Programming with Remote Procedure Calls (RPC)

Search
Advanced Search
Manual Page Search

What's New
New Features and Notes
Support and Updates

Help Topics
Backup and Restore
Command Line (Shells)
Compatibility
DOS and Windows
Desktops
Files and Directories
Filesystems
Hardware
Installation and Licensing
Internet and Intranet
Linux Kernel Personality
Mail and Messaging
Networking
OpenServer Kernel Personality
Printing
Security
System Handbook
System Management
Users and Groups

Developer Documentation
PostgreSQL documentation
Hardware and Driver Development
Software Development

UnixWare 7 Documentation

Welcome to the UnixWare 7 online documentation set. Using DocView, you can access all of the documentation that is currently installed on your system. DocView is based on an Apache server, and the documentation it serves can be viewed with any Internet browser.

You can navigate through the documentation by clicking on the options across the top of the page, by clicking on links on the left side of the page, or by searching.

You can use *Search* to find anything in the documentation set. Use *Advanced Search* to refine your query. Man pages can be located using *Manual Page Search*. See Using DocView for more information, including how to maintain the search database.

For more information:

- The Support Library is a searchable list of solutions to common troubleshooting issues.
- The Product Documentation web site serves the latest documentation available.
- The Compatible Hardware Web Page lists hardware known to work with UnixWare 7.
- The SCO home page provides useful information on product offerings and updates.

UnixWare 7 Release 7.1.4

SITE MAP

- DOC HOME
  - Search
- Advanced Search
- Manual Page Search

- **What's New**
  - **New Features and Notes**
    - **Release 7.1.4 new features**
    - **Release 7.1.3 new features**
    - **Release 7.1.2 new features**
    - **Release 7.1.1 new features**
    - **Release 7.1 new features**
    - **Release 7.0.1 new features**
    - **Release 7.0.0 new features**
    - **Development Kit Features**
  - **Support and Updates**
    - **Software Support and Updates**

- **Help Topics**
  - **Backup and Restore**
    - **Creating and using emergency recovery media**
  - **Command Line (Shells)**
    - **An overview of the system**
    - **Automating tasks with shell scripts**
      - **Manipulating text with sed**
      - **Regular expressions**
    - **Basic concepts**
    - **Customizing your environment**
    - **Editing files**
    - **Introduction to using the system**
    - **Process control and scheduling**
  - **Compatibility**
    - **Command and shell script compatibility**

- Linux Kernel Personality
  - Linux Kernel Personality (LKP)
    - LKP Quick Reference
    - Introduction to LKP
    - How LKP Works
    - System Management for LKP
    - Running Linux Applications under LKP
    - Developing Linux Applications
- Mail and Messaging
  - Mail and Messaging overview
  - The Pine MUA
  - Sendmail Operations Guide
  - Sendmail 8.12.9 Release Notes
  - The Vacation Notification Manager
- Networking
  - Configuring LAN connections
  - Configuring WAN connections
    - Administering the Systems file
    - Configuring outgoing PPP connections
    - Configuring incoming PPP connections
    - Troubleshooting PPP links
    - Configuring the Serial Line Internet Protocol
  - Understanding networking
  - Administering network services
  - Administering TCP/IP and Internet services
  - Secure IP (IPsec)
  - Configuring NetBIOS
  - Administering the Network Information Service (NIS)
  - Administering the Distributed File System (DFS)
  - Administering the Network File System (NFS)
  - Administering the Simple Network Management Protocol
  - Networking manual pages
    - Section 1Msnmp manual pages
    - Section 1bnu manual pages
    - Section 1ldap manual pages
    - Section 1nfs manual pages
    - Section 1nis manual pages
    - Section 1tcp manual pages
    - Section 1Mbnu manual pages
    - Section 1Mldap manual pages
    - Section 1Mnfs manual pages

        - Trusted facility management

- Developer Documentation

    - PostgreSQL documentation
    - Hardware and Driver Development
        - Uniform Driver Interface (UDI)
            - UDI 1.01 Specification
            - UDI Driver Writer's Guide
            - UDI Implementer's Guide
    - Software Development
        - JPEG Documentation
            - Usage instructions for JPEG utilities
            - Advanced usage instructions for JPEG wizards only
            - How to use the JPEG library in your own programs
            - Sample code for calling the JPEG library

Manual Pages

Manual Page:

Search

Section (optional):

Search descriptions also

Display List of Manpages

Most commands and functions are documented on reference pages called "Manual pages", commonly known as "man" pages. Man pages provide quick access to detailed information about one command or function. The man pages are grouped into sections that are numbered from 1 to 8.

To search for a man page, enter the name of the command or function and click Search. If the command exists in more than one section, you will be presented with a list to choose from. If you specify a Section number, then only that section will be searched.

If you don't know the exact name, you can check the **Search descriptions also** box. That will scan the list of commands and descriptions for your search string and give you a list to choose from. If you want to search through the full text of man pages, use the DocView Search function.

If you select **Display List of Manpages**, you will see one long list of all the man pages installed on the system, sorted alphabetically by section. You can scroll through the list, or use the browser Find option to find a specific man page.

Available GNU Info topics:

- As: (as). The GNU assembler.
- Autoconf: (autoconf). Create source code configuration scripts.
- automake: (automake). Making Makefile.in's
- Bfd: (bfd). The Binary File Descriptor library.
- Binutils: (binutils). The GNU binary utilities.
  ar: (binutils)ar. Create, modify, and extract from archives
  nm: (binutils)nm. List symbols from object files
  objcopy: (binutils)objcopy. Copy and translate object files ...
- bison: (bison). GNU Project parser generator (yacc replacement).
- Chill:: Chill compiler
- configure: (configure). The GNU configure and build system
- Cpp: (cpp). The GNU C preprocessor.
- g77: (g77). The GNU Fortran compiler.
- gasp: (gasp). The GNU Assembler Preprocessor
- Gawk: (gawk). A Text Scanning and Processing Language.
- gcc: (gcc). The GNU Compiler Collection.
- Gdb-Internals: (gdbint). The GNU debugger's internals.
- Gdb: (gdb). The GNU debugger.
- gprof: (gprof). Profiling your program's execution
- gzip.info
- Info: (info). Documentation browsing system.
- m4: (m4). A powerful macro processor.
- Make: (make). Remake files automatically.
- Standalone info program: (info-stnd). Standalone Info-reading program.
- Standards: (standards). GNU coding standards.
- Texinfo: (texinfo). The GNU documentation format.
  install-info: (texinfo)Invoking install-info. Update info/dir entries.
  texi2dvi: (texinfo)Format with texi2dvi. Print Texinfo documents.
  texindex: (texinfo)Format with tex/texindex. Sort Texinfo index files. ...

DocView Search

Search

Match: All

Format: Short

Search these languages:

- ☐ Deutsch
- ☑ English
- ☐ Français

You can use DocView Search to search through all of the installed documentation on the system. This includes guides, HOWTOs, man pages, info files, and miscellaneous text files.

- Searches are not case sensitive. You can enter upper, lower or mixed case and get the same result.
- The search will find all of the ending variations a word might have. For example, if you enter ``configure'', you will find ``configure'', ``configures'', ``configured'' and ``configuring''.
- You cannot use wild card characters (*?).
- You cannot search for phrases -- phrases are treated as a list of individual words with no relationship to each other.

Choose the word(s) you search for carefully and make them as specific as possible. This makes the search quicker, and produces a smaller, more relevant list of results.

Options:

Match: All (default)
A match must include all of the words, but each word can appear separately, and might be found anywhere in the document.

Match: Any
A document can have any one of the words you type -- it need not have more than one.

Match: Expression
You can enter ``and'', ``or'', ``not'' operators to form a logical expression. These are especially useful for narrowing your search. Examples:

- performance and tuning: finds all documents that contain both words
- adapters or controllers: finds all documents that contain either word
- alias not mail: finds all documents that contain the word ``alias'' but not the word ``mail''

Format: Short (default)
Only the titles of matching documents are shown in the results list.

Format: Long

Titles are shown along with a short excerpt from the file.

DocView Print Service

◉ PDF
○ PostScript

Book title

3 Heading levels in table of contents

Submit

The DocView Print Service can generate a PDF or PostScript book from HTML content. You select the content by checking one or more checkboxes below. When you are finished, submit the form and you will get a confirmation screen indicating the size of the book. You can then download the generated book.

You can choose between Portable Document Format (PDF) or PostScript output. You can supply a book title, which will appear on the title page of the generated book. You can also select the number of heading levels to appear in the book's table of contents. For example, manual page sections work best when set to "2" heading levels.

The more content you select, the longer it will take to assemble the book. The maximum book size is 1.4 MB of HTML text (about 600 pages).

