1 The Network File System

This chapter provides an overview of the DG/UX™ system implementation of Sun Microsystems' Network File System (NFS®). Advanced users may want to skip to the section "How NFS works," later in this chapter.

NFS is a facility for sharing files in a heterogeneous environment of machines, operating systems, and networks. File sharing is accomplished by allowing two or more machines to access a single disk. NFS allows users to access remote directories and files as if they were local.

NFS allows you to mount a file system from another machine and access its contents. It also allows other systems access to file systems on your machine. The system administrator must export (make available) those file systems that other systems and their users will access. The system administrator for the systems that have been granted access must then mount the exported file systems.

After a file system has been exported and mounted on a remote machine, users on the remote machine can use any file system commands, such as **cd**(1), to access the mounted file system and its contents. NFS, however, does not allow these users to log into the machine where the file system resides. Nor can these users access file systems that have not been exported. While these users do not need a profile on the machine where the file system resides, they must belong to the group that is permitted access.

> **IMPORTANT** The Network Information Service (NIS) is an optional network service that helps the system administrator keep administrative files on networked machines consistent. You do not need to install and set up NIS to use NFS. NIS is a centralized, read-only database that manages such information as passwords and host addresses for an entire domain over a network. NIS contains all the information usually kept in the following files in /**etc**: **passwd**, **hosts**, **group**, **networks**, **protocols**, **netgroup**, **services**, **bootparams**, **aliases**, **rpc**, **ethers**, **netmasks**, **netid**, **publickey** and **ypservers**. Chapter 3 discusses NIS in more detail.

How NFS allows file sharing

In a network environment, sharing programs and data can sometimes be tedious. Files either must be copied to each machine where they are needed, or users must log in to the remote machine owning the required files. Network logins can be time–consuming, and maintaining multiple copies of a file can become confusing as changes are made to separate copies. NFS solves these problems by using a distributed file system. This strategy permits users on one machine access to files on a remote system without having to log in to the remote system. Thus, users can avoid time–consuming logins and a single copy of the shared file can be maintained.

NFS provides a distributed file system by allowing any machine running NFS to act as an NFS client and/or NFS server. Machines requesting resources are called NFS *clients*, while machines providing services are called NFS *servers*. A server machine makes designated local file systems available. Client machines can mount and access these remote file systems as if they were local file systems.

After NFS has been set up and configured, users can access the files they require without knowing on which machine the file resides. To the user, there appears to be no difference between reading or writing a file on a local disk and reading or writing a file on a disk elsewhere on the network.

The network services concept

A distributed file system requires an architecture that provides features without disturbing the software environment. NFS accomplishes this by providing additional capability through server processes that work closely with the operating system, rather than by adding code to the operating system itself. That is, network services use a set of protocols for data exchange rather than integrate network features into the operating system. These protocols can be easily extended and are described in later chapters.

NFS is a standard for the exchange of data between different machines and operating systems that promotes multi–vendor and multiple operating system computing solutions. NFS is not a distributed operating system; it is an interface that allows a variety of operating systems and machines to play the role of client or server.

Maintaining service when a server crashes

The file server protocol is designed so that client systems continue to operate even when the server must be rebooted. Should a client fail, the server (or server administrator) does not need to take any action. Should a server or the network fail, applications on client systems continue trying to execute NFS operations until the server or network is restored.

This resilience is especially important in a complex network of heterogeneous systems. Many systems may not be controlled by an operations staff, or may be rebooted without warning. NFS continues to function after reboot without requiring special recovery operations on the server.

Getting the best use of NFS

You can configure NFS to run efficiently on the machines installed at your particular site. For example, configuring servers with large, high-performance disks and clients with minimal disk storage may yield better performance at lower cost than having many machines with small disks. Furthermore, you can distribute the file system data across many servers and get the added parallelism without losing transparency: NFS remains transparent regardless of the number of servers to which a client is connected. In the case of read-only files, copies can be kept on several servers to prevent bottlenecks.

NFS is integrated into the DG/UX system kernel when you set up the system. NFS requires the DG/UX™ TCP/IP product, which must be built into the kernel as well. After these products are built into the kernel, DG/UX systems supporting NFS can be either clients or servers. You may then access any other system that supports either a client and/or server interface.

A sample computing environment

This section depicts a typical NFS computing environment on a DG/UX system, one comprising an AViiON® system, an AViiON station, and a Sun Workstation®. Such an environment may resemble that illustrated in Figure 1-1.

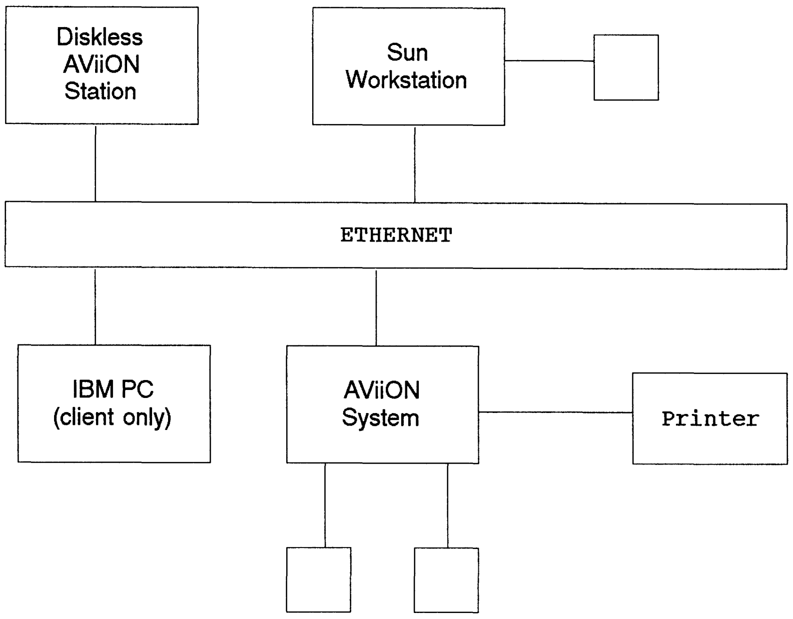

Figure 1–1 Workstation network

Through NFS, all disks on the network become available as you need them. Individual computers can be granted access to all information residing anywhere on the network on hosts supporting NFS.

Understanding NFS terms

This section explains some commonly used NFS terms.

Application
A program or set of programs that runs on a client or server.

External Data Representation (XDR)
A set of library routines that allow C programmers to describe arbitrary data structures in a machine–independent fashion. Data for remote procedure calls are transmitted using the XDR standard.

File system
A data structure residing on disk. Each file system comprises one or more files.

File system data
The data comprising the file system's structure and contents.

File system operations
The code implementing the operations of a file system or the operations a user or administrator perform on a file system.

Local file
A file on disk that is physically connected to the same computer.

Mount point
The directory in which a file system is logically located.

NFS client
: A computer that requests file system resources provided by servers.

NFS server
: A computer that provides file system resources to the clients.

Remote file
: A file on a disk that is physically connected to a computer other than the local computer using the file.

Remote procedure call (RPC)
: A facility that 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.

User
: A person logged in to a client or server computer.

How NFS works

System administrators must know how to set up the NFS server machine so that client workstations can mount all the necessary file systems.

The administrator makes file systems available for remote mounting by placing entries identifying the file systems in the **/etc/exports** file, and then exports them using **exportfs**(1M) or by using the **sysadm** function **addfs**. (For more information on **sysadm** or **addfs**, consult *Installing the DG/UX™ System* and *Managing the DG/UX™ System*.)

Here is an entry from a sample **/etc/exports** file for a typical server machine:

```
/usr/server -access=staff
```

The pathnames specified in **/etc/exports** must be directories that exist on local disks. Often, these directories are the mount–point pathnames of a local file system.

You can give a machine (or group of machines) access to these directories by listing the machine's hostname after the file system. NIS allows you to specify a group of machines under a single netgroup name. Thus, if you have a group of machines with the netgroup name **staff**, remote mounts are limited to machines that are members of the netgroup **staff**. (See **netgroup**(5) and **hostname**(4) for format details.) At any one time, the system administrator can execute the **showmount**(1M) command to see which local file systems have been remote–mounted by other systems.

To enable the use of NFS, your system administrator first must prepare the file system holding the file to be accessed. This action involves entering the file system and server names in certain administrative files. Once the file system has been prepared, you simply make sure the file system is mounted before you use it. The next section gives an example of how you do this.

The system administrator must have appropriate privilege to enable NFS. On a traditional DG/UX system, appropriate privilege is granted by having an effective UID of 0 (root). On a system with DG/UX information security, appropriate privilege is granted by having one or more specific capabilities enabled in the effective capability set of the user. The various roles may have different system prompts. See **cap_defaults**(5) for the default capabilities for this command.

IMPORTANT The conventions for a system prompt on a traditional DG/UX system are:

# for a superuser, and

% for others.

Mounting a remote file system

Suppose you want to view some on-line manual pages that are not available on your principal server machine called **victor** but are available on a machine called **docserv**. Mount the directory containing the manuals as follows:

```
victor# /etc/mount docserv:/usr/catman /usr/catman
```

You must be superuser to issue this command and the directory **/usr/catman** must exist on your local machine. Note that the file system will remain mounted until you issue a **umount** or reboot the system.

Now you can use the **man**(1) command from **victor**, which gives you information about other commands. Run the **df**(1M) command after you've mounted the remote file system. The output from this command will show the amount of free space on all mounted file systems, including the newly-mounted file system **/usr/catman**.

Provided that **/lib** and **/usr/server** already are remotely mounted, the output is similar to the following:

```
victor# df
/                (/dev/dsk/root      ):     29538 blocks    5003 files
/usr             (/dev/dsk/usr       ):     19637 blocks   23805 files
/lib             (docserv:/lib       ):     17935 blocks       0 files
/usr/server      (docserv:/usr/server):     14685 blocks       0 files
/usr/catman      (docserv:/usr/catman):    138685 blocks       0 files
```

Figure 1-2 below diagrams this configuration. In this figure, ellipses represent machines, and boxes represent remote file systems. At the top of the diagram are the file system layouts for the two machines, **victor** and **docserv**, before **victor** mounts/**usr/catman**. Immediately below these file system layouts, the diagram shows **victor**'s file system layout after **/usr/catman** has been mounted.

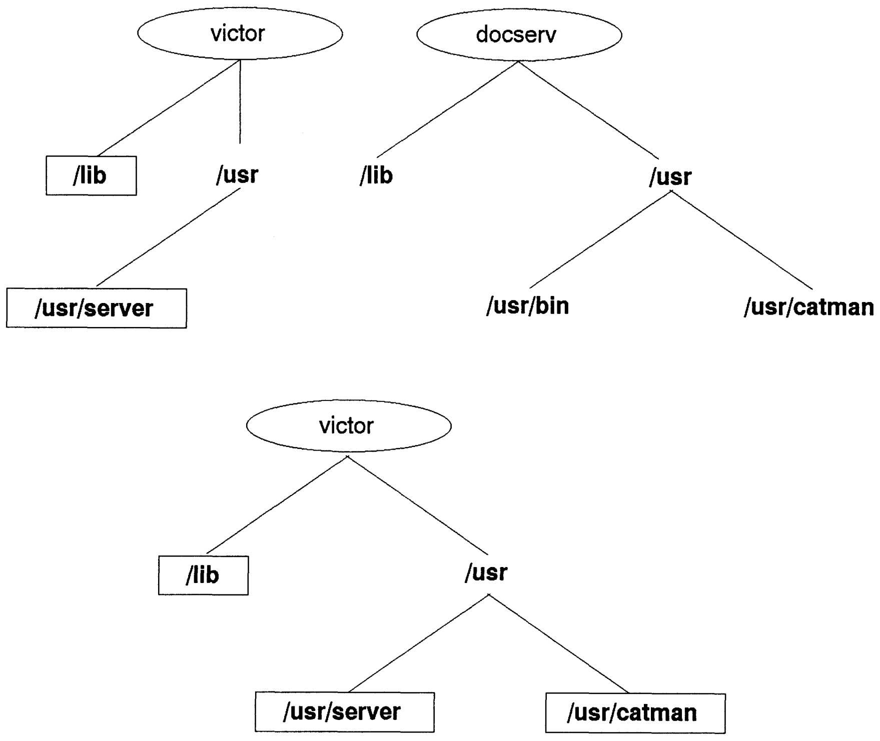

Figure 1-2 Remote file system mount

Exporting a file system

Suppose that you and a colleague collaborate on a programming project. The source code is on your machine, in the directory **/udd/proj**. Your colleague cannot remote-mount your directory until you export it. Until you have exported your directory, attempts to remote-mount it fail, displaying a "permission denied" error message.

To export a directory, you must become superuser and either use the **sysadm** function **addfs** or edit the file **/etc/exports** (see **exports**(5) for details on export file format). If your colleague is on a machine named cohort, you could put the following line in **/etc/exports**:

```
/udd/proj -access=cohort
```

After editing the **/etc/exports** file, you must execute the **exportfs**(1M) command. This command maintains the file **/etc/xtab**, which lists directories currently exported. You can run this command by typing the following in the shell:

```
# /usr/bin/exportfs -a ↵
```

In the absence of the **access=cohort** option, users with superuser privileges on any client machine on the network can remote–mount your directory, **/udd/proj**. The NFS mount request daemon **mountd**(1M) reads the **/etc/xtab** file, if necessary, when it receives a request for a remote mount. Now your colleague can remote–mount the file system on *cohort* by typing:

```
cohort# /etc/mount yourmachine:/udd/proj /udd/colleague/proj ↵
```

Your colleague may now change his or her working directory to the source directory by typing:

```
cohort% cd /udd/colleague/proj ↵
```

Since both you and your colleague now can access files in your project directory, you may want to use a source code control system such as SCCS or RCS (see *UNIX Software Development Tools*).

More guidelines for using NFS

An NFS server can be a client of another NFS server, but will not act as an intermediary between a client and another server. Thus, you must mount the file system from the host that owns the disk. A client may query the server for its remote mounts, then attempt to make similar remote mounts. To query the server, use the command **showmount -e** *server_name*.

NFS does not support all DG/UX file system operations. For example, the special file abstraction of devices is not supported for remote file systems. Thus, only disk devices are accessible through NFS.

NFS is a stateless service. The term *stateless*, when applied to a client, means that it alone is responsible for completing work. When applied to a server, the term means it need not remember anything from one client call to the next.

Finally, due to the use of data caches to enhance NFS performance, file updates are not always instantaneous. That is, the completion of a **write** call to a file accessed through NFS does not guarantee that the **write** call is reflected immediately upon subsequent **read** operations executed from a remote machine. Typically the delay is less than 30 seconds.

IMPORTANT A **close** call immediately forces all locally cached data to the server's disk.

Useful tips

The following information can make using NFS more simple and efficient.

- NFS assumes that user name to **uid** and group name to **gid** mappings are identical on all machines running with NFS. If not, users may lose access over NFS that they have locally, and may obtain access to files that they are not meant to access. To prevent this, you can manually ensure that /etc/**passwd** and /etc/**group** have the same mappings on all systems, or you can use the Network Information Service (NIS) to manage the propagation of **passwd** and **group** data. (See Chapters 3 and 4 for more information on NIS.)

  IMPORTANT You do not have to run NIS to use NFS.

- To avoid confusion and enhance application portability, system administrators should establish similar file hierarchies on systems linked with NFS. See the section entitled "A typical NFS installation" in Chapter 2 for an example.

- If an NFS server is unavailable, commands referencing that system hang, and diagnostic messages appear on the system console. Hard-mounted file systems are file systems in which a program attempts to communicate with a server until the server responds, regardless of whether the server is slow or down. When you use hard-mounted NFS file systems, commands referencing the downed server continue from where they left off when the server comes back up. Alternatively, you may mount remote file systems as interruptible so that you may elect to interrupt or kill the hung processes. You should kill or interrupt hung processes only as a last resort, since file data may be lost as a result. See **mount**(1M) for details of the hard and interruptible options. Writing to soft-mounted file systems may result in data loss.

End of Chapter

2 Administering ONC/NFS

This chapter provides an overview of the installation and administration of the services available on the DG/UX Open Network Computing (ONC) Network File System (NFS) and optional Network Information Service (NIS). It also provides information about maintaining and troubleshooting the ONC/NFS service. Although NIS is an optional service, we recommend its use in environments that allow a large amount of shared file access. Chapter 3 and Chapter 4 discuss NIS in more detail.

In the examples throughout this chapter, we assume:

- that the system administrator has appropriate privilege on the system being addressed, and
- that the remote system's environment is DG/UX.

If the remote system is not a DG/UX system, commands and terminal functions may behave differently. Consult the documentation for that system if you encounter variations from the procedures discussed in this chapter.

Installing and setting up the ONC/NFS services: an overview

The installation and setup information presented in this chapter is intended only as background for a better understanding of the actual procedures. The detailed procedures for installing and setting up NFS and NIS appear in *Installing the DG/UX™ System.*

Before installing ONC/NFS you should read the DG/UX system release notice, which describes any changes to installation procedures that occurred after this manual went to print, and provides an installation checklist.

For purposes of discussion, this chapter classifies the installation and setup procedures as four phases:

Phase 1.	Planning the installation—Cautions you about disk space requirements and tells you what information you must collect before installing NFS.
Phase 2.	Loading the software—Outlines the steps involved in loading the ONC/NFS product files, manual pages, and release notice.
Phase 3.	Setting up NFS and NIS—Discusses the initialization of **rc** (run command) script links and the **/etc/nfs.params** file. It also provides information on booting the kernel.

Phase 4. Customizing NFS and NIS—Discusses the entries required for each of the system database files used by NFS and NIS.

Each of these phases is presented below.

Phase 1: Planning the installation

You must ensure that adequate disk space is available for NFS and (optionally) NIS. Before beginning the installation procedure, be sure that you have enough free space in the **/usr** file system. (Refer to the DG/UX system release notice accompanying this product for the number of free blocks required.) Also, you must collect certain information on NFS and NIS and predetermine the names of the files created during installation.

Information required for setting up NFS

To set up the NFS database files, you must compile a list of the NFS servers and clients you work with, including:

- A list of the remote host file systems you want to mount.
- A list of remote hosts that "own" the file systems.
- A list of remote host administrators you must consult to ensure that the desired file systems are exported.

Information required for setting up NIS

To install NIS on the host, you must supply the following information:

- Your own **hostname**.
- The NIS class (**master**, **server**, **client**) of your host.
- The **domainname** for your NIS domain.

NIS **master** servers must:

- Determine name(s) for the NIS domain(s).
- Develop a policy concerning the addition of new hosts to the NIS network.
- Coordinate the maintenance of the host database with the TCP/IP administrator.

NIS **server** or NIS **client** hosts must:

- Contact the NIS administrator and arrange to be added to the databases.
- Learn the name(s) of the NIS domains(s) you will be using.
- Get the network name and address of the NIS **master** host needed by clients.

If you are an NIS slave **server**, you must also:

- Be able to establish a connection with the NIS **master**. You can verify this condition by issuing the command:

```
# ping MASTER_NAME ↵
```

The message **MASTER_NAME is alive** should appear.

- Confirm with the NIS administrator that you are listed in the **ypservers** map on the master.

Phase 2: Loading the package(s)

To install ONC/NFS for the first time from the DG/UX release tape, you use the installation utility. You load the ONC/NFS package onto an *existing* DG/UX system with the **sysadm** program.

> **IMPORTANT** On the DG/UX release tape, NFS and ONC are distinct software packages. That is, the designator *NFS* refers to the stand-alone version of NFS: selecting **NFS** from the Sysadm Main Menu loads NFS only. The designator *ONC* refers to the remainder of the ONC suite, which includes NIS. Thus to load *both* NFS and NIS, you must select both *NFS* and *ONC* from the menu.

Consult *Installing the DG/UX™ System* for detailed, step-by-step installation instructions. For more information about the **sysadm** program and its many facilities, see *Managing the DG/UX™ System.*

Phase 3: Setting up NFS and NIS

After you have loaded the product files, you must set up (configure) NFS and NIS on your host.

> **IMPORTANT** Normally, NFS and NIS are set up separately. The Sysadm Main Menu, however, allows you to automate the loading and setup procedures for both products:

Setting up NFS

To set up NFS, you don't need to supply any special information: setup proceeds automatically when you select the NFS *Install* or *Setup* option. You will need to perform some final setup tasks after you build and boot the custom kernel. Setup for NFS comprises two tasks: 1) initializing the **rc** script links so the NFS daemons and file system mounts are started properly when the system changes to the appropriate run level; and 2) initializing the /**etc**/**nfs.params** file. Both steps are done for you automatically when you select the *Install* option from the Sysadm Main Menu to load NFS, or when you issue the command **sysadm** setup package and respond to the prompts.

If you are working in a graphics environment, you also can use the **xsysadm** facility to set up NFS; see the manual *Managing the DG/UX™ System* for more information about this program.

Setting up NIS

All classes of NIS hosts (master, slave, and client) must set up their system for NIS. By default, hosts are set up as NIS clients. You must: 1) initialize the **rc** script links and 2) initialize the **/etc/nfs.params** file. The **rc** script links guarantee that the proper NIS daemons are started when you change run levels to make NIS available. Both steps are done for you automatically when you select the *Install* option from the Sysadm Main Menu to load ONC (NIS), or when you issue the command **sysadm** setup package and respond to the prompts. You must, however, supply the NIS domain name, which is available from your network (NIS) administrator. After running **sysadm setup package** and rebooting the kernel, you'll have to run a few commands to complete the setup as an NIS master or server; see "Customizing NIS" below and Chapter 4 for more information.

If you are working in a graphics environment, you also can use the **xsysadm** facility to set up NIS; see the manual *Managing the DG/UX™ System* for more information about this program.

Booting the kernel and run level changes

After loading ONC/NFS and setting up NFS and/or ONC, you must build a kernel. Consult the manual *Installing the DG/UX™ System* for instructions. Once you have built the kernel you can reboot your system. The default **init** run levels behave as follows:

- Run level state 2 makes NIS services available to the network services (TCP/IP), which are also started in run level state 2. The NIS services made available to TCP/IP include the **hosts,networks**, **services**, and other databases typically managed by NIS.
- Run level state 3 starts the NFS daemons, runs **exportfs**, and mounts any NFS file systems listed in **/etc/fstab**.

When you are ready to boot the kernel, follow the same procedure as for any other kernel.

After NIS has been set up, your machine is an NIS client. If you want your machine to be an NIS *server*, proceed to run level state 3 and follow the instructions in the section "Phase 4: Customizing NIS," below. To set up your machine as an NIS *master*, perform the customization steps at run level state 1, then proceed to run level state 3.

You can verify that NFS is working correctly by mounting an NFS file system and trying to access its files. You can check NIS with any of the following commands:

- **ypwhich**, which returns the name of your NIS server.
- **domainname**, which returns your current NIS domain name.
- **ypcat ypservers**, which lists NIS server hosts.

Phase 4: Customizing NFS and NIS

After installing and setting up NFS and NIS, you can customize them by making entries in the system database files used by these services. See the appropriate online manual pages for detailed information about these entries.

Customizing NFS

At this point the installation of NFS is largely complete. All that remains is for you to adapt it to your working environment by making certain entries in the following database files:

- /etc/**exports**, which identifies the local file systems you want to make available to other hosts. If this file does not exist, other systems will be unable to mount your file systems. The file /etc/**exports.proto** is provided as a template.
- /etc/**fstab**, which lists the local and remote file systems you want **init** to mount automatically.
- /etc/**hosts**, which lists the remote hosts you want to access.

After setting up NFS, you can modify the NFS runtime characteristics by editing the /etc/**nfs.params** file. The **nfs.params** file contains the arguments and parameters used to establish the behavior of the NFS and NIS daemons. In most cases, the default values shipped with the product are adequate. (Refer to the discussions of **sysadm** and **xsysadm** utilities in *Managing the DG/UX™ System* for alternate methods of modifying the contents of the **nfs.params** file.)

The /etc/**nfs.params** file shipped with the product can be customized for your particular system. It contains two types of variables that control the way NFS and NIS are invoked and initialized whenever you change to an appropriate run level with **init**. These variables are as follows:

nfsserv_START	starts the daemons required for a machine to be an NFS server.
nfsfs_START	initializes the machine as an NFS client by automatically mounting the remote file systems listed in /etc/**fstab**.
_ARG	variables determine the parameters supplied to the daemons/services when they are started.
nfsd_ARG	defines the number of NFSD server processes available to service remote NFS client requests. The normal value is 8, although you may improve NFS performance on heavily loaded NFS servers by increasing this value to 12 or higher.

You must supply values for these variables for NFS to set up properly. Do this by editing the file **/etc/nfs.params** with **vi** or another text editor, or by using the **sysadm** or **xsysadm** programs discussed in *Managing the DG/UX™ System*. Appendix B contains a sample **nfs.params** file.

Customizing NIS

After setting up your machine as an NIS client, you can then make it an NIS master or slave server by following the procedures in Chapter 4. The next three sections provide overviews of how to set up a host as an NIS client, NIS master, and NIS slave server.

Host as an NIS client

To create a new NIS client, you must do the following on the client machine:

1. Note the possible alterations to the client's /**etc** database as discussed in the section "Altering an NIS client's files to use NIS services" in Chapter 4. Because some files may be missing or altered, it is not always obvious how the ASCII databases are being used. The escape conventions used within those files to force inclusion and exclusion of data from the NIS databases are documented in the manual pages **passwd**(5) and **group**(5).
2. Notice in particular that changing passwords in /**etc/passwd** by editing the file or by running **passwd**(1) affects only the local client's environment. To make a global password change, use **yppasswd**(1) or edit the source **passwd** file on the **master**, then run **make**(1) in /**etc/yp**; see **ypmake**(1M).

Host as a master NIS server

This section provides an overview of configuring an NIS master server. The setup procedures below are performed on the host you are establishing as the NIS master.

Before setting up a host machine as a master NIS server, make sure the following files in /**etc** are complete and up-to-date on your system: **passwd**, **hosts**,**group**, **networks**, **protocols**, **netgroup**, **services**, **bootparams**, **aliases**, **rpc**, **ethers**, **netmasks**, **netid**, and **publickey**. If you have decided on the contents of **netgroup**, set this file up now; see the **netgroup**(5) man page for more information. Otherwise, when you run **ypinit**(1M) the **netgroup** map will be empty.

