once and "get next" until the return value indicates there are no more entries in the map will retrieve each entry once. Making the same calls on the same map at the same YP database server will enumerate all entries in the same order. The actual order, however, is unspecified. Enumerating a map at a different YP database server will not necessarily return entries in the same order.

Entire map retrieval

The `YPPROC_ALL` operation retrieves all key-value pairs in a map, with a single RPC request. This is faster than map entry enumeration, and more reliable, since it uses TCP. Ordering is the same as when enumeration is applied.

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 database servers

For each map, there is one 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.

In order to escape from the idiosyncrasies of any particular implementation, all maps should be uniformly timestamped.

Functions to aid in map propagation

The way a map is transferred from one server to another is not specified by the YP protocol. One possibility would be for the system administrator to do it manually. Another would be for the YP database server to activate another process to perform the map transfer. A third would be for a server to enumerate a recent version of the map, using the normal client map enumeration functions.

The `YPPROC_XFR` procedure requests the YP server to update a map, and permits the actual transfer agent (some server process) to call back the requestor with a summary status.

2.4 Domains

Domains 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 organisations 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.

2.5 Non-features

The following capabilities are not included in the current YP protocols:

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.

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. Use `YPPROC_ALL` to avoid these problems.

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.

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 database server protocol definition

This section describes version 2 of the protocol. It is likely that changes will be made to successive versions as the service matures.

RPC constants

All numbers are in decimal.

YPPROG 100004

The YP database server protocol program number.

YPVERS 2

The current YP protocol version.

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 64

The maximum number of characters in a YP host name.

Remote procedure return values

This section presents the return status values returned by several of the YP remote procedures. All numbers are in decimal.

ypstat

```
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. */
        YP_VERS    = -8  /* YP server version mismatch. */
} ypstat
```

ypxfrstat

```
typedef enum {
        YPXFR_SUCC    =   1,  /* Success */
        YPXFR_AGE     =   2,  /* Master's version not newer */
        YPXFR_NOMAP   =  -1,  /* Can't find server for map */
        YPXFR_NODOM   =  -2,  /* Domain not supported */
        YPXFR_RSRC    =  -3,  /* Local resouce alloc failure */
        YPXFR_RPC     =  -4,  /* RPC failure talking to server */
        YPXFR_MADDR   =  -5,  /* Can't get master address */
        YPXFR_YPERR   =  -6,  /* YP server/map db error */
        YPXFR_BADARGS=  -7,  /* Request arguments bad */
        YPXFR_DBM     =  -8,  /* Local database failure */
        YPXFR_FILE    =  -9,  /* Local file I/O failure */
        YPXFR_SKEW    = -10,  /* Map version skew in transfer */
        YPXFR_CLEAR   = -11,  /* Can't clear local ypserv */
        YPXFR_FORCE   = -12,  /* Must override defaults */
        YPXFR_XFRERR  = -13,  /* ypxfr error */
        YPXFR_REFUSED= -14    /* ypserv refused transfer */
} ypxfrstat
```

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

```
typedef struct {
        domainname
        mapname
        unsigned long ordernum
        peername
} ypmap_parms
```

This contains parameters giving information about map mapname within domain domainname; peername is the name of the map's master YP database server. If any of the three strings is null, it indicates information is unknown or unavailable. The ordernum element contains a binary value representing the value of the map's order number; if unavailable, this is 0.

ypreq_xfr

```
typedef struct {
        struct ypmap_parms map_parms
        unsigned long transid
        unsigned long prog unsigned short port
} ypreq_xfr
```

ypresp_val

```
typedef struct {
        ypstat
        valdat
} ypresp_val
```

ypresp_key_val

```
typedef struct {
        ypstat
        keydat
        valdat
} ypresp_key_val
```

ypresp_master

```
typedef struct {
        ypstat
        peername
} ypresp_master
```

ypresp_order

```
typedef struct {
        ypstat
        unsigned long ordernum
} ypresp_order
```

ypresp_all

```
typedef union switch (boolean more) {
        TRUE:
                ypresp_key_val
        FALSE:
                struct { }
} ypresp_all
```

ypresp_xfr

```
typedef struct {
        unsigned long transid
        ypxfrstat xfrstat
} ypresp_xfr
```

ypmaplist

```
typedef struct {
        mapname
        ypmaplist *
} ypmaplist
```

ypresp_maplist

```
typedef struct {
        ypstat
        ypmaplist *
} ypresp_maplist
```

YP database 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.

Do nothing

Procedure 0, Version 2.

```
0. YPPROC_NULL ( ) returns ( )
```

This takes no arguments, does no work, and returns nothing. It is made available in all RPC services to allow server response testing and timing.

Do you serve this domain?