- What's New
  - New Features and Notes
    - ☐ Release 7.1.4 new features
    - ☐ Release 7.1.3 new features
    - ☐ Release 7.1.2 new features
    - ☐ Release 7.1.1 new features
    - ☐ Release 7.1 new features
    - ☐ Release 7.0.1 new features
    - ☐ Release 7.0.0 new features
    - ☐ Development Kit Features
  - Support and Updates
    - ☐ Software Support and Updates
- Help Topics
  - Backup and Restore
    - ☐ Creating and using emergency recovery media
  - Command Line (Shells)
    - ☐ An overview of the system
    - Automating tasks with shell scripts

- ☐ Manipulating text with sed
- ☐ Regular expressions
- ☐ Basic concepts
- ☐ Customizing your environment
- ☐ Editing files
- ☐ Introduction to using the system
- ☐ Process control and scheduling

- Compatibility
  - ☐ Command and shell script compatibility
  - ☐ OpenServer 5 and UnixWare 2 Commands
  - ☐ Running SCO UnixWare 2.1.X applications
  - ☐ UnixWare standards conformance
- DOS and Windows
  - ☐ DOS command equivalents
  - ☐ Merge User's Guide
  - ☐ Merge man pages
  - ☐ Working with DOS
- Desktops
  - ☐ Using DocView
  - ☐ Administering your system with SCOadmin
  - ☐ Ghostscript PS/PDF Interpreter and Viewer
  - ☐ Using the DocView Documentation Server
  - ☐ X client manual pages
  - ☐ X server manual pages
- Files and Directories
  - ☐ About files and directories
  - ☐ Editing files
  - ☐ Working with DOS
- Filesystems
  - ☐ Administering filesystems
  - ☐ Managing filesystem types
  - ☐ Solving filesystem problems
  - ☐ Overview of VERITAS Documentation
  - ☐ Veritas File System (vxfs) Administrator's Guide
  - ☐ Online Data Manager overview and installation
  - ☐ Volume Manager Administrator's Guide
  - ☐ Volume Manager User's Guide
- Hardware
  - ☐ Hardware configuration overview

- ☐ Mass storage devices overview
- ☐ Using the Device Configuration Utility (DCU)
- ☐ Adding ISDN devices
- ☐ Adding hard disks
- ☐ Adding memory (dynamically addable memory)
- ☐ Adding and configuring modems
- ☐ Adding or removing hardware controllers
- ☐ Adding tape drives
- ☐ Configuring LAN connections
- Configuring WAN connections
  - ☐ Administering the Systems file
  - ☐ Configuring outgoing PPP connections
  - ☐ Configuring incoming PPP connections
  - ☐ Troubleshooting PPP links
  - ☐ Configuring the Serial Line Internet Protocol
- ☐ Configuring Plug and Play devices
- ☐ Configuring audio adapters
- ☐ Configuring serial ports
- ☐ Configuring video adapters
- ☐ Hot adding or removing SCSI storage devices
- ☐ Managing Hot Plugable controllers
- ☐ Mouse administration
- ☐ Working with diskettes
- ☐ Configuration parameters
- ☐ Network driver configuration notes
- ☐ UDI Overview and Configuration
- ☐ Section 7 manual pages

- Installation and Licensing
  - ☐ Getting Started Guide
  - ☐ Installing, managing, and removing software
  - ☐ Licensing and registering products
- Internet and Intranet
  - ☐ Configuring LAN connections
  - Configuring WAN connections
    - ☐ Administering the Systems file
    - ☐ Configuring incoming call services
    - ☐ Configuring outgoing PPP connections
    - ☐ Configuring incoming PPP connections
    - ☐ Configuring the Point-to-Point Protocol (PPP)

- ☐ Troubleshooting PPP links
- ☐ Configuring the Serial Line Internet Protocol

- ☐ Administering TCP/IP and Internet services
- ☐ Secure IP (IPsec)
- ☐ Configuring packet filters
- ☐ Configuring an Address Allocation Server (AAS)
- ☐ Configuring a Dynamic Host Configuration Protocol (DHCP)
- ☐ Configuring File Transfer Protocol (FTP) servers
- ☐ Configuring Domain Name System (DNS) servers
- ☐ BIND 9 Administrator's Guide
- ☐ Configuring Network Time Protocol (NTP) service
- ☐ Configuring name service, NIS, and NTP clients
- ☐ Squid Documentation

- Linux Kernel Personality
  - Linux Kernel Personality (LKP)
    - ☐ LKP Quick Reference
    - ☐ Introduction to LKP
    - ☐ How LKP Works
    - ☐ System Management for LKP
    - ☐ Running Linux Applications under LKP
    - ☐ Developing Linux Applications
- Mail and Messaging
  - ☐ Mail and Messaging overview
  - ☐ The Pine MUA
  - ☐ Sendmail Operations Guide
  - ☐ Sendmail 8.12.9 Release Notes
  - ☐ The Vacation Notification Manager
- Networking
  - ☐ Configuring LAN connections
  - Configuring WAN connections
    - ☐ Administering the Systems file
    - ☐ Configuring outgoing PPP connections
    - ☐ Configuring incoming PPP connections
    - ☐ Troubleshooting PPP links
    - ☐ Configuring the Serial Line Internet Protocol
  - ☐ Understanding networking
  - ☐ Administering network services
  - ☐ Administering TCP/IP and Internet services
  - ☐ Secure IP (IPsec)

- ☐ Configuring NetBIOS
- ☐ Administering the Network Information Service (NIS)
- ☐ Administering the Distributed File System (DFS)
- ☐ Administering the Network File System (NFS)
- ☐ Administering the Simple Network Management Protocol
- Networking manual pages
  - ☐ Section 1Msnmp manual pages
  - ☐ Section 1bnu manual pages
  - ☐ Section 1ldap manual pages
  - ☐ Section 1nfs manual pages
  - ☐ Section 1nis manual pages
  - ☐ Section 1tcp manual pages
  - ☐ Section 1Mbnu manual pages
  - ☐ Section 1Mldap manual pages
  - ☐ Section 1Mnfs manual pages
  - ☐ Section 1Mnis manual pages
  - ☐ Section 1Mtcp manual pages
  - ☐ Section 4bnu manual pages
  - ☐ Section 4ldap manual pages
  - ☐ Section 4nis manual pages
  - ☐ Section 4snmp manual pages
  - ☐ Section 4tcp manual pages
  - ☐ Section 5tcp manual pages

- OpenServer Kernel Personality
  - ☐ OKP Features
  - ☐ Getting Started with OKP
  - ☐ Running Applications under OKP
  - ☐ Running Applications under OKP
- Printing
  - Advanced LP printer configuration
    - ☐ Advanced LP printer configuration
    - ☐ Configuring Hewlett-Packard network printers
  - ☐ Managing LP the print service
  - ☐ Managing LP print jobs
  - ☐ Advanced LP printer configuration
  - ☐ Configuring Hewlett-Packard network printers
  - ☐ Setting Up a USB Printer With LP
  - ☐ CUPS Documentation
  - ☐ HP Inkjet Printer Driver

- Security
  - Managing system security
    - ☐ Introduction to security
    - ☐ Security procedures
    - ☐ Administering privilege
    - ☐ Trusted facility management
  - ☐ Changing the system security profile
  - ☐ Understanding file protection
  - Auditing your system
    - ☐ Overview of the auditing subsystem
    - ☐ Installing the auditing subsystem
    - ☐ Configuring auditing
    - ☐ Auditable events
    - ☐ Maintaining the auditing system
    - ☐ Displaying audit trail information
    - ☐ Summary of auditable events and classes
  - ☐ Using system accounting
  - Security Features User's Guide (SFUG)
    - ☐ Introduction to security
    - ☐ Customizing your environment
  - Trusted Facility Manual (TFM)
    - ☐ Administering user accounts
    - ☐ Changing the system security profile
    - ☐ Understanding file protection
    - ☐ Using system accounting
- System Handbook
  - ☐ System Handbook
- System Management
  - Understanding system administration
    - ☐ Administering systems
    - ☐ Administering your system with SCOadmin
    - ☐ Performing basic system monitoring and tuning
    - ☐ System default files
    - ☐ System directories and files
  - Starting and stopping the system
    - ☐ Starting and stopping the system
    - ☐ Customizing UNIX system startup
  - Monitoring and tuning the system
    - ☐ Performing basic system monitoring and tuning
    - ☐ Managing system performance

- ☐ Process scheduling
- ☐ Managing dynamically loadable kernel modules
- ☐ Tunable parameters

- ☐ Analyzing system dumps using kcrash
- Enhanced Event Logging System
  - ☐ The Enhanced Event Logging System
  - ☐ Using EELS
- Customizing locale settings
  - ☐ Specifying the locale
  - ☐ Customizing device character mapping
- Troubleshooting your system
  - ☐ Troubleshooting system-level problems
  - ☐ Creating and using emergency recovery media

- Users and Groups
  - ☐ Administering user accounts
  - ☐ Customizing your environment
  - Managing system security
    - ☐ Introduction to security
    - ☐ Security procedures
    - ☐ Administering privilege
    - ☐ Trusted facility management

- Developer Documentation
  - ☐ PostgreSQL documentation
  - Hardware and Driver Development
    - Uniform Driver Interface (UDI)
      - ☐ UDI 1.01 Specification
      - ☐ UDI Driver Writer's Guide
      - ☐ UDI Implementer's Guide
  - Software Development
    - JPEG Documentation
      - ☐ Usage instructions for JPEG utilities
      - ☐ Advanced usage instructions for JPEG wizards only
      - ☐ How to use the JPEG library in your own programs
      - ☐ Sample code for calling the JPEG library

Programming with Remote Procedure Calls (RPC)