Follow this procedure to create a new master server:

1. Shut down the system by entering the command:

   ```
   # shutdown -g0 -y ↵
   ```
2. Start run level 1 by entering:

   ```
   # init 1 ↵
   ```
3. Modify the following lines in the /**etc/nfs.params** file to read as follows:

```
domainname_ARG="new_domainname"
ypserv_START="MASTER"
yppasswdd_ARG="/etc/src/passwd  -m SRC_DIR=/etc/src/passwd"
```

IMPORTANT For domainname_ARG and yppasswdd_ARG, be sure to enter information that is appropriate for your system.

4. Set the domain name as follows:

   # domainname new_domainname ↵

5. Set the **hostname**. (By default, **/usr/sbin/init.d/rc.tcpipport** sets the **hostname**, using parameters in **/etc/tcpip.params**.)

   # hostname new_hostname ↵

6. Run **ypinit** with the **-m** switch. You are asked if you want the procedure to die at the first non-fatal error (recommended if you haven't done the procedure before), or to continue despite non-fatal errors.
7. You are then prompted for a list of other hosts that will also be NIS servers. Initially, this list comprises the set of NIS slave servers, any of which could become the NIS master server. Specify other hosts if you are setting up more NIS servers. Doing this can save you some work and incurs very little runtime penalty. But because there is some runtime penalty, do not name every host in the network.
8. Start run level 3 by entering:

   # init 3 ↵

For security reasons, you may want to restrict access to the master NIS machine to a smaller set of users than those defined by the complete **/etc/passwd** file; see **passwd**(5).

Host as an NIS slave server

Follow this procedure to create a new NIS server:

1. On both the slave server and master server machines, ensure that the **/etc/passwd** file provides an entry for the daemon. This entry must precede any other entries in either file that has the same *uid*; you should verify this condition since the setup process that creates the **/etc/passwd** file may reorder the entries. This entry must be of the form:

   daemon:*:1:1::/:

   The values of *uid* and *gid* do not have to be 1.

2. On the slave server machine, run the command **ypinit -s MASTER_HOST**. You are asked if you want the procedure to die at the first non-fatal error (recommended if you haven't done the procedure before), or to continue despite non-fatal errors.

After setting up NIS, you must modify the local host's NIS role (client, master server, or slave server) as defined by the **ypserv_START** variable by editing the **/etc/nfs.params** file. This file contains the arguments and parameters used to establish the behavior of NIS daemons.

Just as with setting up NFS, you must supply values for variables and arguments in the file /etc/**nfs.params**. The variables that must be set for NIS are as follows:

ypserv_START
Turns NIS services on or off and specifies the class of the host (master, server, or client). To turn NIS services off, set this variable to **false**. Set this variable to **MASTER** for an NIS master host, **SERVER** for an NIS server host, and **CLIENT** for an NIS client host.

domainname_ARG
Sets the NIS domain name for the system's default domain.

```
domainname_ARG="sam_kernel"
```

yppasswdd_ARG
Specifies the arguments for the **yppasswd** daemon which runs on the NIS master and services update requests from **yppasswd** commands executed on hosts in the domain. By default the setting is:

```
yppasswdd_ARG=" "
```

which is sufficient for NIS slave servers and clients. If, however, your host is the NIS master and your password file is kept in /**etc**, you should specify:

```
yppasswdd_ARG="/etc/passwd -m passwd"
```

Likewise, if your password is kept in an alternate such as /**etc**/**src**, you would specify:

```
yppasswdd_ARG="/etc/src/passwd -m SRC_DIR =/etc/src/passwd"
```

NIS also requires a + entry in the /**etc**/**passwd** and /**etc**/**group** files. The files shipped with the DG/UX system already contain these entries.

In general, you should remove duplicate entries in the local and NIS databases from the local database files. Your security concerns, network configuration, and user population affect this decision. See "How administrative files are consulted on an NIS network" in Chapter 3 for guidance in setting up /**etc**/**passwd** and /**etc**/**group**.

NIS servers and masters must run the **ypinit** command after setting up as clients and entering run level 3. The command **ypinit** is an interactive script that prompts you for required information. See the man page **ypinit**(1M) for more information.

System administration for the NFS service

This section discusses system administration for NFS. It describes how to set up machines to act as NFS servers and explains how to mount and use remote file systems.

Establishing a machine as an NFS server

NFS is implemented by both user and kernel code. The user programs **mount** and **mountd** initiate NFS service. On the server machines, daemon processes (usually kernel processes) provide NFS services to clients. Note that an NFS server can export only *its own* file systems.

The following tasks must be performed by the system administrator with appropriate privilege. To make a local file system available for mounting on remote machines, follow these steps:

Step 1 Verify that the NFS server daemons are running. Remote mount needs a **mountd** daemon and some number of **nfsd** daemons (see **nfsd**(1M)) to execute on NFS servers. In addition, a number (typically 8) of **biod** daemons should be executed on the NFS client (see **biod**). The daemons are invoked after system boot by the files **/usr/sbin/init.d/rc.nfsserv** and **/usr/sbin/init.d/rc.nfsfs** when run level 3 is entered.

Step 2 Add the file system to the **/etc/exports** file of the server machine. This file declares the file system available for remote mounting use. A user with appropriate privilege on the server machine should place the pathname of the directory you want to export in the file **/etc/exports**. Often, the directory will be the mount-point pathname of a local file system. See **exports**(5) for file format details.

For example:

```
/usr/global/bin                  # export to all hosts
/usr/new  -root=spike1:mars2     # give root access to these hosts
```

are possible entries for **/etc/exports**.

To access the file system, a client machine must be given access rights in the **/etc/exports** entry. Here **/usr/global/bin** is exported to all hosts. The **/etc/exports** entry may include a list of the client machines that can have access to the file system. See **exports**(5) for a full description of the format.

Step 3 Make sure the file system to be exported is mounted locally. If the file system is exported before it is mounted, remote NFS users will not be able to access it. This may happen if a file system fails to mount during system setup. The problem can be corrected by unexporting and re-exporting the file system. See **exportfs**(1M) for more information on these options.

Step 4 Execute the **exportfs**(1M) command with the options **-v** and **-a**. (The command **exportfs**(1M) with no options displays a list of currently exported directories located in the file **/etc/xtab**.) At this point, a client can mount or unmount file systems from the server machine.

The command **exportfs** updates the /etc/**xtab** file and some tables that are internal to the kernel. The /etc/**xtab** file lists directories that are currently exported. **mountd** looks in this file for information on how to mount specified file systems. It has a format that is identical to /etc/**exports**. The /etc/**exports** file's contents, however, are not altered by **exportfs**. *Do not edit* /etc/**xtab** *manually because this causes* **exportfs** *to malfunction*. The command **exportfs -a** is executed automatically by **rc** scripts upon changing to run level 3. Thus, any file system named in /etc/**exports** is exported automatically. The **exportfs** command can be issued manually at a later date to export additional file systems.

How to remote-mount a file system

You can mount any exported file system onto your machine under the following conditions: 1) you can reach the server over the network; 2) you have some **biod** processes running locally; 3) you are included in the /etc/**exports** list for that file system; and 4) **exportfs**(1M) was run on the server. While logged in as superuser on the machine on which you want to mount a file system, you use the **mount** command as follows:

mount [**-o** *options*] *server_name:/server_directory /local_directory*

For example, to mount a global bin (executables) file system from remote machine **elvis** on your directory **/usr/global/bin**, type:

```
# mount elvis:/usr/global/bin /usr/global/bin ↵
```

In the preceding example, the subdirectory **/usr/global/bin** must already exist and cannot be the current directory. To make sure you have mounted a file system correctly, use either /etc/**df**(1) or /etc/**mount**(1M) without an argument. Each of these commands displays the currently mounted file systems. Once mounted, the remote-mounted files appear as if they are local. *Note* that the files in your local directory cannot be accessed when a remote file system is mounted.

By placing an entry in the file /etc/**fstab**, you can automate the mounting of frequently-used file systems at startup; see **fstab**(4).

> **IMPORTANT** When running NFS, be aware that if an NFS server is unavailable, commands hang and diagnostic messages appear on the console. On AViiON workstations, diagnostic messages appear on your screen or, if you are running the X Window System, on your console window. The **mount** command's **-intr** option lets you interrupt most commands when they hang while attempting to access a server that is down. See the end of the section "Remote mount failure" under "Troubleshooting NFS," below, for more information on the interruptability feature.

Hard-mounted file systems are file systems in which a program continues trying to communicate with a server until the server responds, regardless of whether the server is slow or down. Hard-mounted file systems should be used whenever a remote file system is mounted as read-write, to ensure that no data is lost if the server crashes or is slow to respond. When using hard-mounted NFS file systems, you are able to kill hung processes.

Soft-mounted file systems allow a remote access to fail, returning an error to the calling program after a certain number of attempts have been made to contact the server. Soft-mounts should only be used with remote file systems that have been mounted as read-only.

A typical NFS installation

Viewing the output from certain **mount** invocations helps explain the operation of NFS on NFS clients. The output below illustrates the mounted file systems on a server and on one of its clients. The client file systems are mounted under **/usr/catman** and on the global directories **/udd** and **/pdd**. Also, **nfsclient** is a client of **nfsserver**.

```
nfsserver% mount ↵
/dev/dsk/root on / type dg/ux (rw)
/dev/dsk/usr on /usr type dg/ux (rw)
/dev/dsk/tmp on /tmp type dg/ux (rw)
/dev/dsk/projects on /projects type dg/ux (rw)
/dev/dsk/documents on /documents type dg/ux (rw)
/dev/dsk/sac_release on /pdd/sac/release type dg/ux (rw)
/dev/dsk/udd_doc on /udd/doc type dg/ux (rw)
/dev/dsk/usr_catman on /usr/catman type dg/ux (rw)
/dev/dsk/usr_nfsserver on /usr/nfsserver type dg/ux (rw)
nfsclient% mount ↵
/dev/dsk/root on / type dg/ux (rw)
/dev/dsk/usr on /usr type dg/ux (rw)
nfsserver:/pdd/sac/release on /pdd/sac/release type nfs
(rw,hard)
nfsserver:/udd/doc on /udd/doc type nfs (ro,soft)
nfsserver:/usr/catman on /usr/catman type nfs (ro,soft)
```

Meaningful file system and mount point names help provide a consistent environment for users and programs. Here **/udd** is the name given to the *u*ser *d*ata *d*irectory, and **/pdd** the name for the *p*roject *d*ata *d*irectory.

Name selections also can be used to provide information and limit contention for name space. For example (using the **/pdd/***hostname* directory naming convention), the name **/pdd/loach/foo** informs the user that the server is **loach**; if this server is known to be down, the user won't expect to be able to access the file systems on it. Only that host should use **/pdd/loach** in naming file system mount points. Using **/udd/***hostname*/*username* as the login directory and exporting **/udd/***hostname* lets users access their **.login** and data files when they log in to the remote host that has mounted **/udd/***hostname.* If, however, you expect a file system to be relocated to another machine in the future, do not use the *hostname* convention.

Troubleshooting NFS

This section offers some guidelines for performing NFS troubleshooting. It provides some suggestions and discusses problems related to mounting, hung programs, slow response, and architectural incompatibilities between NFS file systems and regular DG/UX file systems.

Before trying to solve an NFS problem, you should read the following manual pages: **biod**(1M), **exportfs**(1M), **mount**(1M), **nfsd**(1M), **showmount**(1M), **rpcinfo**(1M), **mountd**(1M), **fstab**(4), **mnttab**(4) and **exports**(5). While you don't have to understand the man pages fully, you should be familiar with the names and functions of the various daemons and database files.

Most problems involving NFS services lie in one of the following four areas:

1. Either the NFS access control policies do not allow the specified operation or architectural constraints prevent it.
2. The NFS client or NFS server is not operating.
3. The server or client environment is configured incorrectly.
4. The underlying network is not functioning.

When tracking down an NFS problem, remember that—as with all network services—the common points of failure are the server, the client, or the network itself. The troubleshooting strategy outlined attempts to isolate the component that isn't working.

Suggestions for troubleshooting NFS

When network or server problems occur, programs that access hard-mounted remote files fail in a different manner than those that access soft-mounted remote files. Programs attempting to access hard-mounted remote file systems continue making the request until the server responds. Such programs hang as long as the server fails to respond. In this event, NFS displays an appropriate message on the console.

On the other hand, soft-mounted remote file systems return an error after attempting an operation a specified number of times. The number of retries can be specified at mount time. On a soft-mounted file system, the message *Connection timed out* (ETIMEDOUT) appears when a program attempts to access a file located on a server that is not responding. (Because many UNIX programs do not check return conditions on file system operations, you may not see this error message when you access soft-mounted files. Nevertheless, the NFS error message "NFS server *name* not responding" appears on the console.)

If a client is having problems with NFS, first verify that the server is up and running a **mountd** daemon to service the remote mount requester. From the client, use the command **rpcinfo** with the following format:

```
% /etc/rpcinfo -p server_name
```

to check the server's status. If the server is up, a list of program, version, protocol, and port numbers similar to the following should appear:

```
[program, version, protocol, port]:

    100000    2       tcp       111   portmapper
    100000    2       udp       111   portmapper
    100005    1       udp       714   mountd
    100005    1       tcp       716   mountd
```

Having noted that **mountd** is program number *100005* using *udp* protocol, you can also use **rpcinfo**(1M) to check whether the **mountd**(1M) server is running:

```
% rpcinfo -u server_name 100005 1 ↵
```

If **mountd** is running, the following message should appear:

```
program 100005 version 1 ready and waiting
```

If **mountd** is not running, the following message appears:

```
rpcinfo:RPC:Timed out
program 100005 Version 1 is not available
```

See **rpcinfo**(1M) for descriptions of the program and protocols.

Remote mount failure

This section discusses problems resulting from mount requests. It identifies the steps involved in a mount request to help you narrow the problem to its likely source. The section then lists the error messages that may appear and suggests corrective actions.

The **mount**(1M) command can get its parameters either from the command line or from the file /**etc**/**fstab**. The example below assumes command line arguments, but the same troubleshooting techniques apply if /**etc**/**fstab** is used by the **mount -a** command.

Consider this sample **mount** request:

```
# mount krypton:/usr/src /krypton.src ↵
```

For a remote mount to succeed the following events must occur:

1. Local **mount** opens /**etc**/**mnttab** and confirms that this mount has not been done.
2. Local **mount** parses the first argument into the hostname **krypton** and remote directory name /**usr**/**src**.
3. Local **mount** determines the Internet address of the remote host **krypton**.
4. If NIS is running, **mount** calls the NIS binder daemon, **ypbind**, to determine which machine is running the NIS server. It then calls the **ypserv** daemon on that machine to get the Internet address of **krypton**. Otherwise, it reads the local /**etc**/**hosts** file to obtain the INTERNET address.
5. Local **mount** calls **krypton**'s **portmapper** to get the port number of **krypton**'s **mountd**(1M). The **portmapper** is a program that provides NFS client programs with the port number of the NFS server program.
6. Local **mount** calls **krypton**'s **mountd** and passes the directory argument /**usr**/**src** to it.
7. **krypton**'s **mountd** reads /**etc**/**xtab** (a file created by **exportfs**(1M)) and checks for the exported file system that contains /**usr**/**src**.
8. **mountd** expands the hostname and netgroup entries in the export list for /**usr**/**src**. It also links the local host's INTERNET address to its name.
9. If NIS is running, **krypton**'s **mountd** calls the NIS server **ypserv** to expand the hostnames and netgroups in the export list for /**usr**/**src** and to perform the address-to-name translation. Otherwise, **mountd** uses /**etc**/**hosts**.
10. **krypton**'s **mountd** gets the file handle for /**usr**/**src** and returns it to the client machine.
11. On the client machine, **mount** does a **dg_mount**(2) system call with the file handle and /**krypton.src**.
12. Local **dg_mount** checks whether the caller has appropriate privilege and whether /**krypton.src** is a directory.
13. Local **dg_mount** does a **statfs** NFS call to krypton's NFS server (**nfsd**).
14. Local **mount** opens /**etc**/**mnttab** and appends an entry to the file.

Any one of these steps can fail, some of them in several ways. The following discussion identifies the failures associated with specific error messages and suggests corrective actions for them:

- Operation not supported

  The system file has not been updated to make use of NFS, an optional product. Use a text editor to append the letters **NFS** (in uppercase) to the system configuration file, which is used to create the bootable DG/UX file. After editing the system file, rebuild the kernel and reboot.

- /etc/mnttab: No such file or directory

  The mounted file system table is kept in the file **/etc/mnttab**. This file must exist before the mount can succeed. It is recreated after each system boot but may have been inadvertently deleted. If so, type **true>/etc/mnttab** to create an empty **mnttab** file.

- mount: ... already mounted

  The file system you are trying to mount is already mounted; check if you can access the desired file system. An invalid entry for it may occur in **/etc/mnttab**. This error message may appear if you have just rebooted the system to single-user mode but the boot was only partially successful. (The client machine should be running at run level 3.) Clear any invalid entries in **/etc/mnttab** and try again.

- mount: ... Block device required

  You probably omitted the **krypton:** part of the command and have a local directory named **/usr/scr**. The **mount** command assumes you are doing a local mount unless it sees a colon in the file system name or the file system type is **nfs** in **/etc/fstab**. The message *No such file or directory* informs you if the local directory **/usr/scr** does not exist.

- mount: ... not found in /etc/fstab

  The argument you gave to **mount** was not in any of the entries in **/etc/fstab**.

  If **mount** is called with only one argument, it looks in **/etc/fstab** for an entry whose file system or directory field matches the argument. For example:

  # **mount /krypton.src** ↵

  searches **/etc/fstab** for a line that has a directory name field of **/krypton.src**. If it finds an entry such as:

  ```
  krypton:/usr/src /krypton.src nfs rw,hard 0 0
  ```

  it executes the mount as if you had typed the following:

  # **mount -o rw,hard krypton:/usr/src /krypton.src**

- /etc/fstab: No such file or directory

  Mount did not find the directory name in **/etc/fstab**. Create **/etc/fstab** with the associated entry and try again.

- ...not in hosts database

  The host lookup failed. If your system is not running NIS, this message means the hostname you gave to **mount** does not exist in the **/etc/hosts** file on your machine. If NIS is running, NIS could not find the hostname you gave it in the **hosts** map. First check the spelling and the placement of the colon (:) in your **mount**(1M) command line. Then check the **hosts** map by typing the following:

  # **ypmatch** *hostname* **hosts**

  If this fails, do not assume that NIS is not functioning properly. The most likely cause is that the hostname does not exist or is not in the database. In the latter case, the message "No such key in map" appears.

- mount: directory path must begin with /

  The second argument is the pathname of the directory that will be the mount point for the remote file system. This argument must be an absolute pathname starting at /.

- mount: ... server not responding: RPC: Port mapper failure – RPC:Time out

  Either the server you are trying to mount from is down or its port mapper is dead or hung. Try the command:

  # **rpcinfo -p** *server-name*

  A list of registered program numbers should appear. If not, the port mapper on the server should be killed and restarted. Note that restarting the port mapper requires that other RPC daemons on the server be killed and restarted. See **portmap**(1M).

  If your attempt to use **rlogin** to log in to the server fails, but the server is up, check your network connection by trying to **rlogin** to another machine. Also check the server's network connection. If the network connections are fine, use **telnet**(1C) to log in to the server.

- mount: ... server not responding: RPC_PROG_NOT_REGISTERED

  The **mount**(1M) command reached the port mapper but the NFS mount daemon (**mountd**) was not registered. Restart the **mountd** daemon on the server.

- mount: ...: Not a directory

  Either the remote or local path is not a directory. Check your spelling and verify both directories with the command **ls -ld**.

- mount: ...: No such file or directory

  Either the remote or the local directory doesn't exist. Check your spelling and verify both files or directories with the command **ls -ld**. The local mount point must exist.

- mount: permission denied

  Your machine name is not in the export list for an existing file system you want to mount from the server. You can display a list of the server's exported file systems and which hosts and netgroups may access them by using the **showmount** command with the following format:

  % **showmount -e** *server_hostname*

  If the file system you want is not in the list, or if your machine name or netgroup name is not in the user list for the file system, log in to the server and run the command **exportfs -va**. If the file system you want is not in the command's output or the file system isn't exported by the server, check the /etc/**exports** file; see **exportfs**(1M). If the file system is in /etc/**exports**, the **exportfs -va** command could not translate the line in the file, it could not find the file system, or the file system name was not a locally mounted file system. See **exports**(5) for more information.

- mount: ...: Permission denied

  This message indicates that some forms of authentication failed on the server. It could simply be that your system name is not in the export list (see above), or the server couldn't identify your system (**ypbind** dead), or the server doesn't recognize your system. If **mountd** cannot translate your address to a name, the **mount** command fails, even if the /etc/**exports** file specifies that anyone can mount the file system. If NIS is running, check the server's /etc/**exports** and **ypbind**. Check your hostname using **hostname**(1); if it does not match the name in /etc/**exports**, change it and retry the mount.

- mount: ...: RPC: Authentication Error

  This error message may result when attempting to mount a remote file system and the user's id contains more than eight groups (which some earlier versions of NFS support). To check, execute the **id**(1) command and verify the number of groups. If greater than eight, reduce the number to eight or less and retry. (Systems known to have this problem with NFS include those of Silicon Graphics, Inc.; ULTRIX™; and Sun™, versions 3.5 and earlier.)

- Must be root to use mount

  You have to perform the mount as **root** on your machine.

- 167 – Stale file handle error

  This message indicates one of two problems: 1) the file you are trying to edit is in a file system that has not been exported from the server or has been exported without permissions for reading and writing; or 2) the file system was exported properly but the file has been deleted.

  To correct the first problem, be sure that the file system containing the file you want has been exported. To do so, check the server's /etc/**exports** file for an entry for the file system. If the file system has been exported with **rw** (read and write) privileges, check the permissions on the file you are interested in. The permissions must allow reading and writing to the file.

  To correct the second problem, you must check for users on the server that may be working on the file you are interested in.

If programs hang while doing file-related work, you may see a console message indicating that the NFS server is not responding. This message indicates a problem with one of the NFS servers or with the network. Programs can also hang if an NIS server dies (see Chapter 4).

If many processes on your machine hang, check the status of the server(s) from which you have mounted. If one or more of them is down and your file systems are hard-mounted, the server will come back up and your programs will continue automatically: no files are lost.

If an NFS server for a soft-mounted file system dies, other work is not affected. Programs that time-out while trying to access soft-mounted remote files fail with the message **errno** ETIMEDOUT. If the program is writing data to the server, data may be lost. Work with your other file systems is not disrupted.

Use **ping** <*hostname*> to determine whether the low-level network protocol layers (used by NFS) can contact the server.

If all of the servers are running and can be accessed from another client, ask a colleague who also is using the server or servers if they are functioning properly. If more than one machine is having difficulty getting service, most likely the problem is with the server's NFS daemon, **nfsd**(4). Log in to the server and issue the command **ps** to learn if **nfsd** is accumulating CPU time. If not, the **nfsd** daemon processes should be killed and restarted. If this fails to correct the problems, the server should be rebooted.

If other users are accessing the server without difficulty, check your network connection and the connection to the server using a utility such as **ping**. You can check your network connection by using **rlogin** to log in to another machine. If you can log in to the server with **rlogin**, both machines are connected to the network. If **rlogin** logs you in to other machines but not to the server, the problem is with the server or with routing.

IMPORTANT DG/UX 5.4 supports the **mount**(1) command option **-intr** for file systems mounted by NFS. When the **-intr** option is specified, system calls that access files on NFS mounted file systems may return **EINTR** if one of the following signals is received (and a signal handler is invoked) while the call is in progress: **SIGINT**, **SIGQUIT**, **SIGHUP** or **SIGTERM**. These signals terminate a process or, in the case of command interpreters like **sh**(1), terminate a sub-command and return to keyboard input mode.

If the **-intr** option is not specified, only those signals that terminate the program (for example, **SIGKILL**) can interrupt system calls that require remote NFS access. This behavior differs from that of SunOS™, where program termination is only allowed when **-intr** is specified and one of the signals listed above is received.

DG/UX supports the **-intr** option for both hard- and soft-mounted file systems, providing a uniform programming environment. SunOS™ allows the **-intr** option for hard mounts only.

The advantage of using the **-intr** option for NFS mounts is that most commands can be interrupted when they hang trying to access a server that is down. The **-intr** option, however, may produce unexpected results for programs that do not expect systems calls to return **EINTR** for these signals.

- NFS file handle no longer valid

This error message appears when one user removes a file that is currently being accessed by another remote user. This situation can affect both data files and directories, including mounted directories. If a directory that is exported by an NFS server is removed (or is not available because it has not been mounted), this error message will be seen by all users attempting to remotely access the file system mounted on that directory. Attempts to unmount the directory from the server will also fail. Normally, the unmount simply pends until the problem on the server is fixed.

If the remote directory has been removed permanently or has been replaced by a copy of the directory (for example, when a new file system is created following the installation of a replacement disk drive), the client's unmount request will continue to fail forever. In this instance, a different method of mounting the new copy of the directory is required.

The preferred work-around for this problem is to rename the client's mount point for the remote directory. Then a new mount point can be created and the new remote directory can be mounted. For example:

```
 mv /mnt/remote_mount_point  /mnt/remote_mount_point.old
mkdir /mnt/remote_mount_point
mount remote_host:/mount_point  /mnt/remote_mount_point
```

The old mount point (**mnt/remote_mount_point.old**) can be removed after the next reboot of the NFS client.

A second work-around for this problem is to reboot the NFS client machine. This is necessary if the remote file system with which you are having difficulty is mounted on a local read-only file system such as **/usr** on a diskless OS client machine.

Slow response

If access to remote files seems unusually slow, type:

```
# ps -ef ↲
```

on the server to be sure it is not being disrupted by a runaway daemon or bad **tty** line. If the server seems fine and other users are getting good response, make sure the client's block I/O daemons are running; type **ps -e** and look for **biod**.