Procedure 1, Version 2.

```
1. YPPROC_DOMAIN (domain) returns (serves)
        domainname domain;
        boolean serves;
```

This returns `TRUE` if the server serves `domain`, and `FALSE` otherwise. This procedure allows a potential client to determine if a given server supports a certain domain.

Answer only if you serve this domain

Procedure 2, Version 2.

```
2. YPPROC_DOMAIN_NONACK (domain) returns (serves)
        domainname domain;
        boolean serves;
```

This procedure returns TRUE if the server serves `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.

The current implementation does `return` in the FALSE case by forcing an RPC decode error.

Return value of a key

Procedure 3, Version 2.

```
3. YPPROC_MATCH (req) returns (resp)
        ypreq_key req;
        ypresp_val resp;
```

This returns the value associated with the datum `keydat` in `req`. If the `status` element in `resp` has the value `YP_TRUE`, the value data are returned in the datum `valdat`.

Get first key-value pair in map

Procedure 4, Version 2.

```
4. YPPROC_FIRST (req) returns (resp)
        ypreq_key req;
        ypresp_key_val resp;
```

If `status` has the value `YP_TRUE`, this returns the first key-value pair from the map named in `req` to the `keydat` and `valdat` elements within `resp`. When `status` contains the value `YP_NOMORE`, the map is empty.

Get next key-value pair in map

Procedure 5, Version 2.

```
5. YPPROC_NEXT (req) returns (resp)
        ypreq_key req;
        ypresp_key_val resp;
```

If `status` has the value `YP_TRUE`, this returns the key-value pair following the key-value named `req` to the `keydat` and `valdat` elements within `resp`. If the passed key is the last key in the map, the value of `status` is `YP_NOMORE`.

Transfer map

Procedure 6, Version 2.

```
6. YPPROC_XFR (req) returns (resp)
        ypreq_xfr req;
        ypresp_xfr resp;
```

The action taken in response to this request is unspecified, and is implementation dependent. The intention is to indicate to the server that a map should be updated, and to allow the actual transfer agent (whether it be the YP server process, or some other process) to call back the requestor with a summary status.

The transfer agent should call back the program running on the requesting host with program number `req.prog`, program version 1, and listening at port `req.port`. The procedure number is 1, and the callback data is of type `ypresp_xfr`. The `transid` field should turn around `req.transid`, and the `xfrstat` field should be set appropriately.

Reinitialise internal state

Procedure 7, Version 2.

```
7. YPPROC_CLEAR ( ) returns ( )
```

The action taken in response to this request is unspecified, and is implementation dependent. Different server implementations may have different amounts of internal state (open files, or the current map, for example). This request signals that all such state should be expunged.

Get all key-value pairs in map

Procedure 8, Version 2.

```
8. YPPROC_ALL (req) returns (resp)
        ypreq_nokey req;
        ypresp_all resp;
```

This allows all key-value pairs from a map to be transferred with a single RPC request. When the union's discriminant is FALSE, no more key-value pairs will be returned. The status field of the last `rpresp_key_val` structure should be consulted to determine why the flow of returned key-value pairs has stopped.

Get map master name

Procedure 9, Version 2.

```
9. YPPROC_MASTER (req) returns (resp)
        ypreq_nokey req;
        ypresp_master resp;
```

This returns the map's master YP server inside the `resp` structure.

Get map order number

Procedure 10, Version 2.

```
10. YPPROC_ORDER (req) returns (resp)
        ypreq_nokey req;
        ypresp_order resp;
```

This returns a map's order number as an unsigned long integer, which indicates when the map was built. This quantity represents the number of seconds since the midnight before 1 January 1970 GMT.

Get all maps in domain

Procedure 11, Version 2.

```
11. YPPROC_MAPLIST (req) returns (resp)
        domainname req;
        ypresp_maplist resp;
```

This returns a list of all the maps in a domain.

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 the 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 somehow, but that the way is either complicated, time-consuming, or resource-consuming.

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 2 of the protocol. It is likely that changes will be made to successive versions as the service matures.

RPC constants

All numbers are decimal.

YPBINDPROG 100007

The YP binder protocol program number.

YPBINDVERS 2

The current YP binder protocol version.

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 domain */
        YPBIND_ERR_RESC 3/* Can't allocate system resource */
} 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.

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

```
typedef struct {
        unsigned long ypbind_binding_addr
        unsigned short ypbind_binding_port
} ypbind_binding
```

This contains the information necessary to bind a client to a YP database server in the Internet environment: ypbind_binding_addr holds the host IP address (four bytes), and ypbind_binding_port holds the port address (two 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

```
typedef struct {
        union switch (enum ypbind_resptype status) {
                YPBIND_SUCC_VAL:
                ypbind_binding
                YPBIND_FAIL_VAL:
                        ypbinderr
                default:
                        { }
        }
} ypbind_resp
```

This is the response to a YPBINDPROC_DOMAIN request.

ypbind_setdom

```
typedef struct {
        domainname
        ypbind_binding
        version
} ypbind_setdom
```

This is the input data structure for the YPBINDPROC_SETDOM procedure.