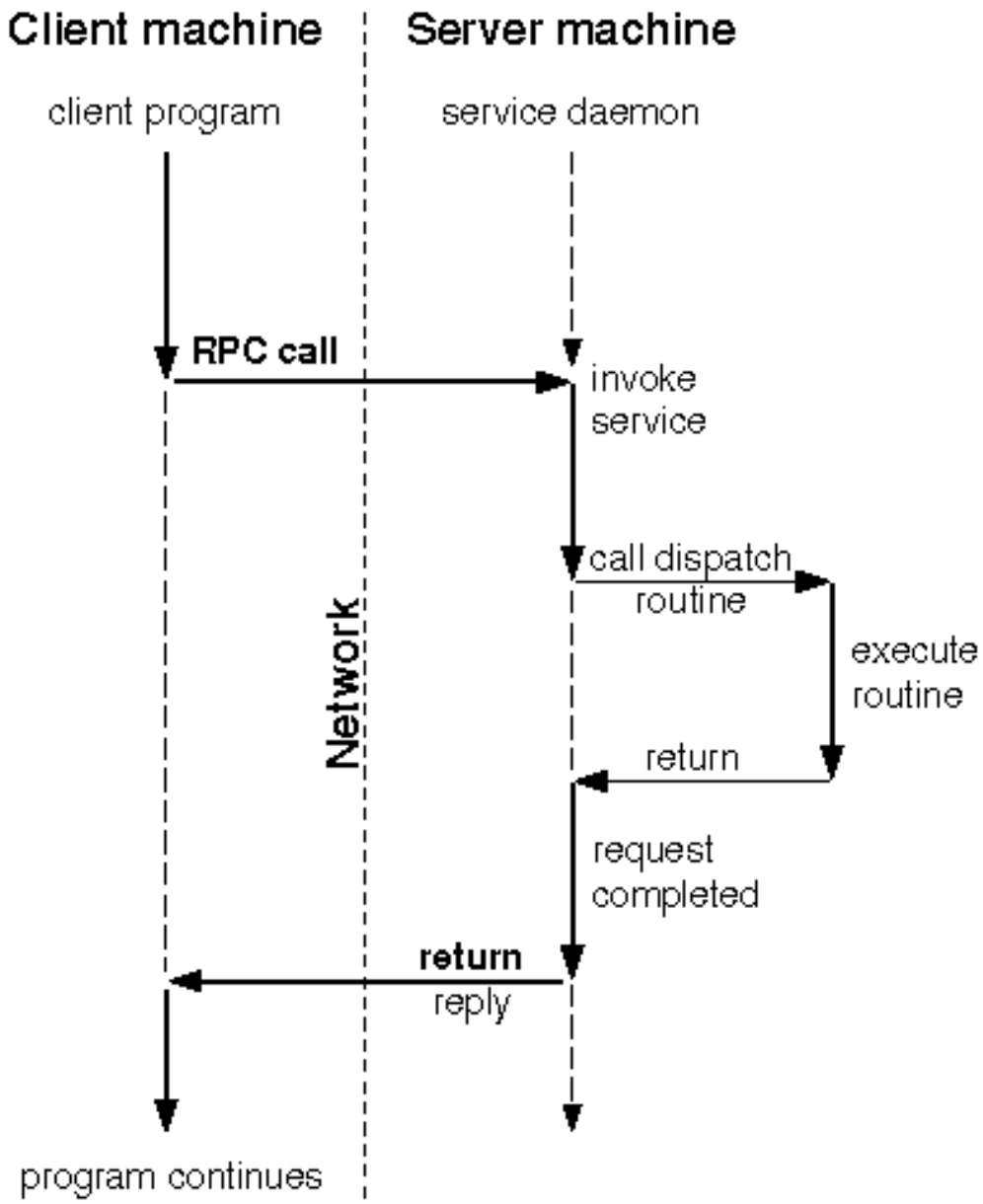
The Remote Procedure Calls (RPC) mechanism is a high-level communications paradigm for network applications. By use of RPC, programs on networked platforms can communicate with remote (and local) resources.

RPC allows network applications to use specialized kinds of procedure calls designed to hide the details of underlying networking mechanisms. RPC is transport-independent, able to take advantage of whatever kinds of networking mechanisms (such as TCP/IP or ISO) may be available. RPC implements a logical client-to-server communications system designed specifically for the support of network applications. Generic facilities, such as **rpcbind**(1Mtcp), associate network services with universal network addresses.

UnixWare 7 Release 7.1.4 - 27 April 2004

How RPC works

As shown in ``Network communication with the remote procedure call'', in RPC, a client makes a procedure call to send data packets to the server. When the packets arrive, the server calls a dispatch routine to perform whatever service is requested. When the dispatch routine returns, the server sends back a reply to the client.



Network communication with the remote procedure call

Programming with RPC produces programs that are designed to run within a client/server network model.

Such programs use RPC mechanisms to avoid the details of interfacing to the network, and provide network services to their callers without requiring that the caller be aware of the existence and function of the underlying network. For example, a program can simply call **rusers**(3N), a C routine that returns the number of users on a remote machine. The caller is not explicitly aware of using RPC -- the call to **rusers** is as simple as a call to **malloc**.

This section addresses only the C interface to RPC, but remote procedure calls can be made from any language. Note that although this section describes the use of RPC for communication between processes on different machines, RPC works just as well for communication between different processes on the same machine.

UnixWare 7 Release 7.1.4 - 27 April 2004

RPC versions and numbers

Each RPC procedure is uniquely identified by a program number, version number, and procedure number.

The program number identifies 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 does not have to be assigned.

To call a procedure to find the number of remote users, for example, you would look up the appropriate program, version and procedure number in a reference manual.

RPC programs should be assigned program numbers according to rules detailed in ``Program number assignment''.

Network selection

Network selection is a simple way by which users and applications may dynamically select transports, according to both their preferences and the available transports. It is based on two mechanisms, the **netconfig** (4bnu) database, which lists the transports available on the host and identifies them by type, and the optional environment variable **NETPATH**, which allows the user to specify preferences among the transports available in */etc/netconfig* that are acceptable to the application. For more details about network selection, see ``Network selection'' and the **getnetconfig**(3N) manual page.

NOTE: To create a service for a particular transport, an application must interface to RPC at a level below the ***top level***, that is, the level composed of **clnt_create** and its associated routines on the **rpc_clnt_create**(3rpc) manual page. Only then can it specify the types of transports that it prefers. See ``Remote Procedure Call programming'' for details about the various RPC levels.

The */etc/netconfig* file contains several lines, each of which corresponds to an available transport (see *netconfig* (4bnu)). These are some possible entries:

```
   # The Network Configuration File.
   #
   # Each entry is of the form:
   #
   # network_id semantics flags protofamily protoname device
nametoaddr_libs
   #
   ticlts tpi_clts v loopback - /dev/ticlts /usr/lib/straddr.so
   ticots tpi_cots v loopback - /dev/ticots /usr/lib/straddr.so
   ticotsord tpi_cots_ord v loopback - /dev/ticotsord /usr/lib/straddr.so
   starlan tpi_cots v osinet - /dev/starlan /usr/lib/straddr.so

   starlang tpi_clts v osinet - /dev/starlang /usr/lib/straddr.so
   tcp tpi_cots_ord v inet tcp /dev/tcp /usr/lib/tcpip.so

   udp tpi_clts v inet udp /dev/udp /usr/lib/tcpip.so

   icmp tpi_raw - inet icmp /dev/icmp /usr/lib/tcpip.so
   rawip tpi_raw - inet - /dev/rawip /usr/lib/tcpip.so
```

The following are a few points about */etc/netconfig*, and the applications interface to it:

- Each entry contains an identifier (the first field) which gives the network identifier by which the transport is commonly known.
- Each entry also contains a flag or set of flags (the third field) that identifies it by type -- the **v** flag, for example, identifies any transport that is ``visible."
- The last field names a run-time linkable module that contains the name-to-address translation routines associated with the transport. (See ``Name-to-address translation").
- The loopback transports are required for registering services with **rpcbind**. They are local transports, available only to local clients and servers, and hence are more secure than other transports.

The format of **NETPATH** is an ordered list of network identifiers separated by colons (:) (for example: **udp: tcp:starlan**). By setting **NETPATH**, the user can specify the order in which the application should try the various networks. If **NETPATH** is not set, the system defaults to all visible transports specified in */etc/netconfig*, in the order they appear.

NOTE: Applications can choose to ignore a user's **NETPATH**.

RPC divides selectable transports into the following types:

``netpath"
: Choose from those transports that have been specified in the **NETPATH** environment variable. If **NETPATH** is not set, the system defaults to all visible transports specified in */etc/netconfig*, in the order they appear.

``"
: (null) -- same as selecting ``netpath".

``visible"
: Choose those transports that have the visible flag (`v') set in their */etc/netconfig* entries.

``circuit_v"
: Same as ``visible", but restricted to connection-oriented transports.

``datagram_v"
: Same as ``visible", but restricted to connectionless transports.

``circuit_n"
: Choose from whatever is defined in **NETPATH**, but restrict to connection-oriented transports.

``datagram_n"

Choose from whatever is defined in **NETPATH**, but restrict to connectionless transports.

``udp''
(For backwards compatibility) -- specifies Internet User Datagram Protocol (UDP).

``tcp''
(For backwards compatibility) -- specifies Internet Transmission Control Protocol (TCP).