If **biod** is running, check your network connection. The **netstat -i** command tells you whether packets are being dropped. Also, the commands **nfsstat -c** and **nfsstat -s** can be used to check whether the client is doing lots of retransmitting or whether there are many bad calls. A retransmission rate of five percent is considered high. Excessive retransmission usually indicates a bad network board, a bad network tap, a mismatch between board and tap, or a mismatch between your network board and the server's board.

Some notes about networking on a UNIX system

The UNIX® operating system was not designed to accommodate networking applications. As a result, three problems must be addressed when using UNIX over high-performance networks:

1. UNIX does not yield to a higher authority like a network authentication server for critical information or services. As a result, some UNIX semantics are hard to maintain over the network. By default, root privileges are reserved for local system administrators. If you are accessing remote files, your root privileges do not exist across the network unless they are granted by the server.
2. Some UNIX execution semantics are difficult to implement using the NFS protocol. Who should open a file on a network: the local or remote system? With DG/UX ONC/NFS, the local system opens the file. A UNIX client machine cannot own an open file; therefore, a server can remove a client's open file.
3. A UNIX machine stops all its applications when it crashes. However, when a network node crashes—whether client or server—it should not kill all of its bound neighbors. Thus, NFS provides flexible behavior for hosts on the network that are down: a system of stateless protocols prevent a crashing server from bringing down its bound clients.

IMPORTANT A stateless protocol is one where a client is independently responsible for completing work, and where a server need not remember anything from one client call to the next.) When a server crashes, the connection established between client and server remains the same. There is no state to recover when the server comes back up. To the client, a crashed server that returns to service appears no different from a very slow server.

Clock skew in user programs

Because the clocks of the NFS server and client are not synchronized, timing problems may arise. Many programs operate with the assumption that a file could not have been created in the future. In a network environment, however, in which clocks are not synchronized, such an event may appear to happen. If possible, you should modify your applications to allow for this condition.

IMPORTANT **Make**(1) is particularly sensitive to clock skew. See **make**(1) for usage warnings concerning NFS.

End of Chapter

3 Administering NIS

ONC/NFS network services include the Network File System (NFS), discussed in Chapter 2, and the Network Information Service (NIS), discussed in this chapter. Each of these services is independent of the other; NFS can be used without NIS, and vice versa. This chapter provides an overview of NIS and its functions, and explains the commands for maintaining it.

What is the Network Information Service (NIS)?

NIS is a centralized, read-only database that helps system administrators maintain consistency among administrative files on networked machines. The information in the database is maintained by the system administrator on a single machine, called the *NIS master server*. This information is automatically distributed to local NIS servers, keeping files consistent for all networked machines.

NIS simplifies the administration of standard files located in subdirectory **/etc**. By default, NIS master and slave servers use NIS maps based on the following files: **/etc/passwd**, **/etc/group**, **/etc/networks**, **/etc/hosts**, **/etc/services**, **/etc/protocols**, **/etc/bootparams**, **/etc/ethers**, **/etc/aliases**, **/etc/rpc**, **/etc/publickey**, and **/etc/netid**.

A new file, **netgroup**, can be created and included in the NIS database. By default, the **ypservers**'s map stores the list of server hosts in the domain.

NIS provides the following:

- A single source for each map. Maps are hashed files derived from ASCII files in the directory **/etc**; see "The NIS map" later in this chapter. This single source is maintained on the NIS master server; copies of these maps are sent to each NIS server.
- A look-up service. It maintains a consistent set of distributed NIS maps for querying. Programs can request the value associated with a particular key, or all the keys, in a database. (Keys represent the particular database information contained in the NIS maps.) For example, the keys in the **/etc/hosts** map are the names of the hosts.
- Transparent service. Applications need not know the location of data or how it is stored. Instead, they communicate with a database server that maintains this information. After the database on the master NIS server has been updated, the information is automatically distributed to servers on the network.

- Redundant databases. Databases are fully replicated on several machines known as NIS servers. The servers' databases are updated only by the master server, ensuring consistency. Once information has been propagated any server may answer a request: the answer is the same everywhere.

Understanding NIS terms

Following are some common NIS terms.

Master NIS server	A host machine running the **ypbind**, **ypserv**, and **yppasswdd** processes. This machine stores the ASCII source files from which the NIS maps are derived. All updates to the database are made on this machine. It disseminates information to the other server machines using **ypmake** and **yppush**.
Slave NIS server	A host machine running the **ypbind** and **ypserv** processes. It provides services to other processes running on the same or different machines. Note that an NIS server is an NIS client as well.
NIS client	A host machine running the **ypbind** process. It makes use of NIS services that are on the same or different machines.
NIS domain	A directory in /**etc**/**yp**/<*domain_name*> that contains a set of NIS maps. It is a named set of NIS maps. Machines that have this directory as their default NIS domain share the data found in its maps. This applies only to servers (master and slave).
NIS map	A hashed file in **dbm**(3X) format derived from ASCII files in /**etc** such as **passwd**. **group**, **hosts**, **networks**. These files store sets of keys and associated values for use by NIS.

The following sections provides a brief overview of NIS and some concepts related to it. Chapter 4 discusses the management of NIS in more depth.

Overview of the Network Information Service (NIS)

DG/UX system releases without NIS provide each machine on the network with its own copy of /**etc**/**hosts**, a file containing the Internet address of each machine on the network. When a machine is added to the network, each networked machine must update its /**etc**/**hosts** file.

Because NIS contains network-wide databases such as /**etc**/**hosts**, only the master copy of the file /**etc**/**hosts** must be updated. Servers throughout the network store copies of the databases. When a given machine on the network wants to access /**etc**/**hosts**, it instead makes an RPC call to one of the servers to get the information it needs.

NIS can serve any number of databases. Normally these include files previously located in /**etc**, such as /**etc**/**hosts** and /**etc**/**passwd**. However, users can add their own databases to NIS.

NIS most commonly is used to administer user IDs in /etc/**passwd**. If, for example, each machine uses its own local copy of the /etc/**passwd** database, users **jones** and **smith** each could have identical id numbers provided they used separate systems. Because NFS uses the UNIX authorization scheme across the network, user **jones** would have the same access to **smith**'s files if they were remotely mounted on **jones**' system.

A common /etc/**passwd** database for all machines on the network provides a single point of administration, preventing users **jones** and **smith** from having the same ID. NIS provides a single point of administration, letting all machines access the most recent data, whether or not it is locally kept.

Because the NIS interface is implemented using RPC and XDR, the NIS service is available to non-Data General machines and operating systems other than UNIX. NIS servers do not interpret data; thus, new databases can take advantage of the NIS service.

The following sections discuss how NIS operates; identify the NIS files; and explain the terms *maps*, *domains*, *servers*, *clients*, *masters*, and *slaves* in some depth.

The NIS map

NIS server information is stored in files called *maps*. Each NIS map contains a set of keys and associated values. For example, the hosts map contains all hostnames on a network and their corresponding Internet addresses. The hostnames are the keys, and the Internet addresses are the associated values. Each NIS map has a unique map name that is used by programs to access its data. To make use of these maps, programs must know the format of the data they contain.

Currently, most maps are derived from ASCII files such as **passwd**, **group**, **hosts**, and **networks**. These files normally are located in /**etc**.

Map data is stored in files with **dbm**(3X) formats that are located in the directory /**etc**/**yp**/*your_domainname* on NIS server machines.

The NIS domain

An NIS domain is a subdirectory in /**etc**/**yp** containing a set of maps. The name of this subdirectory is the name of the NIS domain. Machines with the same default NIS domain share the data found in the domain's maps.

Each machine on the network belongs to the default domain established at boot time by the **domainname**(1) command in the /**etc**/**nfs.params** file.

The command **domainname**(1) displays the name of your default NIS domain. Note that NIS domains differ from both Internet domains and sendmail domains.

An NIS domain name is required for retrieving data from an NIS database. For example, if your NIS domain is **dg** and you need the Internet address of host **dbserver**, you can use the command **ypmatch** from the command line (see **ypmatch**(1)). You also can accomplish this with the **yp_match**(3) procedure for the value associated with the key **dbserver** in the map **hosts.byname** within the NIS domain **dg** (see **ypclnt**(3)).

An NIS server stores all the maps for an NIS domain in a subdirectory of **/etc/yp**. This subdirectory is named after the domain. In the preceding example, maps for the **dg** domain would be stored in **/etc/yp/dg**.

Master servers and slave servers

NIS server hosts and processes are either master or slave. For any map, one NIS server host or process is designated the master. All changes to the map should be made on the master machine. After determining which server is the master, do all database updates and builds there, not on slaves. These changes are then propagated from master to slaves.

Different maps can have different servers as master. Therefore, a given server may be a master to one map, and a slave to another map. Unless there is a compelling reason to split the domain's maps between several masters, let one host serve as the master for all maps created by the command **ypinit**(1M) in a domain.

This chapter assumes the straightforward case in which one server is the master for all maps in the database. This is the only machine whose database should be modified. The other servers are slaves whose data is periodically brought up-to-date with that of the master.

Servers and clients

Servers provide resources; clients consume them. A machine's processes determine whether it can be a client, a server, or both.

Table 3–1 shows the processes that determine clients and servers for NIS.

Table 3–1 NIS client and server processes

NIS Host	Process
client	**ypbind**
server	**ypbind, ypserv**
master	**ypbind, ypserv, yppush, yppasswdd**

Thus, an NIS server is also an NIS client, and an NIS master server is also an NIS client and an NIS server. Unlike NFS servers, NIS master and slave servers *do* require extra space and processing time on a host. The characteristics of NIS clients and servers follow:

NIS clients

- Allow as many clients as you want.
- Require no registration on master and slave servers.
- Run **ypbind** on demand (that is, upon entering run level 2 or 3).
- Use no extra disk space.

NIS slave servers

- Allow as many servers as you want, but each server requires processing time on the master server.
- Require registration by the master server.
- Run **ypbind** and **ypserv** on demand (that is, upon entering run level 2 or 3).
- Use disk space on each server to store copies of NIS maps.

NIS masters

- Allow only one master server per map.
- Run **ypbind**, **ypserv**, **yppush**, and **yppasswdd** on demand.
- Use disk space to store copies of maps and source data files for maps.
- Run **cron** and **ypxfr** periodically.

Commands for maintaining NIS

This section briefly describes NIS commands you can use to set up and edit maps and to perform other NIS-related functions. These commands are described in detail in the manual pages. (You can access manual pages on-line with the **man**(1) command.)

ypserv	Searches for information in its local database of NIS maps. **ypserv** is a server process that runs only on NIS server machines with a complete NIS database.
ypbind	Stores information that allows client processes on a single node to communicate with a particular **ypserv** process. **ypbind** runs on all machines using NIS services, both NIS servers and clients.
ypinit	Automatically constructs maps from files located in /**etc**, such as /**etc**/**hosts**, /**etc**/**passwd**, and others. **ypinit** also constructs initial versions of required maps that are not built from files in /**etc**; for example, **ypservers**. Use **ypinit** to set up the master NIS server and the slave NIS servers for the first time. Typically you do not use **ypinit** as an administrative tool for running systems.

ypmake	Builds the NIS database. **ypmake** can be used to create **dbm** databases for a particular NIS map or all maps that are out-of-date.
makedbm	Converts an input file to a pair of **dbm** files, which then become a valid NIS map. For example, **ypservers.dir** and **ypservers.pag** are both **dbm** files. You can use **makedbm** to build or rebuild maps not built from **/etc/yp/Makefile**. You can also use **makedbm -u** to disassemble a map to reveal the key-value pairs comprising it. The disassembled form is in the format required for input back into **makedbm**, and can be edited.
ypxfr	Moves an NIS map from one NIS server to another, using NIS itself as the transport medium. You can run **ypxfr** interactively, or periodically from a **crontab** file.
yppush	Tells a master NIS server process (**ypserv**) to direct its peer processes to set the master of the named map to that master server, and to get a new copy of the named map. If the host is not the master of the named map, the command will succeed, but no action will be taken by the NIS server. You run **ypserv** on the master NIS server.
ypset	Instructs a **ypbind** process (the local one, by default) to get NIS services for a domain from a named NIS server. *This command is not for casual use*. The **ypset** command can be used by an experienced system administrator to untangle network access: it is a specialized tool. It is useful for binding a client node which is not on a broadcast net, or is on a broadcast net that is not running an NIS server host. It also is useful for debugging NIS client applications: for example, where an NIS map exists only at a single NIS server host. See the **ypset**(1M) man page for more information.
yppoll	Asks any **ypserv** for the information it holds about a single map.
ypcat	Displays the contents of an NIS map. Use it when you do not care which server's version you are seeing. If you need a particular server's map, log in to that server using **rlogin** (or use **.rsh**) and use **makedbm**.
ypmatch	Prints the value for one or more specified keys in an NIS map. Again, you have no control over which NIS server's version of the map you see.
ypwhich	Identifies which NIS server a host currently is using for NIS services, or (with the **-m** option) which NIS server is master of a particular map.

How administrative files are consulted on an NIS network

NIS can serve any number of maps. Typically these include maps based on some files in /etc. NIS services make updating these files much simpler since each system administrator does not have to make the same change to every machine on the network. For example, on networks that do not run NIS, programs read the **/etc/hosts** file to find an Internet address. When the system administrator adds a new machine to the network, he or she must add an entry for this machine to the **/etc/hosts** files on *every* machine on the network. On the other hand, networks running NIS programs that need to consult **/etc/hosts** merely perform a remote procedure call to the NIS server to acquire the same information.

NFS and other programs do not consult the same system administrative files on a network employing NIS that they would on a network where NIS is not installed: they consult NIS maps instead. The following lists the administrative files consulted by programs using a network employing NIS.

/etc/passwd	Always consulted. If there are + or − entries, the NIS password map is consulted; otherwise, NIS is not used. See **passwd**(5).
/etc/group	Always consulted. If there are + or − entries, the NIS group map is consulted; otherwise, NIS is not used. See **group**(5).
/etc/services	Never consulted. The data formerly obtained from this file now is read from the NIS **services** map.
/etc/protocols	Never consulted. The data formerly obtained from this file now is read from the NIS **protocols** map.
/etc/networks	Never consulted. The data formerly obtained from this file now is read from the NIS **networks** map.
/etc/netgroup	Never consulted. The data formerly obtained from this file now is read from the NIS **netgroup** map.
/etc/bootparams	Never consulted. The data formerly obtained from this file now is read from the NIS **bootparams** map.
/etc/ethers	Never consulted. The data formerly obtained from this file now is read from the NIS **ethers** map.
/etc/hosts	By default, consulted only when booting with the **ifconfig** command in the **/usr/sbin/init.d/rc.tcpip-port** file. Subsequently, the NIS map is used instead. The file **/etc/svcorder** determines the order of address resolution; see **svcorder**(4).
/etc/aliases	Always consulted. Local aliases take precedence over those in the NIS database. See **aliases**(5).
/etc/netmasks	Never consulted. The data formerly obtained from this file now is read from the NIS **netmasks** map. (This file is not included with DG/UX; a map is built, however, if **/etc/netmasks** exists.)

/etc/rpc	Never consulted. The data formerly obtained from this file now is read from the NIS **rpc** map. See the man page **rpc**(5).
/etc/publickey	Never consulted. The data formerly obtained from this file now is read from the NIS **publickey.by-name** map.
/etc/netid	Never consulted. The data formerly obtained from this file now is read from the NIS **netid.byname** map.

Accessing information from hosts files

If NIS is running, the system queries the NIS host maps for host information. Otherwise, the system seeks this information in the local **/etc/hosts** file. (The file **/etc/svcorder** determines the order of address resolution; see **svcorder** (4).) Library routines such as **getpwent**(3), **getgrent**(3), and **gethostent**(3N) use NIS when it is available. When a program calls the library routine **gethostbyname**(3N), a single RPC call to a server retrieves the entry from the **hosts.byname** map. Similarly, **gethostbyaddr**(3N) retrieves the entry from the **hosts.byaddr** map. If NIS is not running, **gethostbyname** reads the **/etc/hosts** files, as usual.

You can configure NIS to search the TCP/IP Domain Name System (DNS) if it cannot locate a host entry in its own NIS maps. The TCP/IP Domain Name System supplies host information through a series of distributed Internet data base servers. If DNS is installed and operating, you can enable the Domain Name service from NIS by setting the INTERDOMAIN flag in the **hosts** map. Use the **-b** option with the **make** command to turn on the INTERDOMAIN flag. (See *Managing TCP/IP on the DG/UX™ System* for information on how to install the Domain Name System.)

Enable DNS by making the **hosts** map as follows:

make INTERDOMAIN=-b hosts ↵

You can also set the INTERDOMAIN flag by editing the **Makefile** in **/etc/yp**. Change the line "INTERDOMAIN=" to read "INTERDOMAIN=-b". This latter method enables the Domain Name System automatically whenever a change is made to the hosts map. When typing these commands, do not space before or after the equals (=) sign.

Accessing information from the passwd file

The system goes to the **/etc/passwd** file for password information. The **/etc/passwd** file can have lines that begin with a plus sign (+) or minus sign (–) that forces use of the NIS databases for those lines (see **passwd**(5)).

Accessing information from other NIS files

Of the files in /etc, **/etc/group** is treated like **/etc/passwd**, in that **getgrent**(3N) consults NIS only if instructed by the **/etc/group** file. See **group**(5). The files **/etc/netgroup**, **/etc/protocols**, **/etc/services**, **/etc/networks**, **/etc/bootparams**, **/etc/ethers**, **/etc/rpc**, **/etc/publickey**, **/etc/netmasks**, **/etc/netid**, and **/etc/ypservers** are treated like **/etc/hosts**; for these files, the library routines go directly to NIS without consulting the local files.

How the NIS network service works

After the system administrator has established the NIS files for each administrative database supported by NIS, certain administrative files in **/etc** are no longer accessed. The library routines that access those files (for example, **gethostent**(3N) and **getgrent**(3N)) now access the NIS service; see "Accessing information from hosts files," above. Consequently, programs employing these library routines use NIS implicitly.

How NIS handles naming conflicts

Imagine a company with two networks, each of which maintains a separate list of hosts and passwords. Within each network, usernames, numerical user IDs, and hostnames are unique. Duplication between these two hypothetical networks, however, does occur. Therefore, connecting these two networks would wreak havoc. The hostname, returned by the **hostname**(1) command and the **gethostname**(2) system call, may no longer uniquely identify a machine. To resolve this issue, the command **domainname**(1) and the system call **getdomainname**(2) have been added.

In this example, each of the two networks could be given a different domain name. This would allow the continued use of independent naming within the two domains. If a significant amount of networking between the two domains occurs, however, the domains should be merged into one. Networking services such as **telnet**(1C) often assume that the mappings of names to internal identifiers are universally consistent. When domains are merged, however, naming conflicts must be resolved manually.

How NIS stores data

The NIS data is stored in **dbm**(3X) format. Thus the database **hosts.byname** for the domain **dg** is stored as **/etc/yp/dg/hosts.byname.pag** and **/etc/yp/dg/hosts.byname.dir**. The command **makedbm**(1M) converts an ASCII file such as **/etc/hosts** into **dbm** files suitable for use by NIS. The system administrator on the master normally uses the makefile in **/etc/yp** to create new **dbm** files. This makefile in turn calls **makedbm**(1M).

IMPORTANT For all NIS database files, you can use the **ypcat**(1) program to view an entire database, and **ypmatch**(1) to search for a particular entry in a database. Note also that multiple maps are created from certain /etc files (***.byname**, ***.byaddr**, and so on) and the map suffix identifies the *key* used to make the **dbm** files. Thus, for example, the command **ypmatch 196 passwd.byuid** searches the **passwd.byuid** map for the entry for the user whose *uid* is 196.

Table 3–2 lists the NIS maps and their uses.

Table 3–2 NIS map files

NIS Map	Use
passwd.byname	Obtains passwd entries indexed by username.
passwd.byuid	Obtains passwd entries indexed by user ID.
group.byname	Obtains group entries by group name.
group.bygid	Obtains group entries by group ID.
hosts.byname	Translates hostname to Internet address.
hosts.byaddr	Translates Internet address to hostname.
netgroup.byhost	Obtains netgroup entries by hostname.
netgroup.byuser	Obtains netgroup entries by username.
ethers.byname	Translates ethernet address to hostname.
ethers.byaddr	Translates hostname to ethernet address.
rpc.bynumber	Obtains the name of a defined RPC program indexed by number.
networks.byname	Translates network name to Internet network number.
networks.byaddr	Translates Internet network number to network name.
protocols.byname	Translates protocol name to Internet protocol number.
protocols.bynumber	Translates Internet protocol number to protocol name.
services.byname	Translates Internet service names to port numbers.
ypservers	List of NIS servers in the domain.
bootparams	Obtains root and swap files for diskless client.
mail.aliases	Translates alias entries to mail addresses for users and groups.
publickey.byname	Obtains publickey and password-encrypted secret key by netnames.
netid.byname	Obtains user and group IDs by netnames.
netmasks.byname	Obtains network masks by network name.
netmasks.byaddr	Obtains network masks by network address.

How servers provide information

To become an NIS server, a machine must:

- contain the NIS databases, and
- be running the NIS daemon **ypserv**.
  The **ypinit**(1M) command takes a flag that indicates whether you are creating a master or a slave.

When updating the master copy of a database, an administrator can encourage the immediate propagation to all the slaves with the **yppush**(1M) command. This command distributes the information to all the slaves. The makefile in **/etc/ypfirst** executes **makedbm** to make a new database, then calls **yppush** to propagate the change throughout the network.

How clients obtain information

A machine that is not running an NIS server process does not contain any NIS data. Its NIS client process makes an RPC call to an NIS server process on a remote machine whenever it needs information from an NIS database. When a client boots and goes to run level 2 or 3, **ypbind**(1M) broadcasts and requests the name of an NIS server. Similarly, if a client has previously bound to a server and the client's server crashes, **ypbind** broadcasts a request for the name of a new server. The **ypwhich**(1) command displays the name of the server that **ypbind** currently points to.

Changing your password

To change data in NIS, you (typically the system administrator) must log in to the master machine and edit the appropriate databases there; **ypwhich**(1) identifies the master server. Because changing NIS passwords is a recurring task for regular users, the **yppasswd**(1) command has been provided for this purpose. Its user interface is identical to that of the **passwd**(1) command. The **yppasswdd**(1M) daemon on the NIS master server machine listens for requests and updates the **passwd** maps.

End of Chapter

4 Managing NIS

This chapter discusses NIS administration, troubleshooting, and security issues regarding the NIS environment. (Chapter 3 presents an overview of NIS and its terminology.)

NIS administration

This section explains how to perform the following administrative tasks:

1. Set up a master NIS server
2. Alter an NIS client's database to use NIS services
3. Set up a slave NIS server
4. Set up an NIS client
5. Modify individual NIS maps after installing NIS
6. Propagate an NIS map
7. Make new NIS maps after installing NIS
8. Add an NIS server
9. Change the master server to a different machine

Setting up a master NIS server

IMPORTANT This section assumes you have installed and set up *ONC* (that is, NIS) using the **sysadm** utility; these procedures are explained in *Installing the DG/UX™ System*.

Before you can set up the master NIS server, you must perform the following steps:

- Set the NIS domain name if it differs from the name selected for your network domain during installation with the **sysadm** utility.
- Set the hostname.

By default, **/usr/sbin/init.d/rc.ypserv** sets the **domainname** and **/usr/sbin/init.d/rc.tcpipport** sets the **hostname**, when you go to run level 2 or 3, using parameters in **/etc/nfs.params** and **/etc/tcpip.params**. Your machine should be at run level 1 before becoming a master; thus it maybe necessary to set the domain name and host name manually.

You use the **ypinit**(1M) command to establish an NIS database on an NIS server. It can be used to set up a master or a slave server; see the man page **ypinit**(1M) for more information. Before you run **ypinit**, the following files in /etc should be up-to-date: **passwd**, **hosts**, **ethers**, **group**, **networks**, **protocols**, **netgroup**, **bootparams**, **hosts.equiv**, **aliases**, **netmasks**, **rpc**, and **services**. Also, if you know how **/etc/netgroup** is going to be set up, set that file up before running **ypinit**. Otherwise, **ypinit** creates an empty **netgroup** map.

For security reasons, you may not want to use the contents of the files in **/etc** to create the NIS maps. If so, copy the fourteen files from **/etc** to another directory, for example **/etc/yp/dg_src**. Then edit these files to reflect the contents of the NIS maps. Remove the + entries from **group** and **passwd**. You also may want to remove **root** and other system ID's already in the **/etc/passwd** file.

To build the new NIS maps that do not use the files in **/etc**, you should change the **SRC_DIR** variable in **/etc/yp/Makefile**; in this example, you would change it to **SRC_DIR=/etc/yp/dg_src**. It is important to set the **SRC_DIR** variable in the **Makefile** if you are using an alternate source directory, as this variable determines some **sysadm** values.

After performing these steps, you are ready to create a new master server. Become superuser and make **/etc/yp** the current directory. Then run **ypinit** with the **-m** option.

To create a new master server on an existing network, enter the directory **/etc/yp** on the new master server machine and execute the command **ypinit**. You are asked whether you want the procedure to die at the first non-fatal error (in which case you can fix the problem and restart **ypinit**; recommended if you haven't done the procedure before), or to continue despite non-fatal errors. In the latter event you can try to fix all the problems manually, or fix some, then restart **ypinit.**

ypinit prompts you for a list of other NIS servers. Initially, the master server is the only NIS server. You need not add any other hosts at this time, but if you know that you will be setting up more NIS servers, add them now. You will save yourself some work later, and there is little runtime penalty for adding them. Do not, however, name every host in the network.

You must edit the **/etc/nfs.params** file and change the value of **ypserv_START** to **MASTER**. Also, you should change **yppasswdd_ARG=""** to the appropriate entry (see **yppasswdd**(1M)). Appendix B contains a sample **/etc/nfs.params** file.

To initiate NIS services, go to run level 3. Now NIS starts up automatically from **/usr/sbin/init.d/rc.ypserv** whenever the server boots.