YP binder remote procedures

Like the YP procedures earlier, these procedures are described using the XDR data definition language.

Do nothing

Procedure 0, Version 2.

```
0. YPBINDPROC_NULL ( ) returns ( )
```

This does no work. It is made available in all RPC services to allow server response testing and timing.

Get current binding for a domain

Procedure 1, Version 2.

```
1. YPBINDPROC_DOMAIN (domain) returns (resp)
        domainname domain;
        ypbind_resp resp;
```

This returns the binding information necessary to address a YP database server within the Internet environment.

Set domain binding

Procedure 2, Version 2.

```
2. YPBINDPROC_SETDOM (setdom) returns ( )
        ypbind_setdom setdom;
```

This instructs a YP binder to use the passed information as its current binding information for the passed domain.

Chapter 9 – Inter-Process Communication primer

1 Introduction

This document provides an introduction to the Inter-Process Communication (IPC) facilities on the RISC iX 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.

One of the most important features added in the Berkeley 4.2 release of the UNIX operating system was substantial new IPC facilities. UNIX has previously been weak in doing IPC. Until recently, the only standard mechanism that allowed two processes to communicate were pipes. Unfortunately, pipes are very 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.

The remainder of this document is organised 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.

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; eg 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 that 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. 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.

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. Their use is discussed in Section 5, *Advanced topics*.

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. A socket is like a file descriptor. The user is returned a descriptor (a small integer) 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; for the Internet domain AF_INET. (The manifest constants are named AF_whatever as they indicate the *address format* to use in interpreting names). 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 results in a stream socket being created with the TCP protocol providing the underlying communication support. The TCP protocol requires SOCK_STREAM, while the UDP protocol requires SOCK_DGRAM. 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:

```
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 contain a path name and a family, which is always AF_UNIX. 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 addr;
addr.sun_family = AF_UNIX;
strcpy(addr.sun_path, "/dev/foo");
bind(s, (struct sockaddr*)&addr, strlen(addr.sun_path)+ sizeof(addr.sun_family));
```

Note: In the current implementation, the file name referred to in `addr.sun_path` is created as a socket in the file system. The caller must, therefore, have write permission in the directory where `addr.sun_path` is to reside, and this file should be deleted by the caller when it is no longer required.

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, (struct sockaddr*)&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 server, when willing to offer its advertised services, passively *listens* on its socket. The client requests services from the server by initiating a *connection* to the server's 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, (struct sockaddr*)&server, strlen(server.sun_path)+sizeof(server.sun_family));
```

while in the Internet domain,

```
struct sockaddr_in server;
...
connect(s, (struct sockaddr*)&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, (You must do a *getsockname* call to retrieve the binding). 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 destination host is down, or because problems in the network resulted in transmissions being lost.

`ECONNREFUSED`

The host refused service for some reason. 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: `listen` and `accept`. Note, however, that it isn't necessary to perform either step with UDP sockets.

The first is to indicate a willingness to listen for incoming connection requests:

```
listen(s, 5);
```

The last 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, individual messages comprising 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 five 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.

Second, with a socket marked as listening, a server may `accept` a connection:

```
struct sockaddr_in from;  /*for the internet domain use sockaddr_un for the unix domain*/
...
fromlen = sizeof(from);
snew = accept(s, (struct sockaddr*)&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 initialised 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.

The `accept` call 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,

```
char buf[BUFSIZ];
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 (defined in `sys/socket.h`) 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_PEEK` 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 (eg 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.

When a client or server machine crashes, the socket stays open on the machine that hasn't crashed. Afterwards, writing will result in a `SIGPIPE`, reading in an `EOF`.

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, (struct sockaddr*)&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 recognise 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, (struct sockaddr*)&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, users of 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; a second connect will change the destination address, and a connect to a null address (family AF_UNSPEC) will disconnect. 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). `Accept` and `listen` are not used with datagram sockets.

While a datagram socket is connected, errors from recent `send` calls may be returned asynchronously. These errors may be reported on subsequent operations on the socket, or a special socket option used with `getsockopt`, SO_ERROR, may be used to interrogate error status. A `select` for reading or writing will return true when an error indication has been received. The reset operation will return the error, and the error status is cleared

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:

```
#include <sys/time.h>
#include <sys/types.h>
...
fd_set readmask, writemask, exceptmask;
struct timeval timeout;
 ...
select(nfds, &readmask, &writemask, &exceptmask, &timeout);
```

`Select` takes as arguments pointers to three sets, 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; out-

of-band data is the only exceptional condition currently implemented by the socket If the user is not interested in certain conditions (ie, read, write, or exceptions), the corresponding argument to the select should be a null pointer.