When a transport-dependent application begins execution, it begins by calling the **setnetconfig**, **getnetconfig**, and **endnetconfig** routines, using them to search */etc/netconfig* for a transport of appropriate type. This information is then stored in local data structures of type **struct netconfig** and is available for later use. **setnetconfig**, **getnetconfig**, and **endnetconfig** are described on the **getnetconfig**(3N) manual page; the Network Selection Administrative file */etc/netconfig* is described on the **netconfig**(4bnu) manual page.

Taken together, these mechanisms allow a fine degree of control over network selection: a user can specify a preferred transport, and if it is reasonable, applications will use it. In cases where the specified transport is inappropriate (as, for example, when a remote server does not support a specified transport) the application should automatically try others with the right characteristics.

Name-to-address translation

Each transport has an associated set of routines that convert between universal network addresses (string representations of transport addresses) and their local address representation. These universal addresses are passed around within the RPC system (for example, between **rpcbind** and a client). When any programming interface to the transport layer is made, a transport-specific name-to-address translation routine is called to convert the universal address into local form. Each transport has associated with it a run-time loadable library that contains the name-to-address translation routines. The main translation routines are:

netdir_getbyname:
: Translates from host/service pairs and a **netconfig** structure (for example, **server1**, **rpcbind**) to a set of **netbuf** addresses. **netbuf**'s are X/Open Transport Interface (XTI) structures that are used at run-time to contain transport-specific addresses.

netdir_getbyaddr:
: Translates from **netbuf** addresses and a **netconfig** structure into host/service pairs.

uaddr2taddr:
: Translates from universal addresses and a **netconfig** structure to **netbuf** addresses.

taddr2uaddr:
: Translates from **netbuf** addresses and a **netconfig** structure to universal addresses.

For more details on these routines, see the **netdir**(3N) manual page.

The rpcbind facility

Client programs need a way to find server programs; that is, they need a way to look up the addresses of server programs. Network transport services do not themselves provide such a service; they merely provide process-to-process message transfer across a network. A message is sent to a transport-specific network address. A network address is a logical communications channel; by waiting on a network address, a process receives messages from the network.

RPC, being transport independent, makes no assumptions about the structure of a network address. It deals with universal addresses, specified only as null-terminated strings of characters. RPC translates universal addresses into local transport addresses by using routines specific to each transport provider. For more details on these routines, see the **netdir**(3N) manual page.

Operating systems provide (differing) mechanisms by which a process can wait on a network address, that is, synchronize its activity with arriving messages. Thus, messages are not sent across networks to receiving processes, but rather to the transport address at which receiving processes pick them up. Transport addresses are valuable because they allow message receivers to be specified in a way that is independent of the conventions of the receiving operating system. The **rpcbind** protocol defines a network service that provides a standard way for clients to look up the transport address of any remote program supported by a server. Because the **rpcbind** protocol can be implemented for any transport, it provides a single solution to a general problem that works for all clients, all servers and all networks.

Address registration

Because **rpcbind** is responsible for mapping network services to their addresses, its address must be well known. The name-to-address translation routines for any particular transport should know and reserve a particular address for **rpcbind**.

In the Internet domain, this problem is solved by always assigning **rpcbind** the port number 111. Unfortunately, this simple solution is not acceptable on all transports.

rpcbind begins each session by registering its location on each of the transports supported by the host. **rpcbind** is the only network service that must have such a well-known address. The address must be well-known for a given transport because **rpcbind** is responsible for registering the addresses of other network services and making those addresses available to network clients. Thus, services make their addresses available to clients by registering their addresses with their host's **rpcbind** daemon. Thereafter, the addresses of the services are available to **rpcinfo**(1tcp) and to programs using library routines specified in **rpcbind** (3rpc).

RPC-based servers typically get mapped to network addresses at run time, and then they register with **rpcbind**, and neither they nor their clients can make any assumptions about what those network addresses will be.

rpcbind is started by the system or RPC administrator. Both server programs and client programs call **rpcbind**.

NOTE: Although client and server programs and client and server machines are usually distinct, they need not be. A server program can also be a client program, as when an NFS® server calls an **rpcbind** server. Likewise, when a client program directs a "remote" procedure call to its own machine, the machine acts as both client and server.

As part of its initialization, a server program calls its host's **rpcbind** daemon to register itself in the host's registered-address map. Whereas server programs call **rpcbind** to update address maps, clients call them to query those maps. To find a remote program's address, a client sends an RPC call message to a server machine's **rpcbind** daemon; if the remote program is on the server, the daemon returns the relevant address in an RPC reply message. The client program can then send RPC call messages to that address.

The **rpcbind** protocol (for details, see ``rpcbind protocol'') provides a procedure, **RPCBPROC_CALLIT**, with which **rpcbind** can assist a client in making a remote procedure call. A client program passes the target procedure's program number, version number, procedure number (for a discussion of these numbers, see ``Remote Procedure Call programming'') and arguments in an RPC call message. **rpcbind** then looks up the

target procedure's address in the address map and sends an RPC call message, including the arguments received from the client, to the target procedure.

When the target procedure returns results, **RPCBPROC_CALLIT** passes them on to the client program. It also returns the target procedure's address so the client can later call it directly.

The RPC library provides an interface to all **rpcbind** procedures. Some of the RPC library procedures also call **rpcbind** automatically for client and server programs.

The rpcinfo command

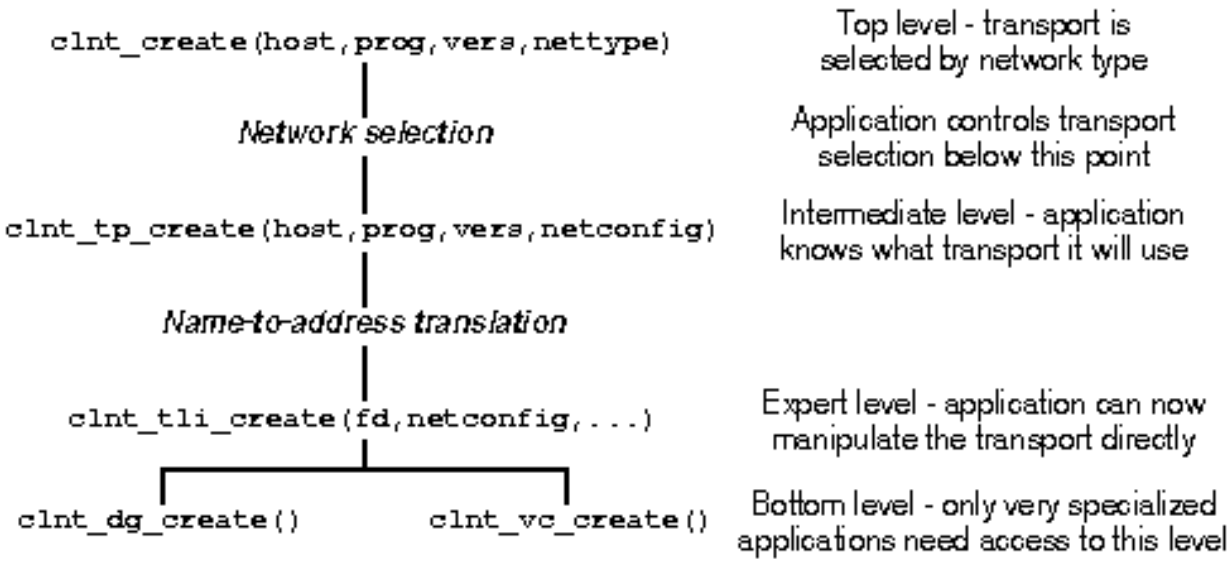
rpcinfo is a shell command that reports current RPC registration information known to **rpcbind** (and can be used, by administrators, to delete registrations). **rpcinfo** can be used to find all the RPC services registered on a specified host and to report their universal addresses and the transports for which they are registered. It can also be used to call (ping) a specific version of a specific program on a specific host using a TCP or UDP transport, and to report whether a response is received. For details, see **rpcinfo**(1tcp).