Altering an NIS client's files to use NIS services

Once you decide to run NIS at your site, all hosts on the network should access the NIS maps, not potentially outdated information in their local administrative files. You can enforce this policy by running a **ypbind** process on the client machine (including machines that may be running NIS servers), and by abbreviating or eliminating files traditionally implemented by NIS maps.

Table 4–1 lists these files.

Table 4–1 List of NIS map source files

/etc/hosts	**/etc/passwd**
/etc/group	**/etc/networks**
/etc/services	**/etc/netgroup**
/etc/aliases	**/etc/netmasks**
/etc/ethers	**/etc/netid**
/etc/protocols	**/etc/bootparams**
/etc/publickey	**/etc/rpc**

The following discussion provides some background information and explains the purpose of some of these files.

- **/etc/aliases**, the aliases file on the master NIS server, is used for the **mail.aliases** NIS map. Thus, the **/etc/aliases** files on the various hosts in a network become largely obsolete. Domain-wide aliases should ultimately be resolved into user names on specific hosts. The **sendmail** program first tries to locate the *username* in **/etc/aliases**; if that fails, it searches **mail.aliases**. Like local logins, local aliases for **sendmail** are allowed.
- **/etc/hosts** must contain entries for the local host's name, and the local loopback interface name. These are accessed at boot time when the NIS service is not yet available. After the system is running, and after the **ypbind** process is up, the **/etc/svcorder** file determines the order in which **/etc/hosts** and NIS are searched for address resolution (see **svcorder**(4)). An example of the hosts file for an NIS client named **ray** is:

```
127.0.0.1     localhost
192.9.1.87    ray  # stefania
```

- **/etc/passwd** should contain entries for the root username and the primary users of the machine, and the + escape entry to force the use of the NIS service. An additional entry of a daemon to allow file-transfer utilities to work is recommended. A sample NIS client's **/etc/passwd** file is similar to the following:

```
root::0:1:     Special Admin login:/:/sbin/sh
sysadm::0:0    Regular Admin login:/admin:/sbin/sh
daemon:*:1:1:  Daemon Login for daemons needing permissions:/:/sbin/sh
bin:*:2:2:     Admin :/bin:
sys:*:3:3:     Admin :/usr/src:
adm:*:4:4:     Admin :/usr/adm:/sbin/sh
uucp:*:5:5:    UUCP Login:/usr/spool/uucp:/usr/lib/uucp/uucico
nuucp:*:5:1:   UUCP Admin Login :/usr/lib/uucp:/sbin/sh
lp:*:6:2:      Printer:/usr/lib:/sbin/sh
mail:*:8:1:    Sendmail Login for mail delivery:/usr/mail/:/usr/bin/mail
sync::19:1:    Disk Update Login without password:/:bin/sync
yp:*:37:37:    YP Admin :/usr/etc/yp:/sbin/sh
nfs:*:38:38:   NFS Admin :/:/sbin/sh
ftp:*:39:39:   FTP guest Login:/var/ftp:/sbin/sh
nobody:*:65534:65534::/:
+:
```

The last line instructs the library routines to use the NIS service. Removing the last line in the **passwd** file disables NIS password access.

A program that calls **/etc/passwd** first opens the password file on your machine; it opens the NIS password file only if your machine's password file contains + (plus sign) entries. Earlier entries in the password file take precedence over, or mask, later ones with the same username or user ID. Therefore, note the order of the entries for root and for **sysadm** (which have the same user ID) and use it in your own file.

- **/etc/group** may be reduced to a single line:

```
+:
```

which forces all translation of group names and group IDs to be made by the NIS service. This is the recommended procedure.

How to set up a slave NIS server

The network must be working before you can set up a slave NIS server—in particular, you must be able to communicate with the master NIS server.

To create a new slave server, run **ypinit** with the **-s** option. You must be superuser when you run **ypinit**. Name a host already set up as an NIS server as the master. Ideally, the named host is the master server, but it can be any host that has its NIS database set up. The host must be reachable. The default domain name on the machine intended to be the NIS slave server must be set to the same domain name as the default domain name on the machine named as the master. Also, an entry for daemon must exist in the **/etc/passwd** files of both slave and master, and that entry must precede any other entries having the same user ID. **sysadm** creates **/etc/passwd**; make sure the order of the entries in your password file has not been altered. Note the example shown in the section above. You are not prompted for a list of other servers, but you do have the opportunity to choose whether or not the procedure quits at the first non-fatal error.

Edit the original files in accordance with the preceding section "Altering an NIS client's files to use NIS services" to insure that processes on the slave NIS server use the NIS services, rather than the local ASCII files. (That is, make sure the NIS slave server is also an NIS client.)

After **ypinit** sets up the the NIS database, type **/usr/etc/ypserv** to begin providing NIS services. Also, edit the /etc/**nfs.params** file to set **ypserv_START** to **ypserv_START=SERVER**. On subsequent reboots, NIS starts automatically from **/usr/sbin/init.d/rc.ypserv**.

How to set up an NIS client

To set up an NIS client, edit the local files as described above in "Altering an NIS client's files to use NIS services." If **/usr/etc/ypbind** is not running already, start it. With the ASCII databases of /**etc** abbreviated and **/usr/etc/ypbind** running, the processes on the machine are clients of the NIS services. At this point, an NIS server must be available; processes hang if an NIS server is unavailable while **ypbind** is running. Note the possible modifications to the client's /**etc** database as discussed above in the earlier section "Altering an NIS client's files to use NIS services." Because some files may be missing or specially altered, it is not always obvious how the ASCII databases are being used. The escape conventions used within those files to force the inclusion or exclusion of data from the NIS databases are found in the following man pages: **passwd**, **hosts.equiv**, and **group**. In particular, note that changing passwords in /**etc/passwd** (by editing the file, or by running **passwd**) only affects the local client's environment. Change the NIS password database by running **yppasswd**.

Modifying NIS maps after installation

Databases served by NIS should always be modified on the master server. Databases expected to change most frequently, like /**etc/passwd**, may be changed by first editing the ASCII file, changing the directory to /**etc/yp**, and running **make.**

Non-standard maps (that is, maps specific to the applications of a particular vendor or site, but not part of Data General's release), or maps that are expected to change rarely, or maps for which no ASCII form exists (for example, files that didn't exist before you installed NIS) may be modified manually. The general procedure is to use **makedbm** with the **-u** option to disassemble the maps into a form that can be modified using standard tools (such as **awk**, **sed**, **or vi**). Then build a new version again using **makedbm**. This may be done manually in two ways:

- The output of **makedbm** can be redirected to a temporary file that can be modified and fed back into **makedbm**, or
- The output of **makedbm** can be processed within a pipeline that feeds directly into **makedbm**. This procedure is appropriate if the disassembled map can be updated by modifying it with **awk**, **sed**, or appending with the **cat** command.

Suppose you want to create a non-standard NIS map called **mymap**. You want it to comprise key-value pairs in which the keys are strings such as al, bl, cl, and so on ("l" stands for left) and the values are ar,br, cr ("r" stands for right). Either of two procedures may be used when you create new maps. One uses an existing ASCII file as input; the other uses standard input.

For example, suppose the ASCII file **/etc/yp/mymap.asc** has been created with an editor or a shell script on the machine **ypmaster** and **home_domain** is the subdirectory where the map is located. You can create the NIS map for this file by typing the following commands:

```
ypmaster% cd /etc/yp ↵
ypmaster% makedbm mymap.asc home_domain/mymap ↵
```

At this point you realize the map should include another key-value pair (dl, dr). Modify the map by first modifying the ASCII file; modifications made to the **dbm** files will be lost. Make the modification by:

1. Using the **cd**(1) command to go the directory **/etc/yp**.
2. Using **vi**(1), or another editor, to edit the ASCII file **mymap.asc**.
3. Using **makedbm**(1M) to update the map as follows:

```
ypmaster% makedbm mymap.asc home_domain/mymap ↵
```

When no original ASCII file exists, create the NIS map from the keyboard. (This example assumes the machine name is **ypmaster**, and the default domain is **home_domain**):

```
ypmaster% cd /etc/yp ↵
ypmaster% makedbm - home_domain/mymap ↵
al ar
bl br
cl cr
<ctrl-D>
```

When you need to modify that map, you can use **makedbm -u** to undo a **dbm** map and to create a temporary ASCII intermediate file, which can be edited using standard tools. For example, you can type the following:

```
ypmaster%  cd /etc/yp ↵
ypmaster%  makedbm -u home_domain/mymap > mymap.temp ↵
```

At this point you can modify "**mymap.temp**." The following commands create a new version of the database:

```
ypmaster% makedbm mymap.temp home_domain/mymap ↵
ypmaster% rm mymap.temp ↵
```

Unless you add non-standard maps to the database or change the set of NIS servers after the system is up and running, the preceding tasks can be accomplished with the **ypinit** and **/etc/yp/Makefile** commands. Whether you use the **Makefile** in **/etc/yp** or some other procedure, the goal is the same: a new pair of well-formed **dbm** files must reside in the domain directory on the master NIS server.

Propagating NIS maps

To propagate a map means to move it from place to place—in general, to copy it from the master NIS server to all slave NIS servers. Initially, **ypinit** copies all the maps from the master to a particular slave server, as described earlier in the section "How to set up a slave NIS server." After a slave NIS server has been initialized, updated maps are transferred from the master server by **ypxfr**. The **ypxfr** program runs on the slave, but is told to do so by the master. You can run **ypxfr** in three different ways: periodically using **cron**; with **ypserv**; or interactively. Examples of each method follow.

Running ypxfr with cron

Maps change at different rates; for instance, **protocols.byname** may not change for months at a time, but **passwd.byname** may change several times daily in a large organization. You can set up entries in a **crontab** file to periodically run **ypxfr** at a rate appropriate for any map in your NIS database. **ypxfr** contacts the master server and transfers the map only if the master's copy is more recent than the local copy.

To avoid having a **crontab** entry for each map, you can group several maps with approximately the same change characteristics in a shell script and run the shell script from a **crontab file**. Instead of directly editing the **crontab** file, you must edit a copy of the file and use the **crontab** command to give the new **crontab** file to the **cron** daemon.

Suggested groupings, mnemonically named, can be found in **/usr/etc/yp**: **ypxfr _1perhour**, **ypxfr _1perday**, and **ypxfr _2perday**. If these rates of change are inappropriate for your environment, you can easily modify or replace these shell scripts.

These same shell scripts should be run at each NIS slave server in the domain. Alter the exact time of execution from one server to another to prevent slowing the master. If you want the map transferred from a server other than the master, use the **ypxfr**'s **-h** option within the shell scripts. Use the **-s** option to transfer maps from another domain. Finally, maps having unique change characteristics can be checked and transferred by invoking **ypxfr** within the system **crontab** file, **/usr/spool/cron/crontabs/root**.

Running ypxfr with ypserv

The **ypxfr** program also gets invoked by **ypserv**, responding to a *transfer map* request. That request is made as an RPC call from **yppush**. **yppush** is run on the master NIS server. It tells the NIS map **ypserver** to generate a list of NIS servers in your domain. To each of the named NIS servers, it sends a *transfer map* request. **ypserv** makes a copy of **ypxfr**, invokes it with the **-C** flag, and passes it the information it needs to identify the map and to return a summary status to the initiating process **yppush**.

In the cases just discussed, **ypxfr**'s transfer attempts and their results can be captured in a log file. If **/etc/yp/ypxfr.log** exists, results are appended to it. The log file is not limited. To turn off logging remove the log file.

Running ypxfr as a command

Typically, you run **ypxfr** as a command only in exceptional situations. For example, you would run **ypxfr** as a command when setting up a temporary NIS server to create a test environment, or when quickly making an idle NIS server consistent with the other servers.

Making new NIS maps after installation

Adding a new NIS map entails copying the map's **dbm** files to the domain directory on each of the NIS servers in the domain. This process is described above in "Propagation of an NIS map." This section describes only how to get the proper files in place on both the master and the slaves so the propagation is successful.

After deciding which NIS server is the master of the map, modify **/etc/yp/Makefile** on the master server to simplify the rebuilding of the map. While specific modifications are too varied to describe here, in general an ASCII file is filtered through **awk**, **sed**, and/or **grep** to make it suitable for input to **makedbm**. Consult the existing **Makefile** as a source for programming examples. You should use the mechanisms already in place in **/etc/yp/Makefile** when deciding how to create dependencies that **make** will recognize; specifically, the use of "**.time**" files tells you when the **Makefile** was last run for the map.

To get an initial copy of the map, each server must execute the **ypxfr** command with the **-h**<*master*> option (see **ypxfr**(1M) for more information). If the map is available from some NIS servers but not all, unpredictable behavior from client programs results. The map must be globally available before clients begin to access it.

How to add a new NIS server to the original set

To add a new NIS slave server, start by modifying maps on the master NIS server. If the new server is a host that has not been an NIS server before, you must add the host's name to the map **ypservers** in the default domain. The sequence for adding a server named **ypslave** to **domain_name** is as follows:

```
ypmaster# cd /etc/yp/domain_name ↵
ypmaster# makedbm -u ypservers > /tmp/temp_file ↵
ypmaster# vi /tmp/temp_file ↵
ypmaster# makedbm /tmp/temp_file ypservers ↵
ypmaster# rm /tmp/temp_file ↵
```

Running the **makedbm** command with the **-u** option undoes the **ypservers dbm** file; that is, it converts it from **dbm** format so you can add the new hostname to the temporary file **temp_file**. Then, running the **makedbm** command with **temp_file** as the input file and **ypservers** as the output map converts **ypservers** back into **dbm** format.

You can set up the new slave NIS server's databases by copying the databases from NIS master server **ypmaster**. To do this, log in remotely to the new NIS slave and run the **ypinit** command as shown below:

```
ypslave# cd /etc/yp ↲
ypslave# ypinit -s ypmaster ↲
```

(To become a slave, the system first needs to be running as a client: that is when the **domainname** is set.) To verify that the **ypservers** file is correct (since no ASCII file for **ypservers** exists), type the following:

```
ypslave# cd /etc/yp/domain_name ↲
ypslave# makedbm -u ypservers ↲
```

> **IMPORTANT** If a hostname is not in **ypservers** it is not informed of updates to the NIS map files.

Then complete the steps described above in the section "How to set up a slave NIS server."

How to change the master server

To change a map's master server, first build the map at the new master. Because the old NIS master's name occurs in a key-value pair in the existing map, it is not sufficient to use an existing copy at the new master or to send a copy there with **ypxfr**. The key must be reassociated with the new master's name. If the map has an ASCII source file, it should be present in its current version at the new master. Remake the NIS map (called "**tech.writers**" below) locally with the following sequence:

```
newmaster# cd /etc/yp ↲
newmaster# make tech.writers ↲
```

The **/etc/yp/Makefile** must be set up correctly for this command sequence to work. If it isn't, do it now. Also, this is a good time to return to the old master (if it will remain an NIS server) and edit **/etc/yp/Makefile** so that "**tech.writers**" is no longer created there—that is, comment out the section of the oldmaster's **/etc/yp/Makefile** that created "**tech.writers**."

If the map exists only as a **dbm** database, you can recreate it on the new master by disassembling an existing copy (one from any NIS server will do) and running the disassembled version through **makedbm**. For example, you can type the following:

```
newmaster# cd /etc/yp ↲
newmaster# ypcat -k tech.writers |makedbm - mydomain/tech.writers ↲
```

After making the map on the new master, you must send a new copy of the map to the other (slave) NIS servers. Do not use **yppush**: the other slaves would request new copies from the old master rather than from the new one.

A typical method (you may discover others) is to become superuser on the old master server and type the following:

```
oldmaster# ypxfr -h newmaster tech.writers ↵
```

Now you have a new copy on the old master and may run **yppush**. The remaining slave servers continue to recognize the old master as the current master, and they attempt to retrieve the current version of the map from the old master. When they do so, they get the new map, which names the new master as the current master.

If this method fails, a cumbersome but effective alternative exists. As superuser, execute the preceding command on each NIS server machine. While this procedures works, it should be considered last. If you are changing the master for all the maps in a domain, you may want to use the command **ypinit -s newmaster** on each server instead.

Troubleshooting an NIS client

This troubleshooting section has two parts—problems seen on an NIS client, and those seen on an NIS server. Because all hosts running NIS are NIS clients, the troubleshooting section for clients applies to hosts that are NIS servers and the NIS master as well.

Before trying to solve a problem with an NIS client, you should be familiar with the information presented in the section "How the NIS network service works" in Chapter 3.

NIS client problems: commands that hang

The most common problem at an NIS client node is for a command to hang and generate console messages similar to the following:

```
yp: server not responding for domain <wigwam>. Still trying
```

Some commands hang even though the system as a whole seems fine and you can run new commands without difficulty.

The preceding message indicates that **ypbind** on the local machine is unable to communicate with **ypserv** in the domain **wigwam**. This often happens when machines that run **ypserv** have crashed. It may also occur if the network or the NIS server machine is so overloaded that **ypserv** cannot respond to your **ypbind** within the time-out period. Under these circumstances, all other NIS client nodes on your network exhibit the same or similar problems. The condition is usually temporary, and the messages usually disappear when the NIS server machine reboots and **ypserv** is able to respond or when the load on the NIS server nodes and/or the network decreases.

In the following circumstances, however, the problem is likely to persist.

- The NIS client has not set, or has incorrectly set, **domainname**. Clients must use a domain name that the NIS servers recognize. First, issue the command **domainname** to determine the current domain name setting, and compare it with the domain name in **/etc/nfs.params** and that used on the NIS servers. If the domain name is set incorrectly, issue the command:

  ```
  # domainname good_domain_name ↵
  ```

  to correct it. If it also is incorrect in the **/etc/nfs.params** file, become superuser on the client machine in question and supply the **domainname_ARG** line with the correct domain name, thus ensuring the domain name will be correctly set each time the machine boots.

- If your domain name is correct, make sure your local network has at least one NIS server machine. You can automatically bind only to a **ypserv** process on your local network, not on another accessible network. There must be at least one NIS server for your machine's domain running on your local network. Two or more NIS servers will improve availability and response characteristics for NIS services.

- If your local network has an NIS server, make sure it is up and running. Check other machines on your local net. If several client machines have problems simultaneously, a server problem is likely. Find a client machine that is behaving normally and run the **ypwhich** command. If **ypwhich** does not return an answer, kill it and type the following on the NIS server machine:

  ```
  # ps -ef | grep yp ↵
  ```

  and look for **ypserv** and **ypbind** processes. If the server's **ypbind** daemon is not running, start it by typing the following:

  ```
  # ypbind ↵
  ```

  If a **ypserv** process is running, type **ypwhich** on the NIS server machine. If **ypwhich** returns no answer, **ypserv** has probably hung and should be restarted. Kill the existing **ypserv** process (you must be logged in as root) and start **ypserv**. Use the **ps** command to find the pid number of the bad **ypserv** process. If, for example, the pid number of the bad **ypserv** process is 10145, start **ypserv** by typing the following:

  ```
  # kill -9 10145 ↵
  # ypserv ↵
  ```

  If **ps** shows no **ypserv** process running, start one.

NIS client problems: NIS service is unavailable

When other machines on the network appear to be functioning properly but the NIS service becomes unavailable on your machine, various aberrations occur. Some commands appear to operate correctly while others terminate, displaying an error message about the unavailability of NIS; some commands limp along in a backup-strategy mode particular to the program involved; and some commands or daemons crash with obscure messages or no message at all. Messages similar to the following may appear:

```
my_machine% ypcat hosts ↵
ypcat: can't bind to NIS server for domain <wigwam>.
    Reason: can't communicate with ypbind.
```

```
my_machine% ypwhich hosts ↵
ypwhich: can't find hosts
```

When such atypical events occur, you might investigate further by typing the command:

```
my_machine% ls -l ↵
```

in a directory containing files owned by many users, including users not in the local machine's **/etc/passwd** file (**/udd/remote_host**, for example). If the **ls -l** commands displays file owners not in the local machine's **/etc/passwd** file as numbers rather than names, this also signifies that the NIS service is not functioning.

These undesirable events usually indicate that your **ypbind** process is not running. Type **ps -ef** to check; if you do not find it, start it by typing the following:

```
my_machine# ypbind ↵
```

Subsequently, NIS problems should disappear.

NIS client problems: ypbind crashes

If **ypbind** crashes almost immediately each time it is started, you should suspect a problem in some other part of the system. First check that the **portmap** daemon is running by typing the following:

```
my_machine% ps -a | grep portmap ↵
```

If it is not running, reboot.

If **portmap** itself will not stay up or if it behaves strangely, look for more fundamental problems: check the network software.

You may be able to access the **portmap** on your machine from a machine that is operating normally. From such a machine, type the following:

```
host_machine# rpcinfo -p my_machine ↵
```

If your **portmap** is fine, the output should look similar to the following:

```
   program vers proto   port
    100000    2   tcp    111    portmapper
    100000    2   udp    111    portmapper
    100004    2   upd    829    ypserv
    100004    2   tcp    830    ypserv
    100004    1   udp    829    ypserv
    100004    1   tcp    830    ypserv
    100007    2   tcp   1024    ypbind
    100007    2   udp   1027    ypbind
    100007    1   tcp   1024    ypbind
    100007    1   upd   1027    ypbind
    100005    1   udp    898    mountd
    100005    1   tcp    900    mountd
    100003    2   udp   2049    nfs
    100026    1   udp   1032    bootparam
    150001    1   udp   1033    pcnfsd
    100017    1   tcp   1030    rexd
    100001    1   udp   1034    rstatd
    100001    2   udp   1034    rstatd
    100001    3   udp   1034    rstatd
    100002    1   udp   1035    rusersd
    100002    2   udp   1035    rusersd
    100012    1   udp   1036    sprayd
    100008    1   udp   1037    walld
    100024    1   udp    785    status
    100024    1   tcp    787    status
    100021    1   udp    790    nlockmgr
    100021    3   udp    793    nlockmgr
    100021    2   tcp    796    nlockmgr
```

On your machine, the port numbers may be different. Four entries represent the **ypbind** process:

```
    100007    2   tcp   1024    ypbind
    100007    2   udp   1027    ypbind
    100007    1   tcp   1024    ypbind
    100007    1   upd   1027    ypbind
```

If these four entries are missing, **ypbind** has been unable to register its services: reboot the machine. If these entries are present but altered whenever you restart **ypbind**, reboot the system, even if the **portmap** is up. If this situation persists after rebooting, call for help.

NIS client problems: ypwhich is inconsistent

The command **ypwhich** tells you which machines are the NIS server and master. When you use **ypwhich** several times at the same client node, it is normal for the answer to vary; the NIS server changes. Often an NIS server machine gets its own NIS services from another NIS server on the net. The binding of NIS client to NIS server changes periodically on a busy net and when the NIS servers themselves are busy. Whenever possible, the system stabilizes at a point where all clients get acceptable response time from the NIS servers.

Troubleshooting an NIS server

This section discusses problems stemming from different versions of an NIS map and those that occur when **ypserv** crashes. Before attempting to solve NIS server problems, you should familiarize yourself with the material presented in the section "How the NIS network service works" in Chapter 3.

Different versions of an NIS map

Because NIS works by propagating maps among servers, you sometimes find different versions of a map on the network servers. This variance is normal if transient, abnormal otherwise.

Most commonly, normal update is prevented when an NIS server or router between NIS server and master is down during a map transfer attempt. (Normal update is described in the section "Propagation of an NIS Map," above). When all NIS servers and the routers between them are up and running, the transfer of maps by **ypxfr** should succeed.

If a particular slave server has problems updating, log in to that server and run **ypxfr** interactively. If **ypxfr** fails it indicates why, and you then can fix the problem. If **ypxfr** succeeds but you suspect it has been failing intermittently, create a log file to enable the logging of messages by typing the following:

```
ypslave# cd /etc/yp ↵
ypslave# touch ypxfr.log ↵
```

This procedures saves all output from **ypxfr**. The output is similar to what **ypxfr** creates when run interactively, with the exception that each line in the log file is time stamped. (Don't worry if the time stamps appear out of order—the time stamp tells you when **ypxfr** completes its work. If copies of **ypxfr** are run simultaneously but their work take differing amounts of time, they may write their summary status line to the log files in an order different from that in which they were invoked.) Any pattern of intermittent failure shows up in the log. When you have fixed the problem, turn off logging by removing the log file. (If you forget to remove it, it grows without limit.)

While still logged in to the problematic NIS slave server, inspect the system **crontab** file, **/usr/spool/cron/crontabs/root** , and the **ypxfr*** shell scripts it invokes. Typos in these files cause propagation problems, as do failures to refer to a shell script within **/var/spool/cron/crontabs/root**, or failures to reference a map within any shell script.

Also, make sure that the NIS slave server is in the map **ypservers** within the domain. If it's not, it continues to function as a server, but **yppush** won't inform it of a new copy of a map.