Each set is actually a structure containing an array of long integer bit masks; the size of the array is set by the definition FD_SETSIZE. The array is be long enough to hold one bit for each of FD_SETSIZE file descriptors.

The macros FD_SET(fd, &mask) and FD_CLR(fd, &mask) have been provided for adding and removing file descriptor fd in the set mask. The set should be zeroed before use, and the macro FD_ZERO(&mask) has been provided to clear the set mask. The parameter nfds in the select call specifies the range of file descriptors (ie one plus the value of the largest descriptor) to be examined in a set.

A timeout value may be specified if the selection is not to last more than a predetermined period of time. If the fields in timeout are 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*. Select normally returns the number of file descriptors selected; if the select call returns due to the timeout expiring, then the value 0 is returned. If the select terminates because of an error or interruption, a –1 is returned with the error number in errno, and with the file descriptor masks unchanged.

Assuming a successful return, the three sets will indicate which file descriptors are ready to be read from, written to, or have exceptional conditions pending. The status of a file descriptor in a select mask may be tested with the FD_ISSET(fd, &mask) macro, which returns a non-zero value if fd is a member of the set mask, and 0 if it is not.

To determine if there are connections waiting on a socket to be used with an accept call, select can be used, followed by a FD_ISSET(fd, &mask) macro to check for read readiness on the appropriate socket. If FD_ISSET returns a non-zero value, indicating permission to read, then a connection is pending on the socket.

2.9 Socket options

It is possible to set and get a number of options on sockets via the `setsockopt` and `getsockopt` system calls. These options include such things as marking a socket for broadcasting, not to route, to linger on close, etc. The general forms of the calls are:

```
setsockopt(s, level, optname, optval, optlen);
```

and

```
getsockopt(s, level, optname, optval, optlen);
```

The parameters to the calls are as follows: s is the socket on which the option is to be applied. Level specifies the protocol layer on which the option is to be applied; in most cases this is the 'socket level', indicated by the symbolic constant SOL_SOCKET, defined in `<sys/socket.h>`. The actual option is specified in optname, and is a symbolic constant also defined in `<sys/socket.h>`. `Optval` and `Optlen` point to the value of the option (in most cases, whether the option is to be turned on or off), and the length of the value of the option, respectively. For `getsockopt`, `optlen` is a value-result parameter, initially set to the size of the storage area pointed to by optval, and modified upon return to indicate the actual amount of storage used.

An example should help clarify things. It is sometimes useful to determine the type (eg, stream, datagram, etc.) of an existing socket; programs under `inetd` (described below) may need to perform this task. This can be accomplished as follows via the SO_TYPE socket option and the `getsockopt` call:

```
#include <sys/types.h>
#include <sys/socket.h>

int type, size;

size = sizeof (int);

if (getsockopt(s, SOL_SOCKET, SO_TYPE, (char *) &type, &size) < 0) {
...
}
```

After the `getsockopt` call, type will be set to the value of the socket type, as defined in `<sys/socket.h>`. If, for example, the socket were a datagram socket, type would have the value corresponding to SOCK_DGRAM.

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 RISC iX 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 by 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* that 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; /* list of addresses null terminated */ }
;
#define h_addr h_addr_list[0]     /*first address, betwork byte order */
```

Note that the `h_addr` field in the structure definition is defined as a pointer to a `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 the address type and a null terminated list of variable length addresses. The routine `gethostbyname` takes a host name and returns a `hostent` structure, while the routine `gethostbyaddr` maps host addresses into a `hostent` structure. The list of addresses is required because it is possible for a host to have many addresses, all having the same name.

The database for these calls is provided either by `/etc/hosts` or by use of a nameserver, named(8).

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`, `getnetbynumber` and `getnetent` are the network counterparts to the host routines described above. the routines extract their instructions from `/etc/networks`.

3.3 Protocol names

For protocols (defined in `/etc/protocols`) the `protoent` structure defines the protocol-name mapping used with the routines `getprotobyname`, `getprotobynumber` and `getprotoent`

```
struct protoent {
        char    *p_name;  /* official protocol name */
        char    **p_aliases; /* alias list */
        int     p_proto;  /* protocol number */
};
```

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.

Note: Internet port numbers below 1024 are reserved for server processes running as root.

Services availables are defined in `/etc/service`. 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 number, network byte order */
        char    *s_proto; /* protocol to use */
};
```

The routine `getservbyname` 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

```
sp = getservbyname("telnet", "tcp");
```

returns only that telnet server which uses the TCP protocol. The routines `getservbyport` and `getservent` 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.