UnixWare 7 Release 7.1.4 - 27 April 2004

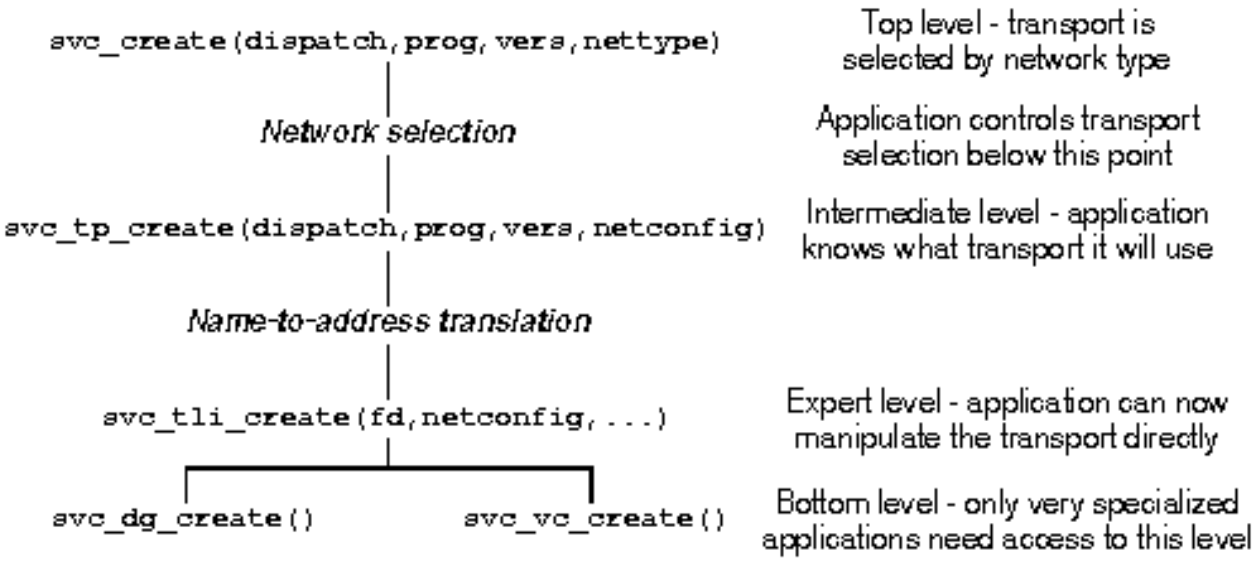
The lower RPC levels

There are various levels at which it is possible to interface to the RPC library services. These levels are described in detail in ``Remote Procedure Call programming''. Understanding the lower levels of RPC is helpful but not necessary if you plan to use **rpcgen** to generate your RPC applications. For usage of **rpcgen**, refer to ``Programming using the rpcgen command''.

See ``Client-side RPC lower levels'' for an illustration of client-side lower level interfaces that are available for transport-handle creation. See ``Server-side RPC lower levels'' for an illustration of transport-handle creation for an RPC server. Note the similarity of hierarchies on each side.



Client-side RPC lower levels



Server-side RPC lower levels

External Data Representation

RPC uses External Data Representation (XDR), a protocol for the machine-independent description and encoding of data. XDR is useful for transferring data between different computer architectures.

RPC can handle arbitrary data structures, regardless of different machines' byte orders or structure layout conventions, by always converting them to XDR representation 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***. For a detailed discussion of XDR, see ``XDR/RPC protocol specification''.

UnixWare 7 Release 7.1.4 - 27 April 2004

Programming using the rpcgen command

The details of programming applications to use Remote Procedure Calls can be overwhelming. Perhaps the most daunting task is writing the XDR routines that are necessary to convert procedure arguments and results into XDR format and back again.

The **rpcgen**(1tcp) compiler helps you write RPC applications simply and directly. It does most of the dirty work, allowing you to debug the main features of your application, instead of requiring you to spend most of your time developing transport interface code.

rpcgen accepts as input a remote program interface definition written in the RPC Language, which is similar to C. **rpcgen** outputs C language code that is suitable for RPC programs. This output includes:

- stub versions of the client routines
- a server skeleton
- XDR filter routines for both parameters and results
- a header file that contains common definitions
- (optionally) dispatch tables that the server can use to check authorizations and then invoke service routines

rpcgen's output files can be compiled and linked in the usual way.

The client stubs interface with the RPC library and effectively hide the transport from their callers. The server skeleton similarly hides the transport from the server procedures that are to be invoked by remote clients.

The developer writes server procedures (in any language that observes system calling conventions) and links them with the server skeleton produced by **rpcgen** to get an executable server program. To use a remote program, the programmer writes an ordinary main program that makes local procedure calls to the client stubs produced by **rpcgen**. Linking this program with stubs produced by **rpcgen** 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 skeleton.

rpcgen reduces the development time that would otherwise be spent coding and debugging low-level routines,

at a small cost in efficiency and flexibility. For speed-critical applications, though, **rpcgen** allows programmers to mix low-level code with high-level code. Hand-written routines can be linked with the **rpcgen** output without any difficulty. Also, one may proceed by using **rpcgen** output as a starting point, and then rewriting it as necessary. (For a discussion of RPC programming without **rpcgen**, see ``Remote Procedure Call programming'').

rpcgen allows you to do such things as:

- convert an application to run over a network
- create XDR routines
- make use of supported preprocessing directives.

``Common RPC programming techniques'' suggests some coding and usage techniques for **rpcgen**.

``RPC language reference'' provides a complete description of the RPC programming language recognized by **rpcgen**.

Converting local procedures into remote procedures

Assume an application that runs on a single machine. Suppose we want to convert it to run over the network. Here we will show such a conversion by way of a simple example program that prints a message to the console. The source file for the original program might look like this:

```
   /* 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 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);
}
```

For local use on a single machine, this program could be compiled and executed as follows:

cc printmsg.c -o printmsg
printmsg "Hello there."

This should return the response ``Message delivered!".

If the **printmessage** function were turned into a remote procedure, it could be called from anywhere in the network. It is not difficult to make a procedure remote.

NOTE: In the context of RPC programming, it has become acceptable to use the term *procedure* to refer to a C-language *function*. The terms are used interchangeably in this guide.

In general, it is necessary to figure out what the types are for all procedure inputs and outputs. Here, the procedure **printmessage** 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**. The RPC language source code for such a specification might look like this:

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



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

```
} = 0x20000001;
```

Remote procedures are always declared as being part of remote programs. The above is actually a declaration for an entire remote program, one that contains the single procedure **PRINTMESSAGE**.

NOTE: In the context of RPC programming, the term ``remote program'' actually refers to a collection of (related) *procedures*.

In this example, the **PRINTMESSAGE** procedure is declared to be procedure **1**, in version **1** of the remote program whose number is **0x20000001**. (Refer to ``Program number assignment'' for guidance on choice of program numbers.)

By convention, all RPC services provide for a procedure **0**; a call to a remote program's procedure **0** should do nothing (a ``no-op'') except succeed. To **ping** means to call procedure **0** of a remote program. Pinging is used to verify the existence and accessibility of a remote program.

Using **rpcgen**, no null procedure (procedure **0**) need be written because **rpcgen** generates it automatically.

Notice that the program and procedure names are declared with all capital letters. This is not required, but is a good convention to follow.

Notice also that the argument type is **string** and not **char *** as it would be in C. This is because a **char *** in C is ambiguous. Programmers usually intend it to mean a null-terminated string of characters, but it could also represent a pointer to a single character or a pointer to an array of characters. In RPC language, a null-terminated string is unambiguously called a **string**.

There are two more things to write:

- the remote procedure itself
- the main client program that calls it

The following is one possible 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"              /* 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 that the declaration of the remote procedure ***printmessage_1*** differs from that of the local procedure ***printmessage*** in three ways:

- 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.
- It returns a pointer to an integer instead of an integer itself. This is also characteristic of remote procedures: they return pointers to their results.

NOTE: When **rpcgen** is used, it is essential to have ***result*** (in this example) declared as ``static".

In the code generated by **rpcgen**, the result address is converted to XDR format after the remote procedure returns. If the result were declared local to the remote procedure, references to its address would be invalid after the remote procedure returned. So the result must be declared ``static" when **rpcgen** is used.

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

The last thing to do is declare the main client program that will call the remote procedure. This is one possibility:

```
/*
 * rprintmsg.c: remote version of "printmsg.c"
 */
#include <stdio.h>
#include <rpc/rpc.h>     /* always needed  */
#include "msg.h"         /* 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.
         */
        cl = clnt_create(server, MESSAGEPROG, MESSAGEVERS, "visible");
        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);
```

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

There are four things to note here:

- 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. (The client handle can be created in other ways as well, see ``Remote Procedure Call programming'' for details.)

- The last parameter to **clnt_create** is ``visible", which specifies that any transport noted as visible in */etc/netconfig* can be used.

- The remote procedure **printmessage_1** is called exactly the same way as it is declared in **msg_proc.c** except for the inserted client handle as the second argument. It also returns a pointer to the result instead of the result itself.

- The remote procedure call can fail in two ways. The RPC mechanism itself can fail or, alternatively, there can be an error in the execution of the remote procedure. In the former case, the remote procedure (in this case **print_message_1)** returns with a **NULL**. In the later case, however, the details of error reporting are application dependent. Here, the error is being reported via ***result**.

This is how to put all the pieces together:

rpcgen msg.x
cc rprintmsg.c msg_clnt.c -o rprintmsg -lnsl
cc msg_proc.c msg_svc.c -o msg_server -lnsl

Two programs are compiled here: the client program **rprintmsg** and the server program **msg_server**. Before doing this, **rpcgen** was used to fill in the missing pieces.

This is what **rpcgen** (called without any flags) did with the input file *msg.x*:

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

2. It created the client ``stub" routines in the *msg_clnt.c* file. Here there is only one, the **printmessage_1** routine, that was called from the **rprintmsg** client program. If the name of an **rpcgen** input file is *FOO.x*, the client stubs output file is called *FOO_clnt.c*.

3. It created the server program in *msg_svc.c* that calls **printmessage_1** from *msg_proc.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*.

NOTE: If invoked with the **-T** argument, **rpcgen** creates an additional output file that contains index information used for the dispatching of service routines.

Once created, the server should be copied to a remote machine and run. (If the machines are homogeneous, the server can be copied as a binary. Otherwise, the source files will need to be copied to and compiled on the remote machine.) For this example, the remote machine is called ***remote*** and the local machine is called ***local***. The server is started from the shell on the remote system:

```
remote$ msg_server
```

NOTE: Server processes, like **msg_server**, created with **rpcgen** always run in the background. It is not necessary to follow the server's invocation with an ampersand (&). Servers generated by **rpcgen** can also be invoked with port monitors like **listen** and **inetd**, instead of from the command line.