If the source of the problem is elusive, you can work around it using **rcp**(1) or **ftp**(1) to copy a recent version from any healthy NIS server. You may not be able to do this as root, but you can probably do it as daemon. For instance, transfer map **busted** as follows:

```
ypslave# chmod go+w /etc/yp/mydomain ↲
ypslave# su daemon ↲
$ rcp ypmaster:/etc/yp/mydomain/busted.\* /etc/yp/mydomain\ ↲
$ ^D
ypslave# chown root /etc/yp/mydomain/busted.* ↲
ypslave# chmod go-w /etc/yp/mydomain ↲
```

The * character has been escaped in the command line to expand it on **ypmaster** instead of locally on **ypslave**. Because the map files should be owned by root, you must change ownership of them after the transfer. Obviously, doing the **rcp** as root is preferable.

ypserv crashes

When the **ypserv** process crashes almost immediately and won't stay up despite repeated activations, the troubleshooting process is virtually identical to that described in the section "**ypbind** Crashes," above. Check for the **portmap** daemon by typing the following:

```
ypserver% ps -ef | grep portmap ↲
```

Reboot the server if you do not find the daemon. If you locate it, type the following:

```
ypserver# rpcinfo -p hostname ↲
```

Output similar to the following should appear:

```
   program vers proto   port
    100000    2   tcp    111    portmapper
    100000    2   udp    111    portmapper
    100004    2   upd    829    ypserv
    100004    2   tcp    830    ypserv
    100004    1   udp    829    ypserv
    100004    1   tcp    830    ypserv
    100007    2   tcp   1024    ypbind
    100007    2   udp   1027    ypbind
    100007    1   tcp   1024    ypbind
```

```
100007    1   upd   1027   ypbind
100005    1   udp    898   mountd
100005    1   tcp    900   mountd
100003    2   udp   2049   nfs
100026    1   udp   1032   bootparam
150001    1   udp   1033   pcnfsd
100017    1   tcp   1030   rexd
100001    1   udp   1034   rstatd
100001    2   udp   1034   rstatd
100001    3   udp   1034   rstatd
100002    1   udp   1035   rusersd
100002    2   udp   1035   rusersd
100012    1   udp   1036   sprayd
100008    1   udp   1037   walld
100024    1   udp    785   status
100024    1   tcp    787   status
100021    1   udp    790   nlockmgr
100021    3   udp    793   nlockmgr
100021    2   tcp    796   nlockmgr
```

Naturally, the port numbers will be different on your machine. The four entries representing the **ypserv** process are as follows:

```
100004    2   upd    829   ypserv
100004    2   tcp    830   ypserv
100004    1   udp    829   ypserv
100004    1   tcp    830   ypserv
```

If these entries are absent, **ypserv** has been unable to register its services: reboot the machine. If these entries are present but they are altered whenever you restart **/usr/etc/ypserv**, reboot the machine. If the problem persists after reboot, call for help.

How security is affected by NIS

Security on a system running NIS depends on how NIS consults the administrative files on which its maps are based. Local files on the host (**passwd**, **group**, **aliases**, for instance) are consulted first, followed by the maps in the NIS domain. For example, the system checks the **/etc/aliases** file for mail aliases, then checks the **mail.aliases** NIS map.

The remaining files on which NIS maps are based are global files: **/etc/hosts** , **/etc/networks**, **/etc/ethers**, **/etc/services**, **/etc/netmasks**, **/etc/protocols**, and **/etc/netgroup**. The information in these files is network-wide data and is accessed only from NIS, with the exception of **/etc/hosts** when the **/etc/svcorder** file specifies a non-default name/address resolution order; see **svcorder**(4). When booting, however, each machine requires an entry in **/etc/hosts** for itself. In summary for this set of files, if NIS is running global files are checked only in the NIS maps; a file on your local machine is not consulted.

Special NIS password change

When you change your password with the **passwd** command, you change the explicit entry in your machine's local **/etc/passwd** file. If your password is not provided explicitly, but rather is pulled in from NIS with a + entry, the **passwd** command displays the following error message:

```
Not in passwd file.
```

To change your password in the NIS password file, you must use the **yppasswd** command. To enable this service, you must start up the daemon **yppasswdd** on the machine serving as the master for the NIS password file.

Netgroups: network-wide groups of machines and users

NIS uses the **/etc/netgroup** file on the master NIS server for permission checking during remote mount, login, remote login, and remote shell. It uses **/etc/netgroup** to generate three NIS maps in the **/etc/yp/***domainname* directory: **netgroup, netgroup.byuser**, and **netgroup.byhost** . The NIS map **netgroup** contains the basic information in **/etc/netgroup**. The two other NIS maps contain a more specific form of the information to accelerate the lookup of **netgroups** given the host or user.

The programs that consult these NIS maps are **login, mountd, rlogin,** and **remsh**. **login** consults them for user classifications if it encounters **netgroup** names in **/etc/passwd**. **mountd** consults them for machine classifications if it encounters **netgroup** names in **/etc/exports**. **rlogin** and **remsh** consult the **netgroup** map for both machine and user classifications if they encounter **netgroup** names in the **/etc/hosts.equiv** or **/.rhosts** file.

Adding a new user to the NIS database

In most cases, the easiest way to add a new user to the NIS database is to use the **sysadm usermgmt** command. For more information about **sysadm** and **usermgmt**, see *Managing the DG/UX™ System.*

Following are some points to remember when adding a new user:

- Local and NIS user passwords should be the same.
- The local user and administrator should determine which shell to use, the home directory, full name, and which groups the user belongs to.
- The NIS master administrator should determine the new *userid* to avoid conflicts with another user in the same NIS domain.

- The new user must be added to the local **/etc/group** and to the NIS group map.

You can add a new user manually by including an entry to the password file and by creating a home directory on the new user's machine.

To add a new user to the NIS servers' maps, first update the **passwd** file. As root on the master NIS server machine, edit the master NIS server's **/etc/passwd** file.

On the master NIS server, add a new line to the password file with the **vipw** command; **vipw** brings the password file into the **vi** editor, preventing others from editing it until you are done. Type the following:

```
# /etc/vipw ↵
```

The following example illustrates an entry in the password file for new user **clement** :

```
clement:3u0mRdrJ4tEVs:1947:101:clement:/udd/myserver/clement:/bin
/csh
```

After you have updated the password file and created a password for the new user, be sure to update the NIS maps by changing your directory to **/etc/yp** and running the **make** command:

```
# cd /etc/yp ↵
# make passwd ↵
```

In a network running NIS, the fields in the **yppasswd** map appear the same as in the **passwd**(4) map; refer to the **yppasswd** man page.

If you do not use NIS

If, after using NIS, you want to disable it, change **ypserv_START** to "" (NULL) from CLIENT, MASTER, or SERVER and set **domainname_ARG** to "" (NULL) in the file **/etc/nfs.params**. If **ypserv** on the master is disabled, you can no longer update any of the NIS maps.

End of Chapter

5 An rpcgen programming guide

The details of programming applications to use remote procedure calls (RPC) can be complicated and tedious. Perhaps most complicated is the writing of the External Data Representation (XDR) routines necessary to convert procedure arguments and results into their network format and vice versa.

The **rpcgen**(1) command exists to help programmers write RPC applications simply and directly. **rpcgen** does most of the tedious 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.

What is rpcgen?

rpcgen is a compiler. It accepts a remote program interface definition written in RPC Language, which is similar to C (see "Defining the RPC language" later in this chapter). It produces a C language output that 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. Output files from **rpcgen** can be compiled and linked in the usual way. The developer writes server procedures—in any language that observes the 88open Object Compatibility Standard (OCS) 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**'s 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.

As with 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, you may proceed by using **rpcgen** output as a starting point, and then rewrite it as necessary. (If you need a discussion of RPC programming without **rpcgen**, see Chapter 6).

Converting local procedures to 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 in a simple example 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!\n");
     exit(0);
 }

/*
 * Print a message to the console.
 * Return a Boolean data type 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**() were turned into a remote procedure, then it could be called from anywhere in the network. To make a procedure remote, it's necessary to figure out what the types are for all procedure inputs and outputs. In this case, we have a procedure **printmessage**() that 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 is the protocol:

```
/*
 * 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 that 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 declarations, such as MESSAGEPROG and MESSAGEVERS are in uppercase letters. Though not required, this is a good convention to use.

The argument type is *string* and not *char* * 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 Qstring.

There are 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 version 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 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 lowercase letters, an underscore (_) is appended to it, and the version number (1) is appended).

The final task is to declare the main client program that will call the remote procedure. The program is as follows:

```
/*
 * 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 that call the remote procedure.
2. The remote procedure **printmessage_1**() is called approximately the same way as it is declared in **msg _proc.c**, except for the inserted client handle as the second 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 **rprintmsg** 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 **msg**.x input file:

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 that 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 can try the program. First, copy the server to a remote machine and run it. For this example, the machine is called **mike**. Server processes are run in the background, because they never exit.

```
mike% msg_server & ↵
```

Then on our local machine called **ray** we can print a message on **mike**'s console, as shown below:

```
ray% rprintmsg mike "Hello, mike." ↵
```

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

Generating XDR routines

The previous example 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;
```

IMPORTANT Types (like **readdir_res** in the example above) can be defined using the struct, union and enum keywords, but those keywords should not be used in subsequent declarations of variables of those types. For example, if you define a union foo, you should declare using only foo and not union foo. In fact, **rpcgen** compiles RPC unions into C structures, and it is an error to declare them using the "union" keyword.

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 is 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.
      * Memory allocated here will be freed by xdr_free
      * next time readdir_1 is called
      */

     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 as shown below:

```
/*
 * rls.c: Remote directory listing client
 */

#include <stdio.h>
#include <rpc/rpc.h>     /* always need this */
#include "dir.h"         /* 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("%s\n", nl->name);
        }
        exit(0);
}
```

At this point, we are ready to compile and run the programs. The following example shows how to run **rpcgen** and the C compiler. It also starts the server program.

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

ray%  dir_svc & ↵

mike%  rls ray /usr/pub ↵
.
..
ascii
eqnchar
greek
kbd
marg8
tabclr
tabs
tabs4
mike%
```

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 **sdb**. 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.**

IMPORTANT If you test the client program and the server procedure together as a single program, you may want to comment out calls to RPC library routines, and have client-side routines call server routines directly.

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 file ending in .x. Four symbols may be defined, depending upon which output file is getting generated.

Table 5–1 lists the symbols.

Table 5–1 Symbols for the C preprocessor

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, because there is no guarantee that the compiler will place the output where you intend.

Some rpcgen programming notes

This section provides some helpful hints for programming with **rpcgen**. These hints include information on how to change the default time-out on RPC calls, handle broadcast RPC on the server, and pass information to server procedures.

Changing the default time-out on RPC calls

RPC sets a default time-out of 25 seconds for RPC calls when **clnt**_create() is used. This time-out may be changed using **clnt_control**(). Here is a small code fragment to demonstrate use of **clnt_control**():

```
struct timeval tv;
CLIENT *cl;

cl = clnt_create("somehost", SOMEPROG, SOMEVERS, "tcp");
if (cl == NULL) {
     exit(1);
}
tv.tv_sec = 60; /* change timeout to 1 minute */
tv.tv_usec = 0;
clnt_control(cl, CLSET_TIMEOUT, &tv);
```

Handling broadcasts on the server side

When a procedure is known to be called via broadcast RPC, it is usually wise for the server to not reply unless it can provide some useful information to the client. This prevents the network from getting flooded by useless replies.

To prevent the server from replying, a remote procedure can return NULL as its result, and the server code generated by **rpcgen** will detect this and not send out a reply.

Here is an example of a procedure that replies only if it thinks it is an NFS server:

```
void *
reply_if_nfsserver()
{
     char notnull;   /* just here so we can use its address */

     if (access("/etc/exports", F_OK) < 0) {
             return (NULL);  /* prevent RPC from replying */
     }

     /*
      * return non-null pointer so RPC will send out a reply
      */

     return ((void *)&notnull);
}
```

Note that if the procedure returns type void, it must return a non-NULL pointer if it wants RPC to reply for it.

Passing information to server procedures

Server procedures will often want to know more about an RPC call than just its arguments. For example, getting authentication information is important to procedures that want to implement some level of security. This extra information is actually supplied to the server procedure as a second argument. Here is an example to demonstrate its use. What we've done here is rewrite the previous **printmessage_1()** procedure to allow only root users to print a message to the console.

```
int *
printmessage_1(msg, rq)
     char **msg;
     struct svc_req  *rq;
{
     static in result;       /* Must be static */
     FILE *f;
     struct suthunix_parms *aup;

     aup = (struct authunix_parms *)rq->rq_clntcred;
     if (aup->aup_uid != 0) {
             result = 0;
             return (&result);
     }

     /*
      * Same code as before.
      */
}
```

Defining the RPC language

RPC language is an extension of XDR language. The only extension is the addition of the program type. For a complete description of the XDR language syntax, see Chapter 9 "External Data Representation Standard: Protocol Specification." For a description of the RPC extensions to the XDR language, see Chapter 7 "Remote Procedure Calls: Protocol Specification."

Because XDR language is so close to C, if you know C you know most of XDR. We describe here the syntax of the RPC language, providing 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.

RPC definitions

An RPC language file consists of a series of definitions as shown below:

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

It recognizes the following six types of definitions:

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

XDR structures

An XDR struct is declared similar to 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 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 similar to the input, except that there is an added `typedef` at the end of the output. This allows you to use `coord` instead of `struct coord` when declaring items.

XDR unions

XDR unions are discriminated unions that look quite different from C unions. They are more analogous to Pascal variant records than they are to C unions. They look like the following:

```
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 enumeration and the C enumeration 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;
```

XDR typedef

XDR typedefs have the same syntax as C typedefs as shown below:

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

Here is an example that defines an **fname_type** used for declaring filename strings that have a maximum length of 255 characters.

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

XDR constants

XDR constants are symbolic constants that may be used whenever an 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
```

RPC 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:

```
/*
 * 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
```

XDR declarations

In XDR, there are only four kinds of declarations, shown below:

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

Each type of declaration is discussed below.

1. Simple declarations are just like simple C declarations.

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

   Example:

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

2. 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];
   ```

3. 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 */
   ```

   Because variable-length arrays have no explicit syntax in C, these declarations are actually compiled into `structs`. 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 have a Boolean type called `bool_t` that is either TRUE or FALSE. Variables or expressions declared as type `bool` 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 are compiled into `char *` 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, sequences of arbitrary bytes. It may be declared as either 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 occur in only two places: union definitions and program definitions (as the argument or result of a remote procedure).

End of Chapter

6 Remote Procedure Call protocol programming guide

This chapter assumes that you have a working knowledge of network theory. It is intended for programmers who wish to write network applications using remote procedure calls (RPC) (explained below), and who want to understand the RPC mechanisms usually hidden by the **rpcgen**(1) protocol compiler. **rpcgen** is described in detail in the previous chapter, "An rpcgen Programming Guide."

> **IMPORTANT** Before you attempt to write a network application or convert an existing non-network application to run over the network, you should understand the material in this chapter. However, for most applications, you can circumvent the need to cope with the details presented here by using **rpcgen**. The "Generating XDR routines" section of the previous chapter contains the complete source for a working RPC service—a remote directory listing-service that uses **rpcgen** to generate XDR routines as well as client and server stubs.

What are RPCs? Simply put, they are the high-level communications model 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.

Layers of RPC

The RPC interface, is, in a sense, divided into three layers. (For a complete specification of the routines in the RPC Library, see the **rpc**(3N) manual page.) The layers are described below.

RPC's highest layer

The highest layer is totally transparent to the operating system, machine, and network upon which it is run. Think of this level as a way of using RPC, rather than as a part of RPC proper. Programmers who write RPC routines should make this layer available to others by way of a simple C front-end interface that entirely hides the networking.

To illustrate, at this level a program can simply make a call to **rnusers**(), a C routine that returns the number of users on a remote machine. Users are not explicitly aware of using RPC—they simply call a procedure, just as they would call **malloc**().

RPC's intermediate layer

The intermediate layer is RPC proper. Here, users don't need to consider details about sockets, the DG/UX system, or other low-level implementation mechanisms. They simply make RPC calls to routines on other machines. The selling point here is simplicity. The intermediate-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 an RPC. At the intermediate level, a call to **rnusers**() is implemented by way of these two routines.

Because of its inflexibility (simplicity), the intermediate layer is, unfortunately, rarely used in serious programming. It does not allow time-out specifications or the choice of transport. It allows no DG/UX system 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 are often necessary.

RPC's lowest layer

The lowest layer does allow these details to be controlled by the programmer, so 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 chapter discusses only the interface to C, RPCs can be made from any language. Even though this chapter 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.

The RPC paradigm

This section shows how RPCs are used to communicate across a network. Figure 6–1 shows how messages travel from one machine to another across a network. The arrows indicate the path of the message from the client program on **Machine A** to the server program on **Machine B** and back. The dotted line separates **Machine A** from **Machine B**. The dashed lines indicate part of the respective machines that are not on the path of the message from the client program to the server program.

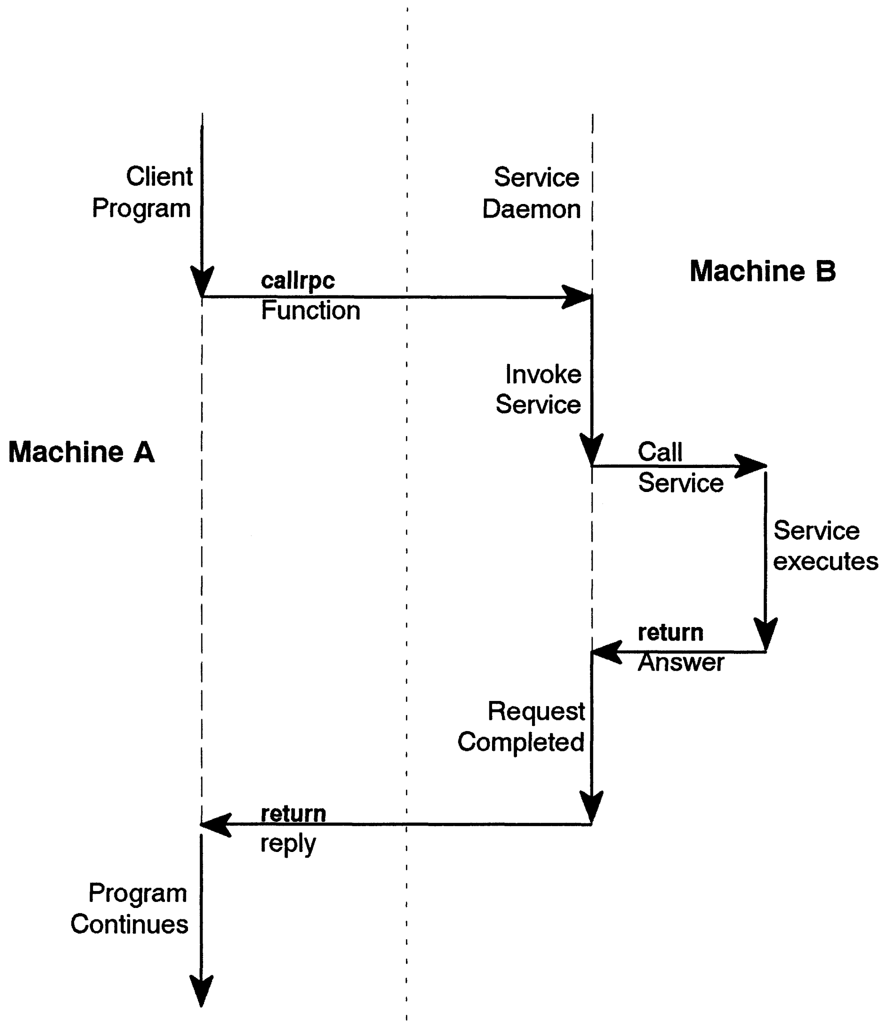

Figure 6–1 Network communication with the remote procedure call

Programming in the high and intermediate layers of RPC

This section discusses how to program in the highest and the intermediate layers of RPC.

Highest layer

Imagine you're writing a program that needs to know how many users are logged in to 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 the following command:

```
% cc program.c -lrpcsvc ↵
```

The **rnusers**() routine,like the other RPC library routines, is documented in the manual pages. See the **intro**(3R) manual page for an explanation of the documentation strategy for these services and their RPC protocols.

Table 6–1 lists some of the RPC service library routines available to the C programmer:

Table 6–1 RPC service library routines

Routine	Description
rusers	Returns information about users on remote machine.
rwall	Writes to specified remote machines.
yppasswd	Updates user password in Network Information Service.

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**(), as discussed in the next section, "Intermediate layer." The RPC services have **rpcgen**(1) protocol description files that can be compiled. (The **rpcgen** protocol compiler radically simplifies the process of developing network applications. See Chapter 5, "An **rpcgen** Programming Guide," for detailed information about **rpcgen** and **rpcgen** protocol description files.)

Intermediate layer

The simplest interface, which explicitly makes RPC calls, uses the functions **callrpc()** and **registerrpc()** With this method, the number of remote users can be gotten 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 RPCs 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 an XDR 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; otherwise, it returns a nonzero value. The return codes (of type cast into an integer) are found in **rpc/clnt.h**.

Because 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. Because RUSERSPROC_NUM takes no argument, the argument parameter of **callrpc**() is **xdr_void**().

If **callrpc**() gets no answer after trying several times to deliver a message, 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 chapter. The remote-server procedure corresponding to the above might look like the following:

```
char *
nuser(indata)
     char *indata;
{
     unsigned long nusers;

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

The procedure takes one argument, which is a pointer to the input of the RPC (ignored in our example), and it returns a pointer to the result. In both versions of C running on AViiON® systems, 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 the following 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**().

CAUTION The UDP transport mechanism can deal only with arguments and results less than 8 Kbytes 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.

Assigning program numbers

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

```
       0x0 through 0x1fffffff   Defined by Sun Microsystems,Inc.
0x20000000 through 0x3fffffff   Defined by customer
0x40000000 through 0x5fffffff   Used temporarily by vendors
0x60000000 through 0x7fffffff   Reserved
0x80000000 through 0x9fffffff   Reserved
0xa0000000 through 0xbfffffff   Reserved
0xc0000000 through 0xdfffffff   Reserved
0xe0000000 through 0xffffffff   Reserved
```

Sun Microsystems, Inc., administers the first group of numbers, which should be identical for all RPC users. The second group of numbers is reserved for specific customer applications. The third group is used temporarily for applications that generate program numbers dynamically. The final groups are reserved by Sun Microsystems, Inc., for future use; they should not be used.

To add an application to the first range and have Sun Microsystems, Inc. define a number for it, send a request by network mail to **rpc@sun** or write to:

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

Please include a **.x** file describing your protocol that can be compiled by **rpcgen**. You will be given a unique program number in return.

The RPC program numbers and protocol specifications of standard RPC services can be found in the include files in **/usr/include/rpcsvc**. These services, however, constitute only a small subset of those that have been registered. The complete list of registered programs, as of the time when this manual was printed, is provided in Table 6–2.

Table 6–2 RPC registered programs

RPC Number	Program	Description
100000	PMAPPROG	portmapper
100001	RSTATPROG	remote stats
100002	RUSERSPROG	remote users
100003	NFSPROG	NFS
100004	YPPROG	Network Information Service
100005	MOUNTPROG	mount daemon
100006	DBXPROG	remote dbx
100007	YPBINDPROG	yp binder
100008	WALLPROG	shutdown msg
100009	YPPASSWDPROG	yppasswd server
100010	ETHERSTATPROG	ether stats
100011	RQUOTAPROG	disk quotas
100012	SPRAYPROG	spray packets
100013	IBM3270PROG	3270 mapper
100014	IBMRJEPROG	RJE mapper

Continued

Table 6–2 RPC registered programs

RPC Number	Program	Description
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_PROG	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
300001	ADT_RFLOCKPROG	ADT file locking

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, such as **xdr_u_long**() in the previous example, or a user-supplied one. XDR has the built-in type routines shown below (see **xdr**(3N)).

```
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 pass only two parameters to their XDR routines. **xdr_wrapstring**() has only two parameters, and is thus okay. It calls **xdr_string**().

An example of a user-defined type routine using **callrpc**(), is as follows. First, you could send the following structure:

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

Second, you would call **callrpc**() as follows:

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

Last, you could write **xdr_simple**(), as follows:

```
#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. We provide a few implementation examples here. For a complete description of XDR, see Chapter 8 "External Data Representation Standard: DG Technical Notes."

In addition to the built-in primitives, there are also the prefabricated building blocks listed below (see **xdr**(3N)):

```
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 the following:

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

Then make an RPC call such as this one:

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

with **xdr_varintarr**() defined like so:

```
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 you know the size of the array in advance, you can use **xdr_vector**(), which serializes fixed-length arrays. This routine can be used like the following:

```
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 serializing. 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 serializing, it gets the string length from **strlen**(), and on de-serializing 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);
 }
```

Note that we could as easily call **xdr_simple**() here instead of **xdr_reference**().

Programming in the lowest layer of RPC

In the examples given so far, RPC has automatically taken care of many details for you. In this section, we'll show you how you can change the defaults by using lower layers of the RPC library. It is assumed that you are familiar with sockets and the system calls for dealing with them. If you are not familiar with the socket interface, see *Programming with TCP/IP on the DG/UX™ System.*

There are several occasions when you may need to use lower layers of RPC. First, you may need to use TCP, because the higher layer uses UDP, which restricts RPC calls to 8 Kbytes of data. Using TCP permits calls to send long streams of data. For an example, see the section called "Using TCP" later in this chapter. Second, you may want to allocate and free memory while serializing or de-serializing with XDR routines. There is no call at the higher level to let you free memory explicitly. For more explanation, see the upcoming section called "Memory allocation with XDR." Third, you may need to perform authentication on either the client or server side, by supplying credentials or verifying them. See the explanation in "Authentication with RPC" later in this chapter.

RPC on the server

The server for the **nusers**() program shown below does the same thing as the one using **registerrpc**() above, but it 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 assign it to the 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 **clnttcp_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 run the **bind** system call on their socket if it is not bound already.

A service may choose to register its port number with the local port mapper service. This is done by specifying a non-zero protocol number in **svc_register()**. Incidentally, clients can discover the server's port number by consulting the port mapper on their server's machine. This can be done automatically by specifying a zero port number in **clntudp_create()** or **clnttcp_create()**.

After you create 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 the procedure NULLPROC (currently zero) returns with no results. This can be used as a simple test for detecting whether a remote program is running. Second, there is a check for invalid procedure numbers. If one is detected, **svcerr_noproc()** is called to handle the error.

The user service routine serializes the results and returns them to the RPC caller via **svc_sendreply()**. Its first parameter is the SVCXPRT handle, the second is the XDR routine, and the third is a pointer to the data to be returned. Not illustrated above is the way 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 in. 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");
              return (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.

Memory allocation with XDR

XDR routines not only handle input and output, they also handle 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));
}
```

If space has already been allocated in **chararr**(), it can be called from a server like this:

```
char chararr[SIZE];

svc_getargs(transp, xdr_chararr1, 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 routine **xdr_finalexample**(), given in the section "Passing arbitrary data types" earlier in this chapter, if **finalp->string** were NULL, then it would not be freed. The same is true for **finalp->simplep**.