When prototyping new services, it is easiest to hard-code a port number not in `/etc/services`, then later install your service there.

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 below:

```
#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 server;
        struct servent *sp;
        struct hostent *hp;
        int s;
        ...
        if ((sp = getservbyname("login", "tcp")) == NULL) {
                fprintf(stderr,
                        "rlogin: tcp/login: not a service\n");
                exit(1);
        }
        if ((hp = gethostbyname(argv[1])) == NULL) {
                fprintf(stderr,
                        "rlogin: %s: unknown host\n", argv[1]);
                exit(2);
        }
        /* only filling in part
         */
        bzero((char *)&server, sizeof(server));
        bcopy(hp->h_addr, (char *)&server.sin_addr, hp->h_length);
        server.sin_family = hp->h_addrtype;
        server.sin_port = sp->s_port;
        if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
                perror("rlogin: socket");
                exit(3);
        }
        ...
        /* let connect do the bind() */
        if (connect(s, (char *)&server, sizeof(server)) < 0) { perror("rlogin: connect");
                exit(5);
        }
        ...
}
```

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.

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 summarises 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 an R140, or machine with similar architecture, host byte order is the reverse of network order. 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 emphasing 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 where they are not needed, 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 recognised 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. That is the server process remains dormant until a correction is requested by a client's connection to the server address.

In order to avoid a lot of dormant servers clogging the system, an alternative mechanism has been implemented for Internet servers. An internet 'super-server' `inetd` listens on a variety of ports (determined at startup by reading a configuration file `/etc/inetd.conf`. When a connection is requested to a port on which `inetd` is listening, `inetd` executes the appropiate server to handle the client.

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 its services by listening at a well-known location. For example, the remote login server's main loop is of the form shown below:

```
#include <stdio.h>
#include <net/inet.h>

main(argc, argv)
        int argc; char **argv;
{
        int f;
        struct sockaddr_in sin, from;
        struct servent *sp;

        if ((sp = getservbyname("login", "tcp")) == NULL) {
                fprintf(stderr,
                        "rlogind: tcp/login: not a 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, (struct sockaddr*)&sin, sizeof(sin)) < 0) {
                ...
        }
        listen(f, 5);
        for (;;) {
                int g, len = sizeof(from);
                g = accept(f, (struct sockaddr*)&from, &len);
                if (g < 0) {
                        if (errno != EINTR)
                                syslog (LOG_ERR,"rlogind:accept%m");
                        continue;
                }
                if (fork() == 0) {
                        close(f);
                        doit(g, &from);
                /* should never return */
                }
                close(g);
        }
}
```

The first step taken by the server is look up its service definition:

```
if ((sp = getservbyname("login", "tcp")) == NULL) {
        fprintf(stderr, "rlogind: tcp/login: not a 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:

```
for (i=0; ic3; ++i)
        close(i);
open ("/", O_RDONLY);
dup2(0,1);
dup2(0,2);
i=open ("/dev/tty", O_RDWR);
if  (i>=0) {
        ioctl(i, TIOCNOTTY, 0);
        close(i);
}
```

Note: Once a server has disassocaited itself it can no longer send reports of errors to a terminal – it must log them via `syslog`.

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, (struct sockaddr*)&from, &len);
        if (g < 0) {
                if (errno != EINTR)
                        syslog(LOG_ERR"rlogind: accept : %m ");
                continue;
        }
        if (fork() == 0) {        /* child */
                close(f);
                doit(g, &from);
                /* should never return */
        }
        close(g);                 /* parent */
}
```

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 for authenticating clients. The `doit` routine communicates using the socket, then closes it and exits when done.

4.2 Clients

The client side of the remote login service was shown earlier. 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:

```
if ((sp = getservbyname("login", "tcp")) == NULL) {
        fprintf(stderr, "rlogin: tcp/login: not a service\n");
        exit(1);
}
```

Next the destination host is looked up with a `gethostbyname` call:

```
if ((hp = gethostbyname(argv[1])) == 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 *)&server, sizeof(server));
bcopy(hp->h_addr, (char *)&server.sin_addr, hp->h_length);
server.sin_family = hp->h_addrtype;
server.sin_port = sp->s_port;
```

A socket is created, and a connection initiated. Note, `connect` will implicitly perform a `bind` call since `s` is inbound.

```
if ((s = socket(hp->h_addrtype, SOCK_STREAM, 0)) < 0) {
        perror("rlogin: socket");
        exit(3);
}
        ...
if (connect(s, (char *)&server, sizeof(server)) < 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* program. The output generated is illustrated below.