Thereafter, a user on *local* can print a message on the console of system *remote* as follows:

rprintmsg remote "Hello there."

Using **rprintmsg**, a user can print a message on any system console (including the *local* console) if the server *msg_server* is running on the target system.

Generating XDR routines with rpcgen

The example in ``Converting local procedures into remote procedures'' illustrated the automatic generation of client and server RPC code. You can also use **rpcgen** to generate XDR routines, that is, the routines necessary to convert local data structures into XDR format and vice-versa.

This example presents a complete RPC service: a remote directory listing service, built using **rpcgen** not only to generate stub routines, but also to generate the XDR routines.

This 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;
} = 0x20000076;
```

NOTE: Types (like `readdir_res` in the example above) can be defined using the **struct**, **union** and **enum** keywords. These keywords should not be used in later declarations of variables of those types. For example, if you define a union *foo*, you should declare using only *foo* and not *union foo*.

rpcgen compiles RPC unions into C structures. It is an error to declare RPC unions using the **union** keyword.

Running **rpcgen** on *dir.x* creates four output files. First are the basic three described in ``Converting local procedures into remote procedures'': the header file, client stub routines and server skeleton.

The fourth contains the XDR routines necessary for converting instances of declared data types from host platform representation into XDR format, and vice-versa. These routines are output in the file *dir_xdr.c*.

For each data type used in the *.x* file, **rpcgen** assumes that the RPC/XDR library contains a routine whose name is the name of the datatype, prepended by *xdr_* (for example, **xdr_int**). If a data type is defined in the *.x* file, then **rpcgen** generates the required **xdr_** routine.

If there are no such data types definitions, in an RPC source file (for example, *msg.x*), then an *_xdr.c* file will not be generated.

An RPC programmer may write a *.x* source file that uses a data type not supported by the RPC/XDR library, and deliberately omit defining the type (in the *.x* file); if so, the programmer has to provide that **xdr_** routine. This is a way for programmers to provide their own customized **xdr_** routines. See ``Remote Procedure Call programming'' for more details on passing arbitrary data types.

This is the server-side implementation of the **READDIR** procedure:

```
/*
 * dir_proc.c: remote readdir implementation
 */
#include <rpc/rpc.h>         /* Always needed */
#include <dirent.h>
#include "dir.h"                    /* Created by rpcgen */



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



readdir_res *
readdir_1(dirname)
     nametype *dirname;
{
     DIR *dirp;
     struct dirent *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);
   }
```

This is the client side program to call the server:

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



   #include <stdio.h>
   #include <rpc/rpc.h> /* always need this */
   #include "dir.h"             /* 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 any visible transport when contacting the server.
         */



        cl = clnt_create(server, DIRPROG, DIRVERS, "visible");
        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);
    }
```

Again using the hypothetical systems named *local* and *remote*, the files can be compiled and run as follows on *remote*:

rpcgen dir.x
cc -c dir_xdr.c
cc rls.c dir_clnt.c dir_xdr.o -o rls -lnsl
cc dir_svc.c dir_proc.c dir_xdr.o -o dir_svc -lnsl
dir_svc

After installing an executable copy of **rls** on system *local*, a user on that system can list the contents of */usr/share/lib* on system *remote* by running the command **rls remote /usr/share/lib** on *local*. This produces output such as:

```
.
..
ascii
eqnchar
greek
kbd
marg8
tabclr
tabs
tabs4
local$
```

Using preprocessing directives

The **rpcgen** compiler supports C and other preprocessing features.

C-preprocessing is performed on **rpcgen** input files before they are compiled. All C-preprocessing directives are valid within a *.x* file. Five symbols may be defined by **rpcgen**, depending on the type of output file being generated. The symbols are:

Symbol	Usage
RPC_HDR	Header file output
RPC_XDR	XDR routine output
RPC_SVC	Server skeleton output
RPC_CLNT	Client stub output
RPC_TBL	Index table output

The **rpcgen** compiler provides an additional preprocessing feature: any line that begins with a percent sign (%) is passed directly into the output file, without any interpretation of the line.

NOTE: The **%** feature is not always useful, owing to a limitation: The **rpcgen** compiler may not place the lines where the programmer intended.

This is a simple example that illustrates **rpcgen** preprocessing features:

```
/*
 * time.x: Remote time protocol
 */
program TIMEPROG {
     version TIMEVERS {
                     unsigned int TIMEGET(void) = 1;
     } = 1;
} = 0x20000044;



#ifdef RPC_SVC
```

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

UnixWare 7 Release 7.1.4 - 27 April 2004

Common RPC programming techniques

This section suggests some coding and **rpcgen** usage techniques.

Topic	RPC capability
Network types	**rpcgen** can produce for specific transport types (or even specific transports)
Timeout changes	Client default timeout periods can be changed
Authentication	Clients may authenticate themselves to servers; interested servers can examine client authentication information
define statements	C-preprocessing symbols can be defined on **rpcgen** command lines
Broadcast calls	Servers need not send **NULL** replies to broadcast calls
Port monitor support	Port monitors can ``listen" for RPC servers
Dispatch tables	Programs can access server dispatch tables
Debugging	Clients and servers created with **rpcgen** can be linked and run on a single system for debugging purposes

Network types (transport selection)

The **rpcgen** compiler takes optional arguments that allow a programmer to specify a desired network type or even a specific network identifier. (For details about network selection, see ``Remote Procedure Call programming".)

NOTE: In the context of RPC programming, the term ***network*** is frequently used (as here) as a synonym for ***transport*** or ***transport type***.

The **-s** flag creates a server that responds to requests on all transports of a specified type. For example, the invocation

rpcgen -s datagram_n prot.x

writes a server to standard output that responds to any of the connectionless transports specified in the **NETPATH** environment variable (or in */etc/netconfig*, if **NETPATH** is not defined or does not specify any connectionless transports).

Similarly, the **-n** flag creates a server that responds only to requests from the transport specified by a single network identifier that must match a network-id in the */etc/netconfig* file.

NOTE: Be careful using servers created by **rpcgen** with the **-n** flag. Because network identifiers are host-specific, the server produced may not run on other hosts in the expected way.

Timeout changes

After sending a request to the server, a client program waits for a default amount of time (25 seconds) to receive a reply. This timeout may be changed using the **clnt_control** routine. (See **rpc**(3rpc).)

NOTE: When considering timeout periods, be sure to allow for the minimal amount of time required for ``round trip'' communications over the network.

This is a small code fragment to illustrate the use of **clnt_control**:

```
struct timeval tv;
CLIENT *cl;



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

Client authentication

The client create routines do not, by default, have any facilities for client authentication, but the client may sometimes want (or be required) to authenticate itself to the server. Doing so is trivial, and looks about like this:

NOTE: The following example illustrates one of the least secure authentication methods in common use. See ``Remote Procedure Call programming'' for information on the more secure DES authentication technique.

```
CLIENT *cl;



cl = client_create("somehost", SOMEPROG, SOMEVERS, "visible");
if (cl != NULL) {
     /* To set AUTH_SYS style authentication */
     cl->cl_auth = authsys_createdefault();
}
```

Servers that want to know more about an RPC call can check authentication information. For example, getting authentication information is important to servers that want to achieve some level of security. This extra information is actually supplied to the server as a second argument. (For details, see the structure of **svc_req**, in ``Authentication''.

This is an example of a remote procedure whose server checks client authentication information. This is a rewrite of **printmessage_1** which is developed in ``Programming using the rpcgen command'' The rewritten procedure will only allow root users to print a message to the console:

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

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



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

UnixWare 7 Release 7.1.4 - 27 April 2004

rpcgen command-line define statements

The **rpcgen** compiler provides a means for defining C-preprocessing symbols and assigning values to them from the command line. Command-line define statements can, for example, be used to compile conditional debugging code that is compiled only when the **DEBUG** symbol is defined. For example:

rpcgen -DDEBUG proto.x

UnixWare 7 Release 7.1.4 - 27 April 2004

Server response to broadcast calls

When a procedure is known to have been called via broadcast RPC, and the called procedure determines that it cannot provide the client with a useful response, it is usually best for the server to send no reply back to the client. This reduces network traffic.

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

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