To summarize, each XDR routine is responsible for serializing, deserializing, and freeing memory. When an XDR routine is called from **callrpc**(), the serializing part is used. When called from **svc_getargs**(), the deserializer is used. And when called from **svc_freeargs**(), the memory deallocator is used. When building simple examples like those in this section, a user doesn't have to worry about the three modes. See Chapter 8, "External Data Representation: DG Technical Notes," for examples of more sophisticated XDR routines that determine which of the three modes they are in and adjust their behavior accordingly.

RPC on the client

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);
     exit(0);
}
```

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

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

The parameters to **clntudp_create()** are the server address, the program number, the version number, a time-out 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()** time-out divided by the **clntudp_create()** time-out.

Note that the **clnt_destroy()** call always deallocates the space associated with the CLIENT handle. If the socket associated with the CLIENT handle was opened by the RPC library, **clnt_destroy()** closes the socket. However, if the socket was opened by the user, it stays open. 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()**, as shown below.

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

There is no time-out 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 zero is specified for either of these, the system chooses a reasonable default.

Other RPC features

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

Using select on the server

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 for 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 for. 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 every time any RPC library routine is called, because descriptors are constantly being opened and closed, for example for TCP connections.

Broadcast RPC

The port mapper is a daemon that converts RPC program numbers into DARPA protocol port numbers; see the **portmap**(1M) man page. You can't do broadcast RPC without the port mapper. Here are the main differences between broadcast RPC and regular RPC calls:

1. Regular RPC expects one answer, whereas broadcast RPC expects many answers (one or more answers from each responding machine).
2. Broadcast RPC can be supported only by packet-oriented (connectionless) transport protocols like UDP/IP.
3. The implementation of broadcast RPC treats all unsuccessful responses as garbage by filtering them out. Thus, if there is a version mismatch between the broadcaster and a remote service, the user of broadcast RPC never knows.
4. All broadcast messages are sent to the portmap port. Thus, only services that register themselves with their port mapper are accessible via the broadcast RPC mechanism.
5. Broadcast requests are limited in size to the MTU (Maximum Transfer Unit) of the local network. For Ethernet, the MTU is 1500 bytes.

Broadcast RPC interface

Broadcast RPC can be used as follows:

```
#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 response 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 .

Batching in RPC

The RPC architecture is designed so that clients send a call message and wait for servers to reply that the call has 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. RPC batch facilities allow you to continue computing while waiting for a response.

RPC messages can be placed in a pipeline of calls to a desired server; this is called batching. Batching assumes that: 1) each RPC call in the pipeline requires no response from the server, and the server does not send a response message; and 2) the pipeline of calls is transported on a reliable byte stream transport such as TCP/IP. Because 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.

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

A contrived example of batching follows. Assume a string-rendering service (such as 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 the following.

```
#include <stdio.h>
#include <rpc/rpc.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))
                     fprintf(stderr, "can't reply to RPC
call\n");
             return;
     case RENDERSTRING:
             if (!svc_getargs(transp, xdr_wrapstring, &s)) {
                     fprintf(stderr, "can't decode arguments\n");
                     /*
                      * Tell caller he screwed up
                      */
                     svcerr_decode(transp);
                     break;
             }
             /*
              * Code 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;
             }
             /*
              * Code 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 test to indicate whether or not the procedure should respond.

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

The following is an example of a client that uses batching to render a number of strings; the batching is flushed when the client gets a null string (EOF).

```
#include <stdio.h>
#include <rpc/rpc.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;
     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);
     exit(0);
}
```

Because 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.

Authentication with RPC

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 file system, 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.

UNIX authentication

This section discusses how UNIX authentication is handled from both client and server sides.

UNIX Authentication on the Client

The caller can create a new RPC client handle like the following.

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

After the handle is created, the appropriate transport instance defaults the associated authentication handle to be the following.

```
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 like this:

```
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. Because the RPC user created this new style of authentication, the user is responsible for destroying it with the following:

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

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

UNIX Authentication on the Server

Service implementors have a more difficult time dealing with authentication issues, because the RPC package passes to the service dispatch routine a request that has an arbitrary authentication style associated with it. Consider the following 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:

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

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

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

2. 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 the UNIX system 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 user's 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");
                     return (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 assign it to the variable nusers
                 */
                if (!svc_sendreply(transp, xdr_u_long, &nusers)) {
                        fprintf(stderr, "can't reply to RPC
call\n");
                        return (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 relationship 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.

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 because **inet** passes a socket as file descriptor 0.

```
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 */
```

Also, because the program will already be registered by **inetd**, **svc_register**() should be called with the final flag as zero.

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

Remember that if you want to exit from the server process and return control to **inet**, you need to explicitly exit, because **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**(5). **server** is the program implementing the server,and program and version are the program and version numbers of the service. For more information, see **inetd.conf**(4).

If the same program handles multiple versions, the version number can be a range, as in the following.

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

More examples that use RPC features

This section presents three examples that use RPC features. These examples show how to use versions, TCP, and callback procedures.

Using 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 short rather than along. If we name this version RUSERSVERS_SHORT, then a server that wants to support both versions would do a double register. Double registers can be done as in the following.

```
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, as shown below:

```
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");
        return (1);
            }
            return;
     case RUSERSPROC_NUM:
             /*
      * Code here to compute the number of users
      * and assign it to the 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;
     }
}
```

Using TCP

This section contains an example of RPC that use TCP. The example is divided into three parts. The first part shows the XDR routine, the second part shows the sender routines, and the third part shows the receiving routines.

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

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

xdr_rcp(xdrs, fp)
     XDR *xdrs;
     FILE *fp;
{
     unsigned long size;
     char buf[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");
                           return (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");
                           return (1);
                     }
             }
    }
}
```

The sender routines are as follows:

```
/*
 * 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);
        }
        exit(0);
}

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);
                return (-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");
                return (-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 are follows:

```
/*
 * 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");
                     return (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;
             }
             return (0);
     default:
             svcerr_noproc(transp);
             return;
     }
}
```

Using callback procedures

Occasionally, it is useful to have a server become a client and then make the RPC call back to its client process. An example is remote debugging, in which 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. Because this will be a dynamically generated program number, it should be in the transient range 0x40000000 through 0x5fffffff. The routine **gettransient**() returns a valid program number in the transient range and registers it with the port mapper. It talks to only the port mapper 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, 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. The following example shows how to do an RPC callback.

```
#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);
}
```

IMPORTANT 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**(3N) manual page for more details on the conversion of network addresses from network-to-host byte order.

The following pair of programs illustrates 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 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 an RPC callback, 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:
exampleprog\n");
                             return (1);
                     }
                     return (0);
             case 1:
```

```
                        if (!svc_getargs(transp, xdr_void, 0)) {
                                svcerr_decode(transp);
                                return (1);
                        }
                        fprintf(stderr, "client got callback\n");
                        if (!svc_sendreply(transp, xdr_void, 0)) {
                                fprintf(stderr, "err: exampleprog");
                                return (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");
        }
}
```

End of Chapter

7 Remote Procedure Calls: protocol specification

IMPORTANT This chapter specifies a protocol that Data General and other companies are using. It has been designated RFC1050 by the ARPA Network Information Center.

Introduction

This chapter specifies a message protocol used in implementing the Remote Procedure Call (RPC) package. The message protocol is specified with the External Data Representation (XDR) language. See Chapter 9, "External Data Representation Standard: Protocol Specification" for the details. Here, we assume that the reader is familiar with XDR and do not attempt to justify it or its uses. For further information on RPCs, refer to the paper by Birrell and Nelson cited at the end of this chapter.

Defining terms

This chapter discusses servers, services, programs, procedures, clients, and versions. A server is a piece of software in which 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 section "Portmapper program protocol," later in this chapter, for an example). Network clients are pieces of software that initiate RPCs to services. A server may support more than one version of a remote program in order to be forward compatible with changing protocols.

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

The RPC model

The RPC model is similar to the local procedure call (LPC) 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 regains control. At that point, the results of the procedure are extracted from the well-specified location, and the caller continues execution.

The RPC 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. After the reply message has been received, the results of the procedure are extracted, and the caller's execution is resumed.

On the server side, a process is dormant while 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 waits for 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 given only as an example. The RPC protocol makes no restrictions on the concurrency model implemented, and others are possible. For example, RPC calls might 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.

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 with only 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 the application knows it is running on top of a reliable transport such as TCP/IP, then most of the work is already done. On the other hand, if the application is running on top of an unreliable transport such as UDP/IP, the application must implement its own retransmission and time-out policy because 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 re-transmits 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 insure some degree of idempotency 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 ID is by the client RPC layer in matching replies to requests. However, a client application may choose to re-use its previous transaction ID when re-transmitting 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 idempotency. 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 the Versatile Message Transaction Protocol (VMTP) is perhaps the most natural transport for RPC. (The VMTP is discussed in a paper by Cheriton, cited at the end of this chapter.)

IMPORTANT RPC is currently implemented on top of both TCP/IP and UDP/IP transports.

Binding and rendezvous independence

The act of binding a client to a service is not part of the specification. This important and necessary function is left up to some higher-level software. (The software may use RPC itself—see the upcoming section called "Portmapper program protocol," later in this chapter).

Implementors should think of the RPC protocol as the jump-subroutine (JSR) instruction 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.

Authentication in RPC

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 upcoming section called "Authentication protocols."

RPC protocol requirements

The RPC protocol must provide for the following:

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

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

1. RPC protocol mismatches.
2. Remote program protocol version mismatches.
3. Protocol errors (such as misspecification of a procedure's parameters).
4. Reasons why remote authentication fail.
5. Any other reasons why the desired procedure does not get called.

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 (such as Sun Microsystems, Inc). Once an implementor has a program number, he/she can implement his/her remote program; the first implementation would most likely have the version number 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 it possible for old and new protocols to communicate through the same server process.

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 can also change. Therefore, the call message also contains the RPC version number (which is always 2, for the version of RPC described here).

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

1. The remote implementation of RPC does speak Protocol Version 2. The lowest and highest supported RPC version numbers are returned.
2. The remote program is not available on the remote system.

3. The remote program does not support the requested version number. The lowest and highest supported remote program version numbers are returned.
4. The requested procedure number does not exist. (This is usually a caller-side protocol or programming error.)
5. The parameters to the remote procedure appear to be garbage from the server's point of view. (Again, this is usually caused by a disagreement about the protocol between client and service.)

Authentication fields

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,
     AUTH_DES     = 3,
     /* and more to be defined */
};

struct opaque_auth {
     auth_flavor flavor;
     opaque body<400>; };
```

An **opaque_auth** structure is an **auth_flavor** enumeration followed by bytes that 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," later in this chapter, for definitions of the various authentication protocols.)

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

Program number assignment

Program numbers are given out in groups of *0x20000000* hexadecimal (decimal 536870912) as shown in the following chart:

Table 7-1 Program number assignments

Program numbers	Description
`0 – 1fffffff`	Defined by Sun Microsystems, Inc.
`20000000 – 3fffffff`	Defined by customer
`40000000 – 5fffffff`	Temporary
`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, Inc., and should be identical for all sites. The second range is reserved for specific customer applications. The third group is used temporarily for applications that generate program numbers dynamically. The final groups are reserved by Sun Microsystems, Inc., for future use, and should not be used.

Other uses of the RPC protocol

The intended use of this protocol is for calling remote procedures. That is, each call message is matched with a response message. However, the protocol itself is a message-passing protocol with which other (non-RPC) protocols can be implemented. Data General currently uses 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 respond only when the request is successfully processed, and they are silent in the face of errors. Broadcast RPC uses the Port Mapper RPC service to achieve its semantics. See the upcoming section called "Portmapper program protocol," for more information.

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 procedure#   */
     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 */
     AUTH_REJECTEDCRED  = 2,  /* client must begin new session */
     AUTH_BADVERF       = 3,  /* bad verifier */
     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 used only 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;
             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.  Following 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.
 */
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 of which the 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; };
```

Authentication protocols

As previously stated, authentication parameters are opaque, but open-ended to the rest of the RPC protocol. This section defines some types (or flavors) of authentication supported by Data General. Customers can invent new authentication types, using the same rules for number assignment of types as there are for program number assignment.

Null authentication

Often, calls are made in which the caller is unknown or the server doesn't need to know who the caller is. In such cases, the value of the type (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. We recommend 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 bytes of the credential's opaque body encode 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 that the caller machine may generate. The *machinename* is the name of the caller's machine. 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 that contain the caller as a member. The verifier accompanying the credentials should be of AUTH_NULL (see "Null authentication" 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 that 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 RPC 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.

Record-marking standard

When RPC messages are passed on top of a byte stream protocol (such as 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). Data General uses this RM TCP/IP transport for passing RPC messages on TCP streams. One RPC message fits into one RM record.

A record is composed of one or more record fragments. A record fragment is a 4-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 value that indicates whether the fragment is the last fragment of the record (bit value 1 implies the fragment is the last fragment), and a 31-bit unsigned binary value, which is the length, in bytes, of the fragment's data. The Boolean value is the 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.)

RPC language

RPC language is an extension of XDR language. It is used to describe the procedures that operate on XDR data-types. This section provides a sample program described in the RPC language, a specification for the RPC language, syntax notes.

An example service described in the RPC 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 requires 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 does not contain the PINGPROC_PINGBACK procedure. It is useful for compatibility with old client programs; as this program matures, it may be dropped from the protocol entirely.

RPC language specification

RPC language is identical to 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

Some additional notes on RPC syntax are as follows:

1. The following key words are added and cannot be used as identifiers: program and version.
2. A version name and version number cannot occur more than once within the scope of a program definition.
3. A procedure name and procedure number cannot occur more than once within the scope of a 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.

Portmapper program protocol

The portmapper program maps RPC program and version numbers to transport-specific port numbers. This program makes dynamic binding of remote programs possible.

Dynamic binding 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 portmapper on a reserved port, the port numbers of other remote programs can be ascertained by querying the portmapper.

The portmapper 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 portmapper, however, does have a fixed port number. So, to broadcast to a given program, the client actually sends its message to the portmapper located at the broadcast address. Each portmapper that picks up the broadcast then calls the local service specified by the client. When the portmapper gets the reply from the local service, it sends the reply back to the client.

Portmapper protocol specification (in RPC language)

This section shows how to use the portmapper program 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;
     opaque res<>;
};

/*
 * Portmapper 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;
```

Portmapper operation

The portmapper program currently supports two protocols: UDP/IP and TCP/IP. The portmapper is contacted by sending messages to it on assigned port number 111 on either of these protocols. (This subject is discussed in "Assigned Numbers," RFC 923, by Reynolds and Postel, cited at the end of this chapter.) 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 portmapper 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 portmapper 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 portmapper'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 for supporting broadcasts to arbitrary remote programs via the well-known portmapper's port. The parameters **prog**,**vers**, and **proc**, and the bytes of args are the program number,version number, procedure number, and parameters of the remote procedure. The procedure returns the remote program's port number, and the bytes of results are the results of the remote procedure.

Note the following:

1. This procedure sends a response only if the procedure was successfully executed and is silent (no response) otherwise..

2. The portmapper communicates with the remote program using UDP/IP only.

Papers cited in chapter text

Birrell, Andrew D., and Nelson, Bruce Jay. Implementing Remote Procedure Calls. XEROX CSL-83-7, October 1983.

Cheriton, D. VMTP: Versatile Message Transaction Protocol, Preliminary Version 0.3. Stanford University, January 1987.

Diffie and Hellman. Net Directions in Cryptography. The Institute of Electrical and Electronics Engineers, Inc (IEEE) Transactions on Information Theory IT-22, November 1976.

Harrenstien, K. Time Server, RFC 738. Information Sciences Institute, October 1977.

National Bureau of Standards. Data Encryption Standard. Federal Information Processing Standards Publication 46, January 1977.

Reynolds, J., and Postel, J. Assigned Numbers, RFC 923. Information Sciences Institute, October 1984.

End of Chapter

8 External Data Representation: DG technical notes

This chapter contains technical notes on Data General'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 Chapter 9 "External Data Representation Standard: Protocol Specification." XDR is the backbone of DG's Remote Procedure Call (RPC) package, in the sense that data for RPCs 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. (For complete specification of the system XDR routines, see the **xdr**(3N) manual page.)

This chapter 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 the "Number filters," "Floating-point filters," and "Enumeration filters" sections. Programmers wishing to implement RPC and XDR on new machines will be interested in the rest of the chapter, as well as Chapter 9, "External Data Representation Standard: Protocol Specification," which will be their primary reference.

> **IMPORTANT** **rpcgen** can be used to write XDR routines even in cases when no RPC calls are being made.

On Data General systems, C programmers that want to use XDR routines must include the file **<rpc/rpc.h>**, which contains all the necessary interfaces to the XDR system. Because the C library *libc.a* contains all the XDR routines, compile as normal. For example, you compile **program.c** as follows.

```
example% cc program.c ↵
```

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);
         }
     }
     exit(0);
}
```

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");
     exit(0);
}
```

The two programs appear to be portable, because they pass *lint* checking, and because they exhibit the same behavior when executed on two different hardware architectures, an AViiON® station and a VAX.

Piping the output of the *writer* program to the *reader* program gives identical results on an AViiON station and a VAX.

```
aviion% writer | reader ↵
0 1 2 3 4 5 6 7
aviion%

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

With the advent of local area networks and Berkeley's 4.2 BSD release came the concept of network pipes—a process that produces data on one machine, and a second process that 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 an AViiON station, and the second consumes data on a VAX.

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

Identical results can be obtained by executing **writer** on the VAX and **reader** on the AViiON station. These results occur because the byte ordering of long integers differs between the VAX and the AViiON station, even though word size is the same. Note that 16777216 is 2^{24}—when 4 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 recognizes the standard representation of a long integer in its external form. The revised versions of **writer** are shown below.

```
#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);
         }
     }
     exit(0);
}
```

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");
     exit(0);
}
```

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

```
aviion% writer | reader ↲
 0 1 2 3 4 5 6 7
 aviion%
 vax% writer | reader ↲
 0 1 2 3 4 5 6 7
 vax%

 aviion% writer | rsh vax reader ↲
 0 1 2 3 4 5 6 7
 aviion%
```

IMPORTANT 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 in which they are defined.

A canonical standard

XDR's approach to standardizing data representations is canonical. That is, XDR defines a single byte order (in big-endian style, which orders the most significant bit first), 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 representations 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 upon the community of existing portable data creators and users. A new machine joins this community by being made able 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. If, for example, two little-endian machines (which order the least significant bit first) are transferring integers according to the XDR standard, then the sending machine converts the integers from little-endian byte order to big-endian byte order (most significant bit first). 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 the difference in cost.

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.

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 makes sense to use the library even when the data is not shared among machines on a network.

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

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

> **IMPORTANT** 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 following 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 serialize or deserialize data. This feature is critical to software engineering of portable data. The idea is to call the same routine for either operation—this almost guarantees that serialized data can also be deserialized. One routine is used by both producer and consumer of networked data. This is implemented by always passing the address of an object rather than the object itself—only in the case of deserialization is the object modified. This feature is not shown in our trivial example, but its value becomes obvious when nontrivial data structures are passed among machines. If needed, the user can obtain the direction of the XDR operation. See the upcoming section called "XDR operation directions" 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. You can write your data structure as follows:

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

The corresponding XDR routine describing this structure would be like this:

```
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 for which the only values are `TRUE` (1) and `FALSE` (0). This document uses the following definitions:

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

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.

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>**.

Number filters

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

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

Specifically, the eight primitives are as follows:

```
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 shown above, `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.

Floating-point filters

The XDR library also provides primitive routines for C's floating-point types, as shown below:

```
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, `fp` or `dp`, is the address of the floating-point number that provides data to the stream or receives data from it. Both routines return TRUE if they complete successfully, and FALSE otherwise.

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

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 type is always `TRUE` (1) or `FALSE` (0).The primitive for the enum is as follows:

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

#define enum_t int

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

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

The second parameters `ep` and `bp` are addresses of the associated type that provides data to, or receives data from, the stream `xdrs`.

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, as shown below:

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

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, opaque data, unions, and pointers to structures.

Constructed data type primitives may use memory management. In many cases, memory is allocated when deserializing data with `XDR_DECODE`. Therefore, the XDR package must provide means to deallocate memory. This is done by an XDR operation, XDR_FREE. To review, the three XDR directional operations are XDR_ENCODE, XDR_DECODE, and XDR_FREE.

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**(), as shown below:

```
  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 filename may be no longer than 255 characters.

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

> **IMPORTANT** Keep `maxlength` small. If it is too big you can run out of memory, because **xdr_string**() will call **malloc**() for space.

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

The effect of deserializing a string is subtle. First, the length of the incoming string is determined; it must not exceed `maxlength`. Next, `sp` is dereferenced; if the 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, and 3) the external representation of the bytes is the same as their internal representation. The primitive *xdr_bytes()* converts between the internal and external representations of byte arrays, as shown below:

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

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

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. The **xdr_array**() routine is called to encode or decode each element of the array, as shown below:

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

Before we define more constructed data types, we will 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** manual page), (b) the user's UID (see the **geteuid** manual page), and (c) the group numbers to which the user belongs (see the **getgroups** manual page.) A structure with this information and its associated XDR routine could be coded as shown below:

```
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 a netuser structure. The declaration and its associated XDR routines are shown below:

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

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

Example C

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

```
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() passes only 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 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 **xdr_opaque()** primitive 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 is not machine-portable.

Fixed-sized arrays

The XDR library provides a primitive, **xdr_vector()**, for fixed-length arrays. This primitive can be used as shown below:

```
#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. Discriminated unions can be used as shown below:

```
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 ordered 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 default arm procedure is called if it is non-null; otherwise the routine returns FALSE .

Example A

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

```
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)serialize the discriminated union, as shown below:

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

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

The routine xdr_gnumbers() was presented in a previous section called "The XDR library." The routine xdr_wrap_string() was presented in example C. The default arm parameter to xdr_union() (the previous 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 explicit 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 xdr_reference() primitive makes it easy to *serialize*, *deserialize*, and free these referenced structures.

```
bool_t xdr_reference(xdrs, pp, ssize, 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 B

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 as follows:

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

The corresponding XDR routine for this structure is as follows:

```
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);
}
```

Pointer semantics and XDR

In many applications, C programmers attach double meaning to the values of a pointer. Typically, the value *NULL* (or zero) means data is not needed, yet some application-specific interpretation applies. In essence, the C programmer is encoding a discriminated union efficiently by overloading the interpretation of the value of a pointer. For instance, in Example B, 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 for which the value is *NULL* to `xdr_reference()` when serializing data will most likely cause a memory fault and, on the DG/UX system, a core dump.

The routine `xdr_pointer()` correctly handles NULL pointers. For more information about its use, see the upcoming section called "Linked lists."

Non-filter primitives

XDR streams can be manipulated with the primitives discussed in this section. The primitives are as follows:

```
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.

CAUTION 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`.

CAUTION 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.

XDR operation directions

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

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 DG/UX files, and memory.

Standard i/o streams

XDR streams can be interfaced to standard I/O using the xdrstdio_create() routine, as shown below:

```
#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.

Memory streams

Memory streams allow the streaming of data into or out of a specified area of memory. The routine xdrmem_create() initializes an XDR stream in local memory, as shown below:

```
#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 memory is pointed to by parameter addr; parameter len is the length in bytes of the memory. The parameter sxdrs 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 into memory before calling the **sendto()** system routine.

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 DG/UX file or 4.2 BSD connection interface. A record stream might look like the following:

```
#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 bi-directional, 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 DG/UX files.

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

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

The XDR stream provides means for delimiting records in the byte stream. The implementation details of delimiting records in a stream are discussed in the upcoming section called "Advanced topics." 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.

XDR stream implementation

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

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. x_getpostn(), x_setpostn(), and x_destroy() are macros for accessing operations. 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. (Use of the resulting buffer is not data-portable. Users are discouraged from using this feature.)

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

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

The operations x_getlong() and x_putlong() receive and put long numbers from and to the data stream. It is the responsibility of these routines to translate the numbers between the machine representation and the (standard) external representation. The 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, as shown below:

```
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.

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. See Chapter 9, "External Data Representation Standard: Protocol Specification," for a detailed description of this language.

Linked lists

The last example in the earlier section called "Pointers" 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_nex*t* field is used to indicate whether or not the object has terminated. Unfortunately, if the object continues, the gn_nex*t* field is also the address of where it continues. The link addresses carry no useful information when the object is serialized.

The XDR data description of this linked list is described by the recursive declaration of `gnumbers_list`, as shown below:

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

struct gnumbers_node {
     gnumbers gn_numbers;
     gnumbers_node *gn_next;
};
```

In this description, the Boolean response indicates whether there is more data following it. If the Boolean response 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 data type 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 using XDR on a list with these routines is that the C stack grows linearly with respect to the number of nodes in the list. This is due to the recursion. The following routine collapses the above two mutually recursive routine 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 or not there is more data, so that this Boolean information can be serialized. Notice that this statement is unnecessary in the XDR_DECODE case, since the value of more_data is not known until we deserialize it in the next statement.

The next statement is the more_data field of the XDR union. Then if there is 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 storage pointed to by gnp will be freed and will no longer be 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 use XDR on the data in the node using the primitive xdr_reference().The xdr_reference() routine is like xdr_pointer(), which we used before, but it does not send over the Boolean response indicating whether there is more data. We use it instead of xdr_pointer() because we have already used XDR on 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 using XDR on 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 uses XDR on 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 there cursive version, this non-recursive routine is far less likely to use all of the available memory. It will also run more efficiently, because 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.

End of Chapter

9 External Data Representation Standard: protocol specification

IMPORTANT This chapter specifies a protocol that Data General and other companies are using. It has been designated RFC1014 by the ARPA Network Information Center.

Introduction

External Data Representation (XDR) is a standard for describing and encoding data. It is useful for transferring data between different computer architectures, and has been used to communicate data between such diverse machines as the AViiON® station, VAX, IBM-PC, and Cray. XDR fits into the International Organization for Standardization (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 be used only to describe data; it is not a programming language. This language allows you 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 (see *The C Programming Language* by Kernighan and Ritchie, cited at the end of this chapter). Protocols such as RPC (remote procedure call) and 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 8 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 (see *On Holy Wars and a Plea for Peace* by Danny Cohen, cited at the end of this chapter), 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 4, then the n bytes are followed by enough (0 to 3) residual 0 bytes, r, to make the total byte count a multiple of 4.