```
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
```

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 below:

```
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));
        ...
        signal(SIGALRM, onalrm);
        onalrm();
        for (;;) {      struct whod wd;
                int cc, whod, len = sizeof(from);
                cc = recvfrom(s, (char *)&wd, sizeof(struct whod),
                        0, (struct sockaddr*) &from, &len);
                if (cc <= 0) {
                        if (cc < 0 && errno != EINTR)
                                syslog(LOG_ERR,"rwhod: recv: %M");
                        continue;
                }
                if (from.sin_port != sp->s_port) {
                syslog(LOG_ERR, "rwhod: %d: bad from port",  ntohs(from.sin_port));
                        continue;
                }
                ...
                if (!verify(wd.wd_hostname)) {
syslog(LOG_ERR,  "rwhod: malformed hostname from %x", ntohl(from.sin_addr.s_addr));
                        continue;
                }
            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);
        }
}
```

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.

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 neighbours (based on the status received). This, unfortunately, requires some bootstrapping information, as a server started up on a quiet network will have no known neighbours 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 neighbours. If each server has at least one neighbour supplying it, status information may then propagate through a neighbour 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. (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.)

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 system attempts to isolate host-specific information from applications by providing system calls which return the necessary information. (An example of such a system call is the *gethostname* call, which returns the host's official name.)

The `rwho` server can (by using an `ioctl`) find the set A, directly connected networks. This, combined with the ability to carry out local network broadcast at the socket level, allows `rwho` to solve the problem of propagating status informamtion in a site independent manner.

5 Advanced topics

A number of facilities has yet to be discussed. For most users of the IPC, mechanisms already described suffice for constructing distributed applications. However, others may find need to use some of the features considered 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 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, supply the `MSG_OOB` flag to a `send` or `sendto` call, and to receive out-of-band data, indicate the `MSG_OOB` flag to a `recv` or `recvfrom` 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 below:

```
#include <sys/ioctl.h>
#include <sys/file.h>
.....
oob()
{
        int out = FWRITE;
        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));
        }
        if (recv(rem, &mark, 1, MSG_OOB <0){
        Perror ("recv");
        ...
}
}
```

5.2 Non-blocking sockets

It is occasionally convenient to make use of sockets which do not block; that is, I/O requests which cannot complete immediately and would therefore cause the process to be suspended awaiting completion are not executed, and an error code is returned. Once a socket has been created via the socket call, it may be marked as non-blocking by `fcntl` as follows:

```
#include <fcntl.h>
 ...
int     s;
 ...
s = socket(AF_INET, SOCK_STREAM, 0);
 ...
if (fcntl(s, F_SETFL, FNDELAY) < 0)
        perror("fcntl F_SETFL, FNDELAY");
        exit(1);
}
 ...
```

When performing non-blocking I/O on sockets, one must be careful to check for the error EWOULDBLOCK (stored in the global variable errno), which occurs when an operation would normally block, but the socket it was performed on is marked as non-blocking. In particular, accept, connect, send, recv, read, and write can all return EWOULDBLOCK, and processes should be prepared to deal with such return codes. If an operation such as a send cannot be done in its entirety, but partial writes are sensible (for example, when using a stream socket), the data that can be sent immediately will be processed, and the return value will indicate the amount actually sent.

5.3 Interrupt driven socket I/O

The SIGIO signal allows a process to be notified via a signal when a socket (or more generally, a file descriptor) has data waiting to be read. Use of the SIGIO facility requires three steps: First, the process must set up a SIGIO signal handler by use of the signal or sigvec calls. Second, it must set the process id or process group id which is to receive notification of pending input to its own process id, or the process group id of its process group (note that the default process group of a socket is group zero). This is accomplished by use of an fcntl call. Third, it must enable asynchronous notification of pending I/O requests with another fcntl call. Sample code to allow a given process to receive information on pending I/O requests as they occur for a socket s is given. With the addition of a handler for SIGURG, this code can also be used to prepare for receipt of SIGURG signals.