```
void *
reply_if_nfsserver_1()
{
     char notnull;   /* just here so we can use its address */



     if (access("/etc/exports", F_OK) < 0) {
             return (NULL);  /* prevent RPC from replying */
     }
     /*
      * assign notnull a non-null value so RPC will send out a reply
      */
     return ((void *)&notnull);
}
```

NOTE: If a procedure returns type **void ***, it must return a non-NULL pointer if it wants RPC to send a reply.

UnixWare 7 Release 7.1.4 - 27 April 2004

Port monitor support

Port monitors such as **inetd**(1Mtcp) and **listen**(1M) can monitor network addresses for specified RPC services. Whenever a request comes in for a particular service, the port monitor spawns a server process to provide for it. After the call has been serviced, the server can exit. This has the advantage of conserving system resources: fewer blocked processes waiting for work.

It may be useful for services to wait for a specified interval after satisfying a service request, on the chance that another request will follow. If there is no call within the specified time, the server will exit and some port monitors, like **inetd**, will continue to monitor for the server. If a later request for the service occurs, the port monitor will give the request to a waiting server process (if any), rather than spawning a new process.

NOTE: Some port monitors, like **listen**, treat requests for a particular service in one of two ways:

- Spawn a new server process for each request for the service. A service to be spawned should exit immediately on completion.
- Pass each request for the service through a named STREAM to the server process (which must already be running). A service to which requests will be passed must poll file descriptor 0 waiting for requests.

By default, services created using **rpcgen** wait for 120 seconds after servicing a request before exiting. The programmer can, however, change that interval with the **-K** flag.

rpcgen -K 20 proto.x

The server will wait only for 20 seconds before exiting. To create a server that exits immediately, **-K 0** can be used. To create a server that never exits (a standing server), the appropriate argument is **-K -1**.

All servers generated by **rpcgen** assume the following support from port monitors:

- the name of the transport provider is passed through the environment variable **NLS_PROVIDER**
- the connection is passed on an open XTI file descriptor 0

See ``Using port monitors'' for a further discussion of port monitors.

UnixWare 7 Release 7.1.4 - 27 April 2004

Dispatch tables

It is sometimes useful for programs to have access to dispatch tables used by the RPC package. For example, the server dispatch routine may need to check authorization and then invoke the service routine; or a client library may want to deal with the details of storage management and XDR data conversion.

When invoked with the **-T** option, **rpcgen** generates RPC dispatch tables for each program defined in the protocol description file, ***proto***.*x*, in the file ***proto***_*tbl*.*i*. For sample protocol description file, *dir*.*x*, given in ``Generating XDR routines with rpcgen", a dispatch table file created by **rpcgen** would be called *dir*_*tbl*.*i*. The suffix **.i** stands for ``index."

Each entry in the dispatch table is a **struct rpcgen_table**, defined in the header file *proto.h* as follows:

```
struct rpcgen_table {
    char        *(*proc)();
    xdrproc_t   xdr_arg;
    unsigned    len_arg;
    xdrproc_t   xdr_res;
    unsigned    len_res;
};
```

Here:

proc
: is a pointer to the service routine

xdr_arg
: is a pointer to the input (argument) **xdr_** routine

len_arg
: is the length in bytes of the input argument

xdr_res
: is a pointer to the output (result) **xdr_** routine

len_res
: is the length in bytes of the output result

The table, named *dirprog_1_table* for the *dir*.*x* example, is indexed by procedure number. The variable **dirprog_1_nproc** contains the number of entries in the table.

An example of how to locate a procedure in the dispatch tables is shown by the routine **find_proc()**:

```
struct rpcgen_table *
find_proc(proc)
     long    proc;
{
     if (proc >= dirprog_1_nproc)
             /* error */
     else
             return (&dirprog_1_table[proc]);
}
```

Each entry in the dispatch table contains a pointer to the corresponding service routine. However, that service routine is usually not defined in the client code. To avoid generating unresolved external references, and to require only one source file for the dispatch table, the **rpcgen** service routine initializer is **RPCGEN_ACTION (proc_ver)**.

This way, the same dispatch table can be included in both the client and the server. Use the following definition when compiling the client:

```
#define RPCGEN_ACTION(routine)       0
```

and use this definition when compiling the server:

```
#define RPCGEN_ACTION(routine)       routine
```

Debugging with rpcgen

When programming with **rpcgen**, the client program and the server procedure can be tested together as a single program by linking them with each other rather than with the client and server stubs. To do this, calls to RPC library routines (for example, **clnt_create**), have to be commented out, and client-side routines have to call server routines directly. The procedure calls will be executed as ordinary local procedure calls and the program can be debugged with a local debugger. After the program is working, the client program can be linked to the **rpcgen**-created client stubs and the server procedures can be linked to the **rpcgen**-created server skeleton.

UnixWare 7 Release 7.1.4 - 27 April 2004

RPC language reference

RPC language is an extension of the XDR language. The sole extension is the addition of the ``program" and ``version" types.

For a complete description of the XDR language syntax and the RPC extensions to the XDR language, see ``XDR/RPC protocol specification".

RPC language is similar to C language. We describe here the syntax of the RPC language, showing a few examples along the way. We also show how the various RPC and XDR type definitions get compiled into C type definitions in the output header file.

UnixWare 7 Release 7.1.4 - 27 April 2004

Definitions

An RPC language file consists of a series of definitions.

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

It recognizes six types of definitions.

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

Enumerations

RPC/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
```

This is a short example of an RCP/XDR **enum**:

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

This gets compiled to the C **enum**:

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

Constants

RPC/XDR symbolic constants may be used wherever 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;
```

This is converted to:

```
#define DOZEN 12
```

Typedefs

RPC/XDR typedefs have the same syntax as C typedefs.

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

This example defines an **fname_type** used for declaring file name strings that have a maximum length of 255 characters.

```
typedef string fname_type<255>;
```

This is converted to:

```
typedef char *fname_type;
```

UnixWare 7 Release 7.1.4 - 27 April 2004

Declarations

In RPC/XDR, there are four kinds of declarations.

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

Simple declarations are just like simple C declarations.

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

For example:

```
colortype color;
```

is converted to:

```
colortype color;
```

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

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

For example:

```
colortype palette[8];
```

is converted to:

```
colortype palette[8];
```

Variable-length array declarations have no explicit syntax in C. The RPC/XDR does have a syntax; it uses 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 compiled into **struct** declarations. 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 RPC/XDR variable.

Pointer declarations are made in RPC/XDR exactly as they are in C. Address pointers cannot really be sent over the network, but RPC/XDR pointers are useful 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
```

For example:

```
listitem *next;
```

is converted to:

```
listitem *next;
```

Structures

An RPC/XDR **struct** is declared almost exactly like its C counterpart:

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



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

As an example, this is an RPC/XDR structure for a two-dimensional coordinate, and the C structure that it gets compiled into in the output header file.

```
struct coord {
     int x;
     int y;
};
```

The output is identical to the input, except for the added `typedef` at the end of the output:

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

This allows one to use `coord` instead of `struct coord` when declaring items.

Unions

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

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



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

This 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, do not return anything.

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

This 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.

Programs

RPC/XDR 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:

```
/*
 * time.x: Get or set the time.  Time is represented as seconds
 * since 0:00, January 1, 1970.
 */
program TIMEPROG {
     version TIMEVERS {
```

```
            unsigned int TIMEGET(void) = 1;
            void TIMESET(unsigned) = 2;
        } = 1;
} = 0x20000044;
```

This file compiles into these definitions in the output header file:

```
#define TIMEPROG 0x20000044
#define TIMEVERS 1
#define TIMEGET 1
#define TIMESET 2
```

UnixWare 7 Release 7.1.4 - 27 April 2004

Special cases

There are a few exceptions to the rules described above.

Booleans

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

For example:

```
bool married;
```

is converted to:

```
bool_t married;
```

Strings

C has no built-in string type, but instead uses the null-terminated **char *** convention. In RPC/XDR language, strings are declared using the **string** keyword, and compiled into type **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.

For example:

```
string name<32>;
string longname<>;
```

are converted to:

```
char *name;
char *longname;
```

Opaque data

describes untyped data, that is, just sequences of arbitrary bytes. It may be declared either as a fixed or variable length array.

For example:

```
opaque diskblock[512];


opaque filedata<1024>;
```

are converted as:

```
char diskblock[512];


struct {
     u_int filedata_len;
     char *filedata_val;
     } filedata;
```

Voids

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

Remote Procedure Call programming

The RPC package provides a multi-level application programming interface for development of network applications using remote procedure calls.

At the ***simplified interface*** (the highest level), the package provides great transparency, but offers only limited control over the underlying communications mechanisms. Program development at the simplified interface can be rapid, and is directly supported by the **rpcgen** compiler -- a C-language code generator that supports remote procedure call program development.

NOTE: ``Generating XDR routines with rpcgen'' 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. For most applications, **rpcgen** and its facilities are fully adequate and the detailed information in this section is not required.

Interfaces to lower levels of the RPC package provide increasing control over remote procedure call communications. Programs that exercise this control pay for the power in terms of greater complexity of code. Effective programming at the lower levels requires knowledge of computer network fundamentals.

In order of increasing control and complexity, these levels are called the ``Top level'', ``Intermediate level'', ``Expert level'' and ``Bottom level''.

This section is intended for programmers who wish to write network applications using remote procedure calls, and who want to use or understand the RPC mechanisms usually hidden by the **rpcgen**(1tcp) protocol compiler.

UnixWare 7 Release 7.1.4 - 27 April 2004

Levels of the RPC package

The RPC interface can be seen as being divided into several distinct levels. The highest level is general, and provides for no fine control of any kind. The lower levels (four can be usefully distinguished) are available for use as necessary, and provide increasingly detailed levels of control. Programmers should only go down to the level necessary for the control needed.

NOTE: For a complete specification of the routines in the RPC library, see **rpc**(3rpc) and related manual pages.

In the ``simplified interface", you do not need to consider the characteristics of the underlying transport, operating system, or other low-level implementation mechanisms. Programmers simply make remote procedure calls to routines on other machines, and need specify only the ***type*** of transport that they wish to use. The selling point here is simplicity. It is this level that allows RPC to pass the ``hello world" test -- that simple things should be simple. The routines at this level are used for most applications.