Figure 9–1 illustrates the basic block size. It is in the familiar graphic box notation for illustration and comparison. Each box (delimited by a plus sign at each of the four corners and bars and dashes) depicts a byte. Ellipses (...) between boxes show zero or more additional bytes where required.

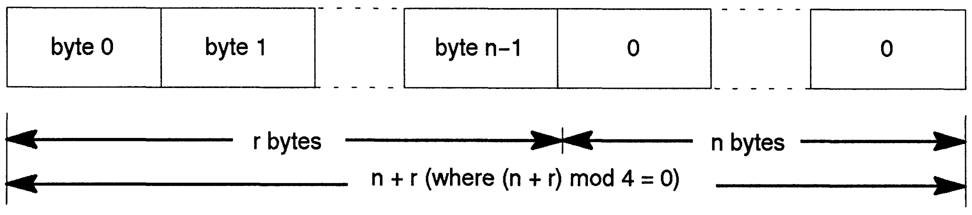

Figure 9–1 Basic block layout

XDR data types

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.

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 upcoming section called "XDR language specification."

For some data types, more specific examples are included. A more extensive example of a data description is in the upcoming section called "A sample XDR data description."

XDR signed integers

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. Figure 9–2 illustrates how signed integers are declared.

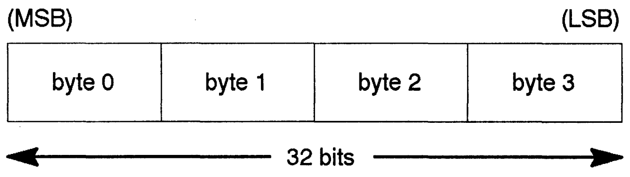

Figure 9–2 Byte layout for signed integer declaration

XDR unsigned integers

An XDR unsigned integer is a 32-bit datum that encodes a nonnegative integer in the range [0,4294967295]. It is represented by an unsigned binary number whose most and least significant bytes are 0 and 3, respectively. Figure 9–3 illustrates how unsigned integers are declared.

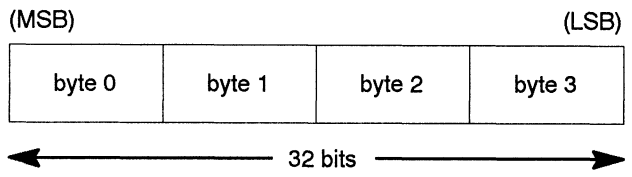

Figure 9–3 Byte layout for unsigned integer declaration

Enumerations

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 colors red, yellow, and blue could be described by the following 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 data types

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

```
bool identifier;
```

This is equivalent to the following:

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

Hyper integers and unsigned hyper integers

The standard also defines 64-bit (8-byte) numbers called hyper integers and unsigned hyper integers. Their representations are the obvious extensions of integers and unsigned integers defined above. They are represented in two's complement notation. The most and least significant bytes are 0 and 7, respectively. Figure 9–4 illustrates how hyper integers and unsigned hyper integers are declared.

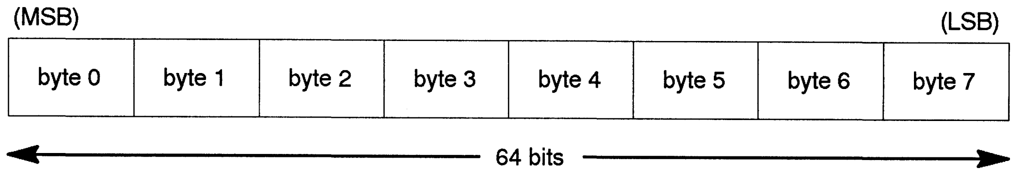

Figure 9–4 Byte layout for signed and unsigned hyper integers

Floating-point data types

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 (see the *IEEE Standard for Binary Floating-Point Arithmetic*, cited at the end of this chapter.) 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 numbers 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. Twenty-three bits are devoted to this field.

Therefore, the floating-point number is described as follows:

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

Figure 9–5 illustrates how single-precision floating-point numbers are declared.

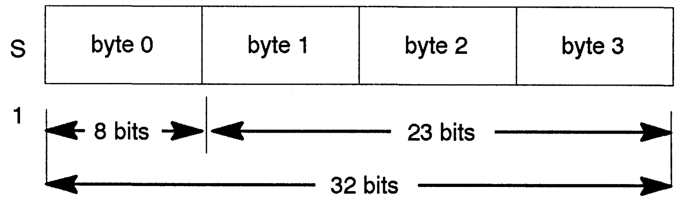

Figure 9–5 Byte layout for single-precision floating-point numbers

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). (This subject is discussed in *IEEE Standard for Binary Floating-Point Arithmetic*, cited at the end of this chapter.) According to IEEE specifications, the "NaN" (not a number) is system-dependent and should not be used externally.

Double-precision floating-point data types

The standard defines the encoding for the double-precision floating-point data type "double" (64 bits or 8 bytes). The encoding used is the IEEE standard for normalized double-precision floating-point numbers. 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 numbers respectively. One bit.

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

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

Therefore, the floating-point number is described as follows:

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

Figure 9–6 illustrates how double-precision floating-point numbers are declared.

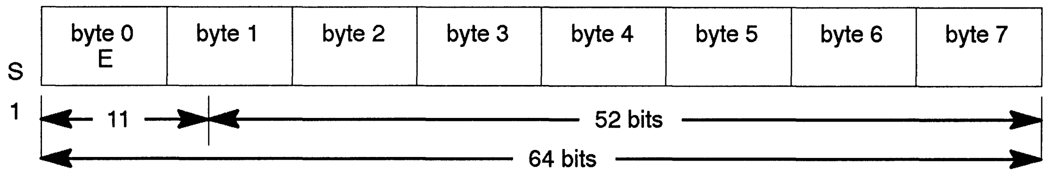

Figure 9–6 Byte layout for double-precision floating-point numbers

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 denormalized numbers (underflow). 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 0 bytes, r, to make the total byte count of the opaque object a multiple of 4.

Figure 9–7 illustrates how fixed-length opaque data is declared.

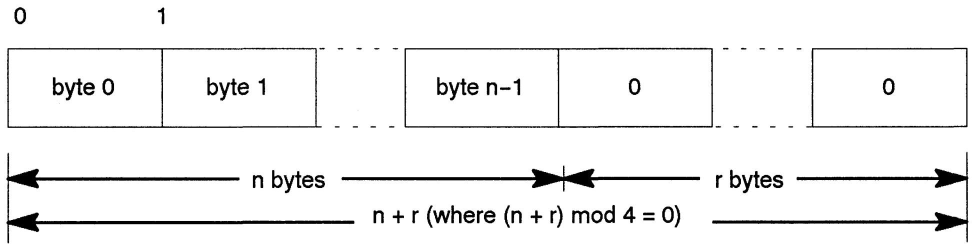

Figure 9–7 Byte layout for fixed-length opaque data

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). Byte n–1 is followed by enough (0 to 3) residual 0 bytes, r, to make the total byte count a multiple of 4. 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>;
```

Figure 9–8 illustrates how variable-length opaque data can be declared.

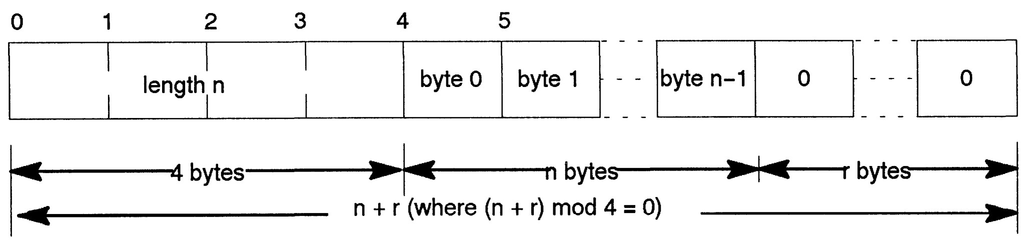

Figure 9–8 Byte layout for variable-length opaque data

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 4, then the *n* bytes are followed by enough (0 to 3) residual 0 bytes, *r*, to make the total byte count a multiple of 4. 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 filename can be no longer than 255 bytes, as follows:

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

Figure 9–9 illustrates how a string can be declared.

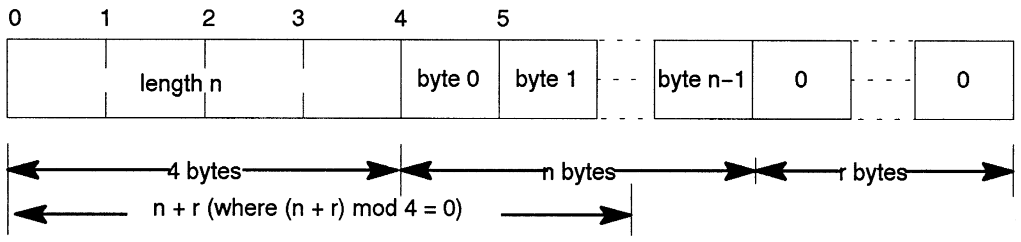

Figure 9–9 Byte layout for a string

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 may vary in its length.

Figure 9–10 illustrates how fixed-length arrays are declared.

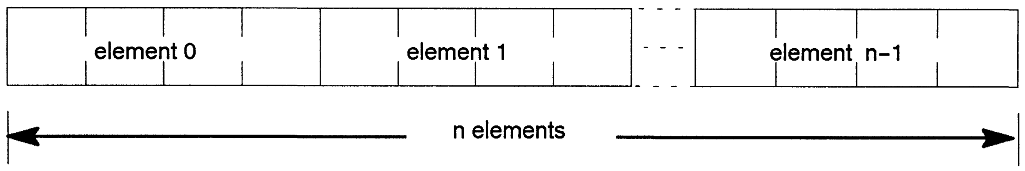

Figure 9–10 Byte layout for fixed-length arrays

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. Figure 9–11 illustrates how a counted array is declared.

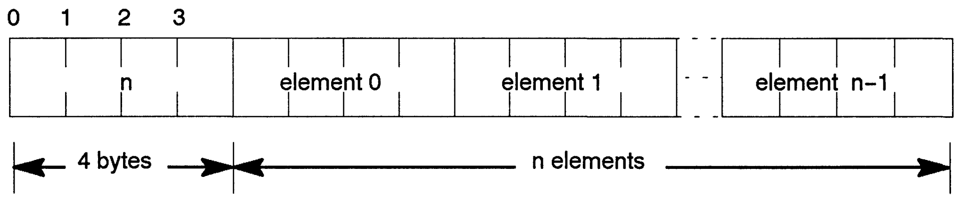

Figure 9–11 Byte layout for counted arrays

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. Figure 9–12 illustrates how structures are declared.

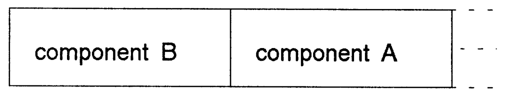

Figure 9–12 Components of a structure

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. Figure 9–13 illustrates how discriminated unions are encoded.

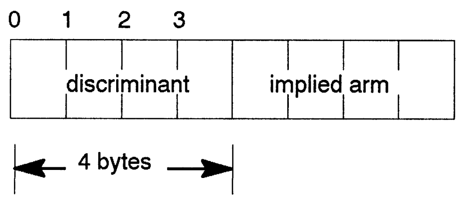

Figure 9–13 Layout for encoding a discriminated union

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 as follows:

```
void;
```

Figure 9–14 illustrates how voids are declared.

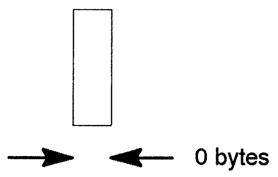

Figure 9–14 Layout for voids

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 data type

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

```
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 were 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
        };
```

This syntax is preferred because you do 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;
};
```

All 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, because 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 possible kind of data; 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.

Major features of the XDR standard

This section addresses some of the major features of XDR. It explains why the features are defined as they are.

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 quick 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. Because 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, because no higher level protocol is necessary for determining the byte-order.

Why does XDR use big-endian byte-order?

XDR uses big-endian byte-order because 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. We chose four 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 small advantages. 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 that is the XDR data description of the encoded data, and second, then the encoded data itself. Another way is to assign a value to all the types in XDR, and then define a universal type that takes this value as its discriminant, and for each value, describes the corresponding data type.

XDR language specification

This section shows how the XDR language is specified. It discusses notational conventions, lexical notes, syntax information, and syntax notes.

Notational conventions

This specification uses an extended Backus-Naur Form notation for describing the XDR language. Here is a brief description of the notation:

1. The characters |, (,), [,], ", and * are special.
2. Terminal symbols are strings of any characters surrounded by double quotation marks.
3. Non-terminal symbols are strings of non-special characters.
4. Alternative items are separated by a vertical bar (|).
5. Optional items are enclosed in square brackets ([]).
6. Items are grouped together by enclosing them in parentheses.
7. An asterisk following an item means zero 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

When using XDR, note the following:

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

Syntax information

This section presents a formal definition of what you can specify in the XDR language. This definition uses an extended Backus-Naur Form notation. For a brief description of the symbols used in the definition, see the section "Notational conventions" earlier in the chapter.

```
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

Some notes on XDR syntax are listed below.

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.

A sample XDR data description

This section contains, as an example, a short XDR data description of a function called *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 interpreter<MAXNAMELEN>;/*program interpreter*/ };


/*
 * 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)".

Table 9–1 shows how the LISP program would be encoded.

Table 9–1 How to encode a LISP program using XDR

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 interpreter = 4
24	6c 69 73 70	lisp	Interpreter 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

Works cited in text

Kernigan, Brian W., and Ritchie, Dennis M. *The C Programming Language.* Bell Laboratories, Murray Hill, New Jersey, 1978.

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

IEEE. Standard for Binary Floating-Point Arithmetic ANSI/IEEE Standard 754–1985. Institute of Electrical and Electronics Engineers, August 1985.

End of Chapter

A Using the automounter

You can mount file hierarchies shared through NFS using a different method: *automounting*. The **automount** program lets you mount and unmount remote directories as needed. Whenever a user on a client machine that is running the automounter invokes a command that needs to access a remote file or directory, the file system to which that file or directory belongs is mounted and remains mounted for as long as it is required. When a specified amount of time has elapsed without the file system being accessed, it is automatically unmounted. No mounting occurs at boot or run-level change time, and the user does not require the superuser password to mount a directory or use the **mount** or **umount** commands.

Mounting some file hierarchies with **automount** does not exclude mounting others with **mount**. Indeed, a diskless machine *must* mount / and **/usr** during system initialization: the automounter cannot be used to mount these file hierarchies.

The following subsections explain how the automounter works and how to set it up.

How the automounter works

IMPORTANT The material in this section is directed toward system administrators and programmers. Novice users may find the summary at the end of this section more helpful.

Unlike **mount**, the automounter does not consult the file **/etc/fstab** for a list of hierarchies to mount. Instead it consults a series of *direct* or *indirect* maps. The names of these maps can be passed to **automount** from the command line or from another (master) map.

The automounter mounts everything under the directory **/tmp_mnt** and provides a symbolic link from the requested mount point to the actual mount point under **/tmp_mnt**. For example, if a user wants to mount the remote directory **src** under **/usr/src**, the actual mount point is **/tmp_mnt/usr/src**. The name **/usr/src** becomes a symbolic link to that location. As with other mounts, a mount done through the automounter on a non-empty mount point obscures the original contents of the mount point while the mount is in effect.

The automounter's actions comprise two stages:

- The initial stage, boot time, when **rc.nfsserv** starts the automounter. **automount** must be specified in the entry **NFSSERV_DEMONS**, which is located in the file **/etc/nfs.params**.

- The mounting stage, when a user tries to access a file or directory on a remote machine.

At the initial stage, when **rc.nfsserv** invokes automount, it opens a UDP socket and registers it with the **portmap** service as an NFS server port. It then forks off a server daemon that listens for NFS requests on the socket. The parent process mounts the daemon at its mount points within the file system, as specified by the maps. Through the **mount**(2) system call, it passes the server daemon's port number and an NFS file handle that is unique to each mount point.

The arguments to the **mount**(2) system call vary according to the kind of file system. For NFS file systems, the call is in the format:

```
# mount ( "nfs", "/usr", &nfs_args );
```

where **&nfs_args** contains the network address for the NFS server. By having the network address in **&nfs_args** refer to the local process (the automount daemon), **automount** in effect deceives the kernel into treating it as if it were an NFS server. Instead, once the parent process completes its calls to **mount**(2) it exits, leaving the daemon to serve its mount points.

In the second stage, when the user requests access to a remote file hierarchy, the daemon intercepts the kernel NFS request and looks up the name in the map associated with the directory. Taking the locations (*server:pathname*) of the remote file system from the map, the daemon mounts the remote file system under the directory **/tmp_mnt**. It answers the kernel, telling it it is a symbolic link. The kernel sends an NFS READLINK request, and the automounter returns a symbolic link to the read mount point under **/tmp_mnt**.

The behavior of the automounter depends on whether the name is found in a direct or an indirect map:

- If the name is found in a direct map, the automounter emulates a symbolic link. It responds as if a symbolic link exists at its mount point. In response to a GETATTR request, it describes itself as a symbolic link. When the kernel follows up with a READLINK, it returns a path to the *real* mount point for the remote hierarchy in **tmp_mnt**.
- If, on the other hand, the name is found in an indirect map, the automounter emulates a directory of symbolic links, describing itself as a directory. In response to a READLINK request, it returns a path to the mount point in **/tmp_mnt**, and a **readdir**(3) of the automounter's mount point returns a list of the entries that are *currently* mounted.

Whether the map is direct or indirect, if the file hierarchy is already mounted and the symbolic link has been read recently, the cached symbolic link is returned immediately. Because the automounter is on the same host, the response is much faster than a READLINK request to a remote NFS server. On the other hand, if the file hierarchy is not mounted the mount is slightly delayed.

Summary

When automount is called from the command line or from **rc.nfsserv**, **automount** forks a daemon to serve each mount point in the maps to trick the kernel into believing the mount has taken place. The daemon lies dormant until a request is made to access the corresponding file hierarchy. At that time the daemon does the following:

1. Intercepts the request
2. Mounts the remote file hierarchy.
3. Creates a symbolic link between the requested mount point and the actual mount point under **/tmp_mnt**.
4. Passes the symbolic link to the kernel, and steps aside.
5. Unmounts the file hierarchy when a predetermined amount of time has elapsed since the link was last accessed (generally five minutes) and resumes its previous position.

Preparing the maps

A server is indifferent whether the files it shares are accessed through **mount** or **automount**. Therefore, you need not do anything differently on the server for **automount** than for **mount**.

A client, however, requires special files for the automounter. As previously mentioned, **automount** does not consult **/etc/fstab**; instead, it consults the map file(s) specified on the command line (see "Invoking **automount**," below). All automounter maps are located in the directory **/etc**. Their names begin with the prefix **auto**.

The three types of automount maps are:

- master
- indirect
- direct

Each map type is described below.

The master map

Each line in a master map, by convention called **/etc/auto.master**, has the following syntax:

```
Mount-point          Map       [ Mount-options ]
```

where:

- *mount-point* is the full pathname of a directory.

- *map* is the name of the map the automounter must use to locate the mount points and locations.
- *mount-options* is a comma-separated list of options that regulates the mounting of the entries in the map, unless the map entries themselves list other options.

A line whose first character is a pound sign (#) is treated as a comment: everything following it until the end of line is ignored. A backslash character (\) at the end of line splits long lines into shorter ones.

Direct and indirect maps

Lines in direct and indirect maps have the syntax:

```
key        [ mount-options ]        location
```

where

- *key* is the pathname of the mount point.
- *mount-options* are the options you want to apply to the mount.
- *location* is the location of the resource, specified as *server:pathname.*

As in the master map, a line whose first character is the pound sign character (#) is treated as a comment; everything following it until end of line is ignored. A backslash character (\) at the end of line splits long lines into shorter ones.

The only formal difference between a direct and an indirect map is that the key in a direct map is a full pathname, while in an indirect map it is a simple filename (without slashes). For example, the following is an entry in a direct map:

```
/usr/man        -ro,intr        goofy:/usr/man
```

and the following is an entry in an indirect map:

```
parsley        -ro,intr        veggies:/usr/greens
```

As you can see, the *key* in the sample indirect map requires more information: Where is the mount point **parsley** located? You must either provide that information at the command line or through another map. For instance, if the line in the second example is part of a map called **/etc/auto.veggies**, you must call it as:

```
automount       /veggies   /etc/auto.veggies
```

or specify, in the master map:

```
/veggies        /etc/auto.veggies    -ro,soft,nosuid
```

In either case, you are associating a mount directory (**veggies**) with the entries (**parsley**, in this case) in the indirect map **/etc/auto.veggies**. The result is that the hierarchy **/usr/greens** from the machine **veggies** is mounted on **/veggies/parsley** when required.

Writing a master map

As previously stated, the syntax for each line in the master map is:

```
Mount-point         Map        [ Mount-options ]
```

A typical **auto.master** file might contain the following entries:

```
#Mount-point   Map                 Mount-options
/net           -hosts
/home          /etc/auto.home      -rw,intr
/-             /etc/auto.direct    -ro,intr
```

The automounter recognizes some special mount points and maps, explained below.

Mount point /-

In the example above, the mount point /- is a filler that the automounter takes as a directive not to associate the entries in /etc/**auto.direct** with any directory. Rather, the mount points are those provided in the map. (Remember, in a direct map the key is a full pathname.)

Mount point /home

The mount point /**home** is the directory under which the entries listed in /etc/**auto.home** (an indirect map) are to be mounted. That is, they will be mounted under /**tmp_mnt**/**home**. A symbolic link will be provided between /**home**/*directory* and /**tmp_mnt**/**home**/*directory*.

Mount point /net

Finally, the automounter mounts under the directory /**net** all the entries under the special map -**hosts**. This is a built-in map that does not use any external files except the hosts database /etc/**hosts** (or **hosts.byname** NIS map). Because the automounter does not mount the entries until needed, the specific order of the entries is unimportant. Once the automount daemon is in place, the command:

```
example   $  cd /net/gumbo
```

changes directory to the top of the hierarchy (that is, the root file system) of the machine **gumbo** as long as the machine is in the hosts database. The user may not, however, view the files and directories under /**net**/**gumbo**. This is because the automounter can mount only the shared file systems of host **gumbo**, in accordance with the restrictions placed on the sharing.

When the preceding command is issue, the automounter performs the following tasks:

1. **ping**s the null procedure of the server's mount request to see if it's active.
2. Requests the list of shared hierarchies from the server.
3. Sorts the shared list according pathname length to ensure proper mounting order:

   ```
   /usr/src
   /export/home
   /usr/src/sccs
   /export/root/blah
   ```

4. Proceeds down the list, mounting all the file systems at mount points in **/tmp_mnt**, creating mount points at needed.
5. Returns a symbolic link that points to the top of the recently mounted hierarchy. Note that the automounter must mount all the file systems that the server in question advertises for sharing. For example, in the following request:

```
example  $ ls /net/gumbo/usr/include
```

the automounter mounts *all* of **gumbo**'s shared file systems, not just **/usr**.

In addition, the unmounting that occurs automatically after a specified period takes place from the bottom up. If one of the directories at the top of the list is busy, the automounter must remount the entire hierarchy and try again later.

Nevertheless, the **-hosts** special map provides a convenient way for users to access directories on many hosts without using **rlogin** or **rsh**. (These remote commands must establish communication through the network each time they are invoked.) Also, system administrators do not have to modify individual /**etc/fstab** files or mount directories manually.

Note that both /**net** and /**home** are conventional names. The automounter creates them if they do not exist.

Writing an indirect map

The syntax for an indirect map is:

```
key         [ mount-options ]         location
```

where *key* is the name (not the full pathname) of the directory to be used as mount point. Once the key is obtained by the automounter, it is suffixed to the mount point associated with it either from the command line or by the master map that invokes the indirect map.

Consider this entry in the master map presented earlier:

```
/home        -rw,intr         /etc/auto.home
```

Here **/etc/auto.home** is the name of the indirect map that will contain the entries to be mounted under **/home**.

A typical **auto.home** map might contain the following:

```
#key      mount-options   location
willow                    willow:/home/willow
cypress                   cypress:/home/cypress
poplar                    poplar:/home/poplar
pine                      pine:/export/pine
apple                     apple:/export/home
ivy                       ivy:/home/ivy
peach     -rw,nosuid      peach:/export/home
```

Assume that the preceding map is located on host **oak**. If user *laura* has an entry in the password database specifying her home directory as **/home/willow/laura**, whenever she logs into machine **oak** the automounter mounts (as **/tmp_mnt/home/willow**) the directory **/home/willow** residing on machine **willow**. If one of the directories indeed is **laura**, she will be in her home directory, which is mounted as read/write and interruptible.

Suppose, however, that *laura*'s home directory is **/home/peach/laura**. Whenever she logs into the machine **oak**, the automounter mounts **/export/home** from peach under **/tmp_mnt/home/peach**. Her home directory is mounted as **read/write**, nosuid. Any option in the file entry overrides all options in the master map or entered on the command line.

Now, assume the following conditions occur:

- User *laura*'s home directory is listed in the password database as **/home/willow/laura**.
- Machine **willow** shares its **home** hierarchy with the machines mentioned in **auto.home**.
- A copy of the same **auto.home** and password databases resides on each of these machines.

Furthermore, *laura* now can enter the command:

```
%  cd ~brent
```

and the automounter will mount *brent*'s home directory for her (if all permissions apply).

On a network without NIS, all relevant databases (such as **/etc/passwd**) on all systems on the network must be changed to accomplish this. If NIS is running, all relevant databases must be propagated throughout the network.

Writing a direct map

The syntax for a direct map is identical to that for an indirect map:

```
key        [ mount-options ]        location
```

where:

- *key* is the *full* pathname of the mount point. (Remember that in an indirect map this is not a full pathname.)
- *mount-options* are optional but, if present, override—for the entry in question—the options of the calling line, if any, or the defaults. (See "Invoking **automount**," below.)
- location is the location of the resource, specified as *server:pathname.*

Of all the maps, the entries in a direct map most closely resemble, in their simplest form, what their corresponding entries in /**etc**/**fstab** might look like. An entry that appears in /**etc**/**fstab** as:

```
dancer:/usr/local    - /usr/local/tmp nfs - YES ro
```

appears in a direct map as:

```
/usr/local/tmp -ro dancer:/usr/local
```

The following is a typical /**etc**/**auto.direct** map:

```
/usr/local \
                    /bin      -ro,soft       ivy:/export/local/sun3  \
                    /share    -ro,soft       ivy:/export/local/share  \
                    /src      -ro,soft       ivy:/export/local/src
/usr/man                      -ro,soft       oak:/usr/man \
                                             rose:/usr/man \
                                             willow:/usr/man
/usr/games                    -ro,soft       peach:/usr/games
/usr/spool/news               -ro,soft       pine:/usr/games
/usr/frame                    -ro,soft       redwood:/usr/frame2.1  \
                                             balsa:/export/frame
```

This map has a couple of unusual features, which are addressed in the next two subsections.

Multiple mounts

Multiple mounts can be hierarchical. When file systems are mounted hierarchically, each file system is mounted on a subdirectory within another file system. When the root of the hierarchy is referenced, the automounter mounts the entire hierarchy. The concept of *root* here is important. The symbolic link returned by the automounter to the kernel request is a path to the mount root. This is the root of the hierarchy that is mounted under /**tmp_mnt**. To be accurate, this mount point should be specified as:

```
parsley   /    -ro,intr       veg:/usr/greens
```

In practice, however, it is not specified because in the simple case of a single mount, it is assumed that the location of the mount point is at the mount root or "/." Thus it is preferable to enter:

```
parsley   -ro,intr        veg:/usr/greens
```

The mount point specifications, however, becomes important when mounting a hierarchy: here the automounter must have a mount point for each mount within the hierarchy. The example above illustrates multiple, non-hierarchical mounts under **/usr/local** when the latter is already mounted.

The following illustration shows a true hierarchical mounting:

```
/usr/local \
                /        -rw,intr   peach:/export/local \
                /bin     -ro,soft   ivy:/export/local/sun3 \
                /share   -rw,intr   willow:/usr/local/share \
                /src     -ro,intr   oak:/home/jones/src
```

The mount points in this example are /, **/bin**, **/share**, and **src**. These mount points are relative to the *mount* root, not the host's file system root. The first entry in the example above has / as its mount point. It is mounted *at* the mount root. There is no requirement that the first mount of a hierarchy be at the mount root. The automounter will execute **mkdir** commands to build a path to the first mount point if it is not at the mount root.

> **IMPORTANT** A true hierarchical mount can be problematic if the server for the root of the hierarchy goes down. Any attempt to unmount the lower branches will fail, since the unmounting must proceed through the mount root, which also cannot be unmounted while its server is down.

Multiple locations

In the example used for a direct map, repeated here:

```
/usr/local \
                  /bin      -ro,soft       ivy:/export/local/sun3 \
                  /share    -ro,soft       ivy:/export/local/share \
                  /src      -ro,soft       ivy:/export/local/src
/usr/man                    -ro,soft       oak:/usr/man \
                                           rose:/usr/man \
                                           willow:/usr/man
/usr/games                  -ro,soft       peach:/usr/games
/usr/spool/news             -ro,soft       pine:/usr/games
/usr/frame                  -ro,soft       redwood:/usr/frame1.3 \
                                           balsa:/export/frame
```

the mount points **/usr/man** and **/usr/frame** list more than one location (three for the first, two for the second). This means the mounting can be done from any of the replicated locations. This procedure makes sense only when you are mounting a hierarchy read-only, since theoretically you want some control over the locations of files you write or modify.

A good example are manual pages. In a large network, more than one server may export the current set of man pages. It doesn't matter which server you mount them from, as long as the server is up and running and sharing its file systems. In the preceding example, multiple mount locations are expressed as a list of mount locations in the map entry:

```
/usr/man -ro,soft oak:/usr/man  rose:/usr/man  willow:/usr/man
```

This entry also could be expressed as a comma-separated list of servers, followed by a colon (:) and the pathname (the pathname must be the same for all replicated servers):

```
     /usr/man -ro,soft oak,rose,willow:/usr/man
```

In this example you can mount the man pages from the servers *oak*, *rose* or *willow*. From this list of servers the automounter first selects those on the local network and **ping**s these servers. This launches a series of RPC requests to the null procedure of the mount service in each server. (The list does not imply any order.) The first server to respond is selected, and an attempt is made to mount from it.

This redundancy, very useful in an environment where individual servers may or may not be sharing their file systems, occurs only at mount time. The automounter performs no status checking of the mounted-from server after the mount takes place. If the server goes down while the mount is in effect, the file system becomes unavailable. In this event, one option is to wait five minutes until the auto-unmount takes place and try again. The automounter then will choose one of the available servers. A second option is to use the **umount** command, inform the automounter of the change in the mount table (as specified in the section "The mount table," below), and retry the mount.

Specifying subdirectories

The examples in this section rely on the sample **auto.home** indirect file used in the section "Writing an indirect map":

```
#key      mount-options   location
willow                    willow:/home/willow
cypress                   cypress:/home/cypress
poplar                    poplar:/home/poplar
pine                      pine:/export/pine
apple                     apple:/export/home
ivy                       ivy:/home/ivy
peach     -rw,nosuid      peach:/export/home
```

In this example, whenever a user wants to access a home directory in, for instance, **/home/willow**, all the directories under it are mounted. An alternate way to organize an **auto.home** file is by user name, as in:

```
#key      mount-options        location
john                           willow:/home/willow/john
mary                           willow:/home/willow/mary
joe                            willow:/home/willow/joe
```

This example assumes that home directories are of the form **/home/user** rather than **/home/server/user**. If a user now enters the following command:

```
% ls ~john ~mary
```

the automounter must perform the *equivalent* of the following actions:

```
mkdir /tmp_mnt/home/john
mount willow:/home/willow/john   /tmp_mnt/home/john
ln -s /tmp_mnt/home/john   /home/john

mkdir /tmp_mnt/home/mary
mount willow:/home/willow/mary   /tmp_mnt/home/mary
ln -s /tmp_mnt/home/mary   /home/mary
```

The whole syntax of a line in a direct or indirect mp is:

```
key  [ mount-option ]    server:pathname[:subdirectory]
```

You have used the form *server:pathname* to indicate the location. You also can specify the subdirectory:

```
#key      mount-options       location
john                          willow:/home/willow:john
mary                          willow:/home/willow:mary
joe                           willow:/home/willow:joe
```

In this case *john*, *mary* and *joe* are entries in the *subdirectory* field. Now when a user refers to *john*'s home directory, the automounter mounts **willow:/home/willow**. It then places a symbolic link between **/tmp_mnt/home/willow/john** and **/home/john**.

If the user then requests access to *mary*'s home directory, the automounter sees that **willow:/home/willow** is already mounted. It merely returns the link between **/tmp_mnt/home/willow/mary** and **/home/mary**. In other words, the automounter does only the following:

```
mkdir /tmp_mnt/home/john
mount willow:/home/willow   /tmp_mnt/home
ln -s /tmp_mnt/home/john   /home/john

ln -s /tmp_mnt/home/mary   /home/mary
```

In general, it is a good idea to provide a *subdirectory* entry in the *location* when different map entries refer to the same mounted file system from the same server.

Substitutions

A map specifying many subdirectories, for example:

```
#key      mount-options  location
john                     willow:/home/willow:john
mary                     willow:/home/willow:mary
joe                      willow:/home/willow:joe
able                     pine:/export/home:able
baker                    peach:/export/home:baker
     [ ... ]
```

may benefit from string substitutions. The ampersand (&) character substitutes for the key wherever it appears. Substituting the ampersand character, the preceding map now looks as follows:

```
#key      mount-options   location
john                      willow:/home/willow:&
mary                      willow:/home/willow:&
joe                       willow:/home/willow:&
able                      pine:/export/home:&
baker                     peach:/export/home:&
     [ ... ]
```

If the name of the server is the same as the key itself, for example:

```
#key      mount-options   location
willow                    willow:/home/willow
peach                     peach:/home/peach
pine                      pine:/home/pine
oak                       oak:/home/oak
poplar                    poplar:/home/poplar
     [ ... ]
```

the use of the ampersand results in the following map:

```
#key      mount-options   location
willow                    &:/home/&
peach                     &:/home/&
pine                      &:/home/&
oak                       &:/home/&
poplar                    &:/home/&
     [ ... ]
```

Note that the preceding entries have the same format. This permits you to use the asterisk (*) substitute character, reducing the map to:

```
*    &:/home/&
```

where each ampersand is substituted by the value of any given key. Once the automounter detects the asterisk it does not continue reading the map. Thus, the following map is possible:

```
#key      mount-options   location
oak                       &:/export/&
poplar                    &:/export/&
*                         &:/home/&
```

while in the following map, the last two entries are always ignored:

```
#key      mount-options   location
*                         &:/home/&
oak                       &:/export/&
poplar                    &:/export/&
```

Key substitutions may be used in a direct map, as the following example illustrates:

```
/usr/man       willow,cedar,poplar:/usr/man
```

can be written as:

```
/usr/man        willow,cedar,poplar:&
```

The ampersand substitution uses the entire key string; if the key in a direct map starts with a slash (/), the slash is duplicated. As a result, the automounter interprets the following:

```
/progs    &1,&2,&3:/export/src/progs
```

as

```
/progs    /progs1,/progs2,/progs3:/export/src/progs
```

Special characters

Under certain circumstances you may have to mount directories whose names may confuse the automounter's map parser. An example might be a directory called **:dk1**, which could result in an entry like:

```
/junk     -ro       vmsserver:rc0:dk1
```

The presence of the two colons (:) in the locations field confuses the automounter's parser. To avoid this confusion, use a backslash character (\) to prevent the second colon from being interpreted as a separator:

```
/junk     -ro       vmsserver:rc0\:dk1
```

You also can use double quotes, as in the following example, where they are used to hide the blank space in the name:

```
/smile    dentist:"front teeth"/smile
```

Environment variables

You can use the value of an environmental variable by prefixing its name with a dollar sign character ($). Braces also may be used to delimit the name of the variable from appended letters or digits.

The environmental variables can be inherited from the environment or can be explicitly defined with the **-D** command line option. For example, if you want each client to mount client-specific files in the network in a replicated format, you can create a map specifically for each client according to its name. Thus, the relevant line for host *oak* would be:

```
/mystuff  cypress,ivy,balsa:/export/hostfiles/oak
```

and for *willow*:

```
/mystuff  cypress,ivy,balsa:/export/hostfiles/willow
```

This scheme is practical within a small network, but maintaining such maps over a large network soon becomes infeasible. The solution in this case is to invoke the automounter with a command line similar to the following:

```
automount -D HOST='hostname' .....
```

with the following entry in the direct map:

```
/mystuff   cypress,ivy,balsa:/export/hostfiles/$HOST
```

Now each host locates its own files in the directory **mystuff**, and the task of administering and distributing the maps is simplified.

Invoking automount

Once the maps are written, you should make sure that **/etc/fstab** contains no equivalent entries, and that all entries in the maps refer to NFS shared files.

The syntax to invoke the automounter is:

automount [**-mnTv**] [**-D** *name=vvalue*] [**-f** *master-file*] [**-M** *mount-directory*] [**-t** *sub-**options*** [*directory map* [*-mount-options*]] ...

The **automount**(1M) man page describes all options. The sub-options are the same as those for a standard NFS mount: **bg** (background) and **fg** (foreground) do not apply.

Given the following set of three maps:

- auto.master

```
#Mount-point    Map                   Mount-options
/net            -hosts
/home           /etc/auto.home        -rw,intr
/-              /etc/auto.direct      -ro,intr
```

- auto.home

```
#key      mount-options        location
willow                         willow:/home/willow
cypress                        cypress:/home/cypress
poplar                         poplar:/home/poplar
pine                           pine:/export/pine
apple                          apple:/export/home
ivy                            ivy:/home/ivy
peach     -rw,nosuid           peach:/export/home
```

- auto.direct

```
/usr/local \
               /bin      -ro,soft  ivy:/export/local/sun3 \
               /share    -ro,soft  ivy:/export/local/share \
               /src      -ro,soft  ivy:/export/local/src
/usr/man                 -ro,soft  oak:/usr/man \
                                    rose:/usr/man \
                                    willow:/usr/man
/usr/games                -ro,soft  peach:/usr/games
```

```
/usr/spool/news         -ro,soft  pine:/usr/spool/news
/usr/frame              -ro,soft  redwood:/usr/frame2.1  \
                                  balsa:/export/frame
```

you can invoke the automounter (either from the command line or, preferably, from **rc.nfsserv**) in one of the following ways:

1. You can specify all arguments to the automounter without reference to the master map, as in:

```
automount /net -hosts /home /etc/auto.home -rw,intr\
          /- /etc/auto.direct -ro,intr
```

2. You can include the preceding line in the auto.master file, and instruct the automounter to open this file for instructions:

```
automount -f /etc/auto.master
```

3. You can specify more mount points and maps in addition to those referenced in the master map, as follows:

```
automount -f /etc/auto.master /src /etc/auto.src -ro,soft
```

4. You can nullify one of the entries in the master map. (This procedure is particularly useful if you use a map that you cannot modify and that does not meet your machine's requirements.)

```
automount -f /usr/lib/auto.master /home -null
```

5. You can replace one of the entries with your own:

```
automount -f /usr/lib/auto.master /home /myown/auto.home -rw,intr
```

In this example, the automounter first mounts all items in the map **/myown/auto.home** under the directory **/home**. Then, when it consults the master file **/usr/lib/auto.master** and reaches the line corresponding to **/home**, it simply ignores it (because it already has mounted on it).

Given the **auto.master** file in the previous example, the first two commands are equivalent, *as long as your network does not have a* distributed **auto.master** *file*. This file is available only on networks running NIS. If your network includes a distributed **auto.master** file, the second example must be modified in the following way to make it equivalent to example 1:

```
automount -m -f /etc/auto.master
```

The **-m** option instructs the automounter not to consult the master file distributed by NIS. If you don't run NIS, you do not have to specify the **-m** option; the automounter is silent when it does not find a distributed master file.

You can log in as superuser and type any of the above commands at shell level to start the automounter. Ideally, however, you should let **rc.nfsserv**. start the automounter. To do so, modify the **/etc/nfs.params** file as follows:

- Add **automount** to the entry **NFSSERV_DEMONS**.
- Modify **automount_arg** if you want to change the options supplied to **automount**.

The temporary mount point

The default name for all mounts is **/tmp_mnt**. Like the other names, this name is arbitrary: it can be changed at invocation time by the -M option. For example:

```
automount -M /auto ...
```

causes all mounts to take place under the directory **/auto**, which the automounter creates if necessary. Obviously you should not designate a directory that is in a read-only file system, as the automounter would not be able perform necessary file modifications.

The mount table

When the automounter mounts or unmounts a file hierarchy, it modifies **/etc/mnttab** to reflect the current mounting status. The automounter maintains an image of **/etc/mnttab** in memory, and refreshes this image whenever it performs a mounting or an automatic unmounting. If you use the **umount** command to unmount one of the automounted hierarchies (a directory under **/tmp_mnt**), you should force the automounter to re-read the **/etc/mnttab** file by entering the following command:

```
$ ps -ef | grep automount | egrep -v grep
```

This command returns the process ID of the automounter. The automounter is designed to re-read **/etc/mnttab** upon receiving a SIGHUP signal. To send it that signal, enter the following command:

```
% kill -1 PID
```

where *PID* stands for the process ID obtained from the previous **ps** command.

Modifying the maps

You can modify the automounter maps at any time. Recall, however, that the automounter consults the master and indirect maps only when it is invoked. Thus you must reboot the machine for modifications to maps to become effective.

On the other hand, changes to a direct map take effect when the automounter next mounts the modified entry. For example, suppose you modify the file /etc/**auto.direct** so that the directory /**usr**/**src** is now mounted from a different server. The new entry takes effect immediately (if /**usr**/**src** is not currently mounted) when you attempt to access the directory. If the directory is currently mounted, you can wait until the auto-unmounting occurs, then access it. If this procedure is not satisfactory, you can unmount the directory with the **umount** command, notify **automount** that the mount table has changed (see above, "The mount table"), then access it. The mounting should now be initiated from the new server.

Setting up automount in a client–server environment

Set up automount in a client–server environment as follows:

1. On the NIS master server, create a direct map file named /etc/**auto.fstab** for the clients to use. (See the section "Writing a direct map" earlier in this appendix for the syntax of direct map files.)

   If mount information already exists in a /**etc**/**fstab** file, use the script below to convert /**etc**/**fstab** to a /**etc**/**auto.fstab** file.

```
#!/bin/sh
#
# Call this tool fs2direct
#
# Copy /etc/fstab to /etc/fstab.tmp
#
# Run fs2direct /etc/fstab.tmp > /etc/auto.fstab
#
# rm /etc/fstab/tmp
#
# Extract the remote file mount lines from /etc/fstab;
# (bypass local mount lines of the format /dev/dsk/??)
# rearrange the information into automount order;
# remove the bg option, which is not used by automount
# save the result as /etc/auto.fstab
#
if [ $# -ne 1 ]
then
    echo "Usage: fs2direct </etc/fstab>" >&2
fi

egrep "^[^#]*:/" $1 | awk 'BEGIN {FS =" "}
{print $2, " -"$4," ",$1}' |
sed -e 's/bg,//g' -e 's/,bg//g' -e 's/[^,]bg[^,]/ /g' |
sort

#
# End of tool fs2direct
#
```

IMPORTANT The section "Multiple mounts," earlier in this chapter, describes a shorthand notation equivalent to the output of the tool shown above. You do not need to modify the **/etc/auto.fstab** file produced by the tool.

2. Enter the directory **/etc/yp** and type the following command:

   # **make auto.fstab** ↵

3. On each NIS client, add the line below to the **/etc/auto.master** file. (If the file does not exist, create it.)

   ```
   /- auto.fstab
   ```

   This line tells automount that the **auto.fstab** file is in direct map file format. (See "Writing a direct map" earlier in this chapter for information on direct file format.)

4. For each line in **auto.fstab**, remove the corresponding line from **/etc/fstab**. To see the relevant lines in **/etc/fstab**, type:

   # **ypcat auto.fstab** ↵

5. Add **automount** to **NFSSERV_DEMONS** in the file /etc/**nfs.params**.
6. Reboot your system.

   IMPORTANT You must halt the system and reboot. Taking the system down to single-user mode and back up to init 3 is not sufficient.

Error messages

This section explains the error messages generated when the automounter fails.

- **no mount maps specified**

The automounter was invoked with no maps to serve, and it cannot find the NIS **auto.master** map. This message appears only when the **-v** option is not specified. Check the command syntax or restart NIS.

- *mapname:* **Not found**

The required map cannot be located. This message appears only when the **-v** option is not specified. Check the spelling and pathname of the map name.

- **dir** *mountpoint* **must start with '/'**

The *mountpoint* exists but it is not a directory. Check the spelling and pathname of the mount point.

- **hierarchical mountpoint:** *mountpoint*

Automounter will not allow itself to be mounted within an automounted directory: use another strategy.

- **WARNING:** *mountpoint* **not empty!**

The *mountpoint* is not an empty directory. This warning appears only when the **-v** option is not specified; the previous contents of *mountpoint* will not be accessible.

- **Can't mount** *mountpoint: reason*

Automounter cannot mount itself at *mountpoint*. The *reason* should be self-explanatory.

- *hostname:file system* **already mounted on** *mountpoint*

Automounter has been mounted on an already mounted-on mountpoint and is attempting to mount the same file system there. This occurs if an entry in /**etc/fstab** also appears in an automounter map (either by accident or because the output of **mount -p** was redirected to **fstab**). Delete one of the redundant entries.

- **WARNING:** *hostname:file system* **already mounted on** *mountpoint*

The automounter is mounting itself on top of an existing mount point (warning only).

- **couldn't create** *directory: reason*

Couldn't create the specified directory: the *reason* should be self-explanatory.

- **bad entry in map** *mapname "map entry"*
- **map** *mapname*, **key** *map key:* **bad**

The automounter cannot interpret the map entry. Recheck the entry; perhaps it contains characters that need escaping.

- *mapname: yp_err*

Error in looking up an entry in an NIS map.

- *hostname:* **exports:** *rpc_err*

Error retrieving share list from hostname, indicating a server or network problem.

- **host** *hostname* **not responding**
- *hostname: filesystem* **server not responding**
- **Mount of** *hostname:filesystem* **on** *mountpoint: reason*

These messages appear after the automounter attempted to mount from *hostname* but either got no response or failed. This indicates a server or network problem.

- *mountpoint - pathname from hostname: absolute symbolic link*

While mounting a hierarchy, the automounter has detected that *mountpoint* is an absolute symbolic link (it begins with "/"). The content of the link is *pathname*. This may have undesirable consequences on the client: the contents of the link may be /**usr**.

- **Cannot create socket for broadcast rpc:** *rpc_err*
- **Many_cast select problem:** *rpc_err*
- **Cannot send broadcast packet:** *rpc_err*
- **Cannot receive reply to many_cast:** *rpc_err*

These error messages indicate problems attempting to **ping** servers for a replicated file system, indicating a network problem.

- **tyrmany: servers not responding:** *reason*

No server in a replicated list is responding, indicating a network problem.

- Remount *hostname:filesystem* **on** *mountpoint:* **server not responding**

An attempted remount after unmount failed, indicating a server problem.

- **NFS server (pid*n* \f3@*mountpoint*) not responding still trying**

An NFS request made to the automount daemon with PID *n* serving *mountpoint* has timed out. The automounter may be overloaded or dead. If the condition persists after several minutes, reboot the client. The alternative is to exit all processes that use automounted directories (or, change to a non-automounted directory in the case of a shell), kill the current automount process, and restart it from the command line. If all else fails, reboot.

End of Appendix

B A sample nfs.params file

The file **/etc/nfs.params** contains default values for the variables you must set. A sample **nfs.params** file follows.

```
# /etc/nfs.params
#
#
#   The parameters for nfs/nis must be setup for your particular
#   system. Two kinds of variables in the /etc/nfs.params file
#   control the way NFS and NIS will be invoked and initialized
#   each time you change to an appropriate run level with init.
#
#   These variables are either _START arguments which determine
#   the services made available automatically, or _ARG variables
#   which set the parameters used by the services/demons when
#   they're started.
#
#   Each variable has a description of its purpose and a
#   recommended default value. See "Managing ONC/NFS and Its Facilities"
#   for information specific to each service/demon.

####################################################################
# REQUIRED  NIS  VALUES

   # IF you are using NIS, THEN you MUST set the domainname_ARG
   # There is NO default for this argument
   # Running NIS with no domainname is an ERROR causing NIS to FAIL.

domainname_ARG=""


   # You must indicate whether this is a NIS master, server or client.

   # Masters maintain NIS map sources and supply NIS map info to others.
   # Servers supply NIS map info to other NIS hosts on demand.
   # A client consumes NIS services without supplying any.

   # Set the ypserv_START to reflect the status of your host.

                 # the default value is: ypserv_START=""
                 # any value other than "MASTER" "SERVER" "CLIENT"
                 # is equivalent to the default, which turns NIS OFF!
                 # IF you are a MASTER, set to: ypserv_START="MASTER"
                 # IF you are a SERVER, set to: ypserv_START="SERVER"
                 # IF you are a CLIENT, set to: ypserv_START="CLIENT"
                 # a good default value is: ypserv_START="CLIENT"
ypserv_START=""
   # indicate whether this host is passwd master so yppasswdd demon
   # will start. See the nfs/nis manual for information on yppasswdd.

   # if this host is master for the NIS passwd map, set yppasswdd_ARG
   # so that it will update and build the passwd map correctly.

   # If your NIS passwd file is in /etc then "/etc/passwd -m passwd" will
   # suffice, however, if you keep your passwd file in say /etc/src then set it
   # to "/etc/src/passwd -m SRC_DIR=/etc/src passwd" (or appropriate value).
   # IF THIS IS NOT THE yppasswd MASTER, set yppasswdd_ARG to ""
```

```
yppasswdd_ARG=""

###################################################################
# OPTIONAL NFS  VALUES

   # The verbose_FLAG flag indicates whether or not verbose messages
   # should be displayed from the rc scripts.

             # the default value is: verbose_FLAG="false"

verbose_FLAG="false"

   # nfsserv starts the demons associated with NFS services
   #   (portmap, rwalld ruserd, mountd, nfsd).
   # nfsfs mounts remote nfs file systems listed in /etc/fstab

   # IF you intend to mount remote nfs file systems THEN
   #   nfsserv_START and nfsfs_START must both be true

              # the default value is: nfsserv_START="true"
              # any value other than "false" equals the default
nfsserv_START="true"

              # the default value is: nfsfs_START="true"
              # any value other than "false" equals the default
nfsfs_START="true"

              # Demons to be started/stopped by rc.nfsserv
              # choose from: "mountd nfsd lockd automount"
NFSSERV_DEMONS="mountd nfsd lockd"


   # number of demons to be started is based on the amount
   # of network (nfs) traffic.
   # the nfsds are used when acting as an NFS SERVER
   # the biods are used when acting as an NFS CLIENT and started by the
   #    dgux init scripts
   # See the manpage for details.

              # default value is: nfsd_ARG="8"
nfsd_ARG="8"

   # remote locking demons

              # default value is: lockd_sleep_ARG="15"
              # See the manpage for details.
lockd_sleep_ARG="15"

              # default value is: lockd_ARG="-g ${lockd_sleep_ARG:-0}"
lockd_ARG="-g ${lockd_sleep_ARG:-0}"

              # default value is: statd_ARG=""
statd_ARG=""


        # default value is: mountd_ARG=""
        # setting this to "-n" disallows mounting only from secure ports
        # see mountd(1m) for more details.
mountd_ARG=""

        # default value is: automount_ARG="-m -f /etc/auto.master"
automount_ARG="-m -f /etc/auto.master"
```

This sample file above sets the variables **nfsserv_START** and **nfsfs_START** to **true**, allowing the host to mount remote machines.

End of Appendix