```
#include <fcntl.h>
 ...
int     io_handler();
 ...
signal(SIGIO, io_handler);

/* Set the process receiving SIGIO/SIGURG signals to us */
  if (fcntl(s, F_SETOWN, getpid()) < 0) {
        perror("fcntl F_SETOWN");
        exit(1);
}
/* Allow receipt of asynchronous I/O signals */

if (fcntl(s, F_SETFL, FASYNC) < 0) {
        perror("fcntl F_SETFL, FASYNC");
        exit(1);
}
```

5.4 Signals and process groups

Due to the existence of the SIGURG and SIGIO signals each socket has an associated process number, just as is done for terminals. This value is initialized to zero, but may be redefined at a later time with the F_SETOWN `fcntl`, such as was done in the code above for SIGIO. To set the socket's process id for signals, positive arguments should be given to the `fcntl` call. To set the socket's process group for signals, negative arguments should be passed to fcntl. Note that the process number indicates either the associated process id or the associated process group; it is impossible to specify both at the same time. A similar `fcntl`, F_GETOWN, is available for determining the current process number of a socket.

Another signal which is useful when constructing server processes is SIGCHLD. This signal is delivered to a process when any child processes have changed state. Normally servers use the signal to "reap" child processes that have exited without explicitly awaiting their termination or periodic polling for exit status. For example, the remote login server loop shown may be augmented as shown below.

```
int reaper();
 ...
signal(SIGCHLD, reaper);
listen(f, 5);
for (;;) {
        int g, len = sizeof (from);
  g = accept(f, (struct sockaddr *)&from, &len,);
        if (g < 0) {
                if (errno != EINTR)
                        syslog(LOG_ERR, "rlogind: accept: %m");
                continue;
        }
        ...
}
 ...
#include <wait.h>
reaper()
{
         union wait status;
        while (wait 3 (&status, WNOHANG, 0) > 0)
}
```

If the parent server process fails to reap its children, a large number of 'zombie' processes may be created.

5.5 Psuedo terminals

Many programs will not function properly without a terminal for standard input and output. Since sockets do not provide the semantics of terminals, 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, while data written on the master side are processed as terminal input for the slave. 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 as if it were manipulating the keyboard and reading the screen on a real terminal. The purpose of this abstraction is to preserve terminal semantics over a network connection, that is, the slave side appears as a normal terminal to any process reading from or writing to it.

For example, 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 a character that generates an interrupt on the remote machine that flushes terminal output, the pseudo-terminal generates a control message for the server process. The server then sends an out of

band message to the client process to signal a flush of data at the real terminal and on the intervening data buffered in the network.

Under 4.3BSD, the name of the slave side of a pseudo-terminal is of the form `/dev/ttyxy`, where x is a single letter starting at 'p' and continuing to 't'. y is a hexadecimal digit (ie, a single character in the range 0 through 9 or 'a' through 'f'). The master side of a pseudo-terminal is `/dev/ptyxy`, where x and y correspond to the slave side of the pseudo-terminal.

In general, the method of obtaining a pair of master and slave pseudo-terminals is to find a pseudo-terminal which is not currently in use. The master half of a pseudo-terminal is a single-open device; thus, each master may be opened in turn until an open succeeds. The slave side of the pseudo-terminal is then opened, and is set to the proper terminal modes if necessary. The process then forks; the child closes the master side of the pseudo-terminal, and execs the appropriate program. Meanwhile, the parent closes the slave side of the pseudo-terminal and begins reading and writing from the master side. Sample code making use of pseudo-terminals is given below; this code assumes that a connection on a socket `s` exists, connected to a peer who wants a service of some kind, and that the process has disassociated itself from any previous controlling terminal.

```
gotpty = 0;
for (c = 'p'; !gotpty && c <= 's'; c++) {
        line = "/dev/ptyXX";
        line[sizeof("/dev/pty")-1] = c;
        line[sizeof("/dev/ptyp")-1] = '0';
        if (stat(line, &statbuf) < 0)
                break;
        for (i = 0; i < 16; i++) {
                line[sizeof("/dev/ptyp")-1] = "0123456789abcdef"[i];
                master = open(line, O_RDWR);
                if (master > 0) {
                        gotpty = 1;
                        break;
                }
        }
}
       syslog(LOG_ERR, "All network ports in use");
         exit(1);
}

line[sizeof("/dev/")-1] = 't';
slave = open(line, O_RDWR);     /* slave is now slave side */
if (slave < 0) {
        syslog(LOG_ERR, "Cannot open slave pty %s", line);
        exit(1);
}

ioctl(slave, TIOCGETP, &b);     /* Set slave tty modes */
b.sg_flags = CRMOD|XTABS|ANYP;
ioctl(slave, TIOCSETP, &b);

i = fork();
if (i < 0) {
        syslog(LOG_ERR, "fork: %m");
        exit(1);
} else if (i) {         /* Parent */
        close(slave);
        ...
} else {                 /* Child */
        (void) close(s);
        (void) close(master);
        dup2(slave, 0);
        dup2(slave, 1);
        dup2(slave, 2);
        if (slave > 2)
                (void) close(slave);
        ...
}
```

5.6 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 = htonl (INADDR_ANY);
sin.sin_port = htons (MYPORT);
bind(s, (struct sockaddr*)&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:

```
hp = gethostbyname(hostname);
if (hp == NULL) {
        ...
}
bcopy(hp->h_addr, (char *) sin.sin_addr, hp->h_length);
sin.sin_port = htons(0);
bind(s, (struct sockaddr *) &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 Internet port number in the privileged range the `rresvport` library routine may be used as follows to return a stream socket in with a privileged port number:

```
int lport = IPPORT_RESERVED - 1;
int s;
s = rresvport(&lport);
if (s < 0) {
        if (errno == EAGAIN)
                fprintf(stderr, "socket: all ports in use\n");
        else
                perror("rresvport: socket");
        ...
}
```

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:

```
int     on = 1;
 ...
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
bind(s, (struct sockaddr *) &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).

5.7 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.

Consequently, the ability to send broadcast packets has been limited to sockets which are explicitly marked as allowing broadcasting. Broadcast is typically used for one of two reasons: it is desired to find a resource on a local network without prior knowledge of its address, or important functions such as routing require that information be sent to all accessible neighbours.

To send a broadcast message, a datagram socket should be created:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

The socket is marked as allowing broadcasting,

```
int     on = 1;
setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof (on));
```

and at least a port number should be bound to the socket:

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

The destination address of the message to be broadcast depends on the network(s) on which the message is to be broadcast. The Internet domain supports a shorthand notation for broadcast on the local network, the address INADDR_BROADCAST

(defined in <netinet/in.h>. To determine the list of addresses for all reachable neighbours requires knowledge of the networks to which the host is connected. Since this information should be obtained in a host-independent fashion and may be impossible to derive, a method of retrieving this information from the system data structures is provided. The SIOCGIFCONF ioctl call returns the interface configuration of a host in the form of a single ifconf structure; this structure contains a 'data area' which is made up of an array of of ifreq structures, one for each network interface to which the host is connected. These structures are defined in <net/if.h>. The actual call which obtains the interface configuration is:

```
struct ifconf ifc;
char buf[BUFSIZ];

ifc.ifc_len = sizeof (buf);
ifc.ifc_buf = buf;
if (ioctl(s, SIOCGIFCONF, (char *) &ifc) < 0) {
        ...
}
```

After this call buf will contain one ifreq structure for each network to which the host is connected, and ifc.ifc_len will have been modified to reflect the number of bytes used by the ifreq structures.

For each structure there exists a set of 'interface flags' which tell whether the network corresponding to that interface is up or down, point to point or broadcast, etc. The SIOCGIFFLAGS ioctl retrieves these flags for an interface specified by an ifreq structure as follows:

```
struct ifreq *ifr;

ifr = ifc.ifc_req;


for (n = ifc.ifc_len / sizeof (struct ifreq); --n >= 0; ifr++) {
        /*
         * We must be careful that we don't use an interface
         * devoted to an address family other than those intended;                */
        if (ifr->ifr_addr.sa_family != AF_INET)
                continue;
        if (ioctl(s, SIOCGIFFLAGS, (char *) ifr) < 0) {
                ...
        }
        /*
         * Skip boring cases.
         */
        if ((ifr->ifr_flags & IFF_UP) == 0 ||
            (ifr->ifr_flags & IFF_LOOPBACK) ||
            (ifr->ifr_flags & (IFF_BROADCAST | IFF_POINTTOPOINT)) == 0)
                continue;
```

Once the flags have been obtained, the broadcast address must be obtained. In the case of broadcast networks this is done via the SIOCGIFBRDADDR ioctl, while for point-to-point networks the address of the destination host is obtained with SIOCGIFDSTADDR.

```
struct sockaddr dst;
 if (ifr->ifr_flags & IFF_POINTTOPOINT) {
 if (ioctl(s, SIOCGIFDSTADDR, (char *) ifr) < 0) {
                ...
        }
        bcopy((char *) ifr->ifr_dstaddr, (char *) &dst, sizeof (ifr->ifr_dstaddr));
} else if (ifr->ifr_flags & IFF_BROADCAST) {
        if (ioctl(s, SIOCGIFBRDADDR, (char *) ifr) < 0) {
                ...
        }
        bcopy((char *) ifr->ifr_broadaddr, (char *) &dst, sizeof (ifr->ifr_broadaddr));
}
```

After the appropriate ioctl's have obtained the broadcast or destination address (now in dst), the sendto call may be used:

```
sendto(s, buf, buflen, 0, (struct sockaddr *)&dst, sizeof (dst));
}
```

In the above loop one sendto occurs for every interface to which the host is connected that supports the notion of broadcast or point-to-point addressing. If a process only wished to send broadcast messages on a given network, code similar to that outlined above would be used, but the loop would need to find the correct destination address.

Received broadcast messages contain the senders address and port, as datagram sockets are bound before a message is allowed to go out.