Included in the simplified interface are only three basic RPC routines:

rpc_reg
: **rpc_reg** registers a routine as an RPC routine and obtains a unique, system-wide procedure-identification number for it.

rpc_call
: Given such a unique, system-wide procedure-identification number, **rpc_call** uses it to make a remote call to that routine on a specified host.

rpc_broadcast
: Like **rpc_call**, except that it broadcasts its call message across all transports of the specified type.

At the ``top level", the interface is still simple, but the programmer does have to create client and server handles before making a call. Like the routines in the simplified interface, the routines here require a **nettype** argument that specifies a general class (type) of transports.

The top level essentially consists of two routines:

clnt_create
: The generic client creation. The programmer tells **clnt_create** where the server is located and the type

of transport to use to get to it.

svc_create
Creates server handles for all the transports of the specified ***nettype***. The programmer tells **svc_create** which dispatch function should be used.

The simplified interface and the top level of RPC, while simple, do not allow the choice of a specific transport (but see the following discussion of **NETPATH**). At these levels, all routines take a **nettype** argument, which serves to define the class of transport to be used. On the client side, programs do network selection, and hence may be slightly inefficient depending on the **nettype**. On the server side, programs may have to listen on many transports, and hence may waste system resources.

In both of these cases, however, efficiency can be improved by judicious assignment to the **NETPATH** environment variable. If the programmer wishes the application to run on all transports, this is the interface that should be used.

The ``intermediate interface'' of RPC, and the two interfaces below it, allow many details to be controlled by the programmer, and for that reason their use is necessary for special applications. Programs written at these lower levels are more complicated, but also more efficient.

The intermediate differs from the two levels above it in that it allows the programmer to specify directly the transport to be used. It consists of two routines:

clnt_tp_create
Creates a client handle for a specified transport.

svc_tp_create
Likewise, **svc_tp_create** creates a server handle for a specified transport.

The ``expert level'' consists of a larger set of routines with which the programmer can specify more parameters, but those parameters are still all directly transport related. It includes the following routines:

clnt_tli_create
Creates a client handle for a specified transport, allowing fine control of the client characteristics.

svc_tli_create
Creates a server handle for a specified transport, allowing fine control of the server characteristics.

rpcb_set,
Provides a programmatic interface to **rpcbind**, one that establishes a mapping between an RPC service and a network address.

rpcb_unset
Destroys a mapping of the type established by **rpcb_set**.

rpcb_getaddr
Provides a programmatic interface to **rpcbind**, one that returns the transport address of specified RPC service.

svc_reg
Associates a given program and version number pair with a given dispatch routine.

svc_unreg
Destroys an association of the type established by **svc_reg**.

The ``bottom level'' consists of routines called when the programmer requires full control, even down to the smallest details of transport options. It consists of the following routines:

clnt_dg_create
Creates an RPC client for the specified remote program, using a connectionless transport.

svc_dg_create
Creates an RPC server handle, using a connectionless transport.

clnt_vc_create
Creates an RPC client for the specified remote program, using a connection-oriented transport.

svc_vc_create
Creates an RPC server handle, using a connection-oriented transport.

The simplified interface to RPC

The easiest interface to RPC does not require the programmer to use the interface at all. ``RPC library-based network services'' describes using functions that hide all details of the RPC package.

Some RPC services are not available as C functions, but are available as RPC programs. ``Remote Procedure Call and registration'' shows how easy it is to use these services, and how easy it is to create new services that are equally simple to use.

Data types passed to and received from remote procedures can be any of a set of predefined types, or can be programmer-defined types. ``Passing arbitrary data types'' explains how such types are declared and used.

RPC library-based network services

Imagine writing a program that needs to know how many users are logged into a remote machine. This can be done by calling an RPC library routine, **rusers**, as illustrated below:

```
#include <stdio.h>



/*
 * a program that calls rusers()
 */



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



     if (argc != 2) {
             fprintf(stderr, "usage: %s hostname\n", argv[0]);
             exit(1);
     }
     if ((num = rusers(argv[1])) < 0) {
```

```
                fprintf(stderr, "error: rusers\n");
                exit(1);
        }
        printf("%d users on %s\n", num, argv[1]);
        exit(0);
}
```

NOTE: For **rusers** to work, the **rusers** daemon must be running on the remote host.

RPC library routines such as **rusers** are in the RPC services library **librpcsvc.a**. Thus, the program above should be compiled with

cc program.c -lrpcsvc -lnsl

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

Routine	Description
rusers	Return information about users on remote machine
rwall	Write to specified remote machines
spray	Spray packets to a specific machine

Remote Procedure Call and registration

The simplest interface to the RPC functions is based on the routines **rpc_call**, **rpc_reg**, and **rpc_broadcast**. These functions provide direct access to the RPC facilities, and are appropriate for programs that do not require fine levels of control.

Using the simplified interface, the number of remote users can be gotten as follows:

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



/*
 * a program that calls the RUSERSPROG RPC program
 */
```

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



     if (argc != 2) {
            fprintf(stderr, "usage: rusers hostname\n");
            exit();
     }
     if (clnt_stat = rpc_call(argv[1],
       RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
       xdr_void, 0, xdr_u_long, &nusers, "visible") != 0) {
            clnt_perrno(clnt_stat);
            exit(1);
     }
     printf("%d users on %s\n", nusers, argv[1]);
     exit(0);
}
```

The rpc_call routine

The simplest way of making remote procedure calls is with the RPC library routine **rpc_call**. It has nine 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 remote procedure to be called.
- The fifth and sixth parameters are an XDR filter for encoding and an argument that has to be passed to the remote procedure.
- The next 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.
- Finally, there is the nettype specifier.

Multiple arguments and results are handled by embedding them in structures. If **rpc_call** completes successfully, it returns zero; otherwise, it returns a nonzero value. The return codes (of type `enum`

`clnt_stat`, cast to an `int` in the previous example) are found in <**rpc/clnt.h**>.

Because data types may be represented differently on different machines, **rpc_call** needs both the type of, and a pointer to, the RPC argument (similarly for the result). For **RUSERSPROC_NUM**, the return value is an unsigned long, so **rpc_call** 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 **rpc_call** is `xdr_void`.

If **rpc_call** gets no answer within a certain time period, it returns with an error code. In the example, it tries all the transports listed in */etc/netconfig* that are flagged as visible. Adjusting the number of retries requires use of the lower levels of the RPC library, discussed later in this section. The remote server procedure corresponding to the above might look like this:

```
char *
rusers()
{
     static unsigned long nusers;



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

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

The rpc_reg routine

Normally, a server registers all the RPC calls it plans to handle, and then goes into an infinite loop waiting to service requests. If **rpcgen** is used to provide this functionality, it will generate much code, including a server dispatch function and support for port monitors. But programmers can also write servers themselves using **rpc_reg**, and it is appropriate that they do so if they have simple applications, like the one shown as an example here. In this example, there is only a single procedure to register, so the main body of the server would look like this:

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

```
char *rusers();


main ()
{


     if (rpc_reg(RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
       rusers, xdr_void, xdr_u_long, "visible") == -1) {
            fprintf(stderr, "Couldn't Register\n");
            exit(1);
     }
     svc_run();              /* Never returns */
     fprintf(stderr, "Error: svc_run returned!\n");
     exit(1);
}
```

The **rpc_svc_reg**(3rpc) routine registers a C procedure as corresponding to a given RPC procedure number. The registration is done for each of the transports of the specified type, or if the type parameter is **NULL**, for all the transports named in **NETPATH**. The first three parameters, **RUSERPROG**, **RUSERSVERS**, and **RUSERSPROC_NUM** are the program, version, and procedure numbers of the remote procedure to be registered; **rusers** is the name of the local procedure that implements the remote procedure; and `xdr_void` and `xdr_u_long` name the XDR filters for the remote procedure's arguments and results, respectively. (Multiple arguments or multiple results are passed as structures.) The last parameter specifies the desired nettype.

When using **rpc_reg**, programmers are not required to write their own dispatch routines. Also, the dispatcher in **rpc_reg** takes care of decoding remote procedure arguments and encoding results, using the XDR filters specified when the remote procedure was registered.

After registering the local procedure, the server program's main procedure calls **svc_run**, the RPC library's remote procedure dispatcher, which is described on the **rpc_svc_reg**(3rpc) manual page. It is this function that calls the remote procedures in response to RPC call messages.

NOTE: The **svc_run** routine is used at all levels of RPC programming. Strictly speaking, it does not ``belong'' to this or to any other level.

CAUTION: The **svc_run** routine is not thread-safe and should not be called from multiple threads of execution. See ``Writing multithreaded RPC procedures'' and ``Multithreaded network programming'' for more information about using RPC with the Threads Library.

Passing arbitrary data types

In the previous example, the RPC call returned 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 standard transfer syntax called External Data Representation (XDR) before sending them over the transport. 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 **rpc_call** and **rpc_reg** can name an XDR primitive procedure, like **xdr_u_long** in the previous example, or a programmer supplied procedure (that may take a maximum of two parameters). XDR has these ``built-in'' primitive type routines: **xdr_bool**, **xdr_char**, **xdr_enum**, **xdr_int**, **xdr_long**, **xdr_short**, **xdr_u_char**, **xdr_u_int**, **xdr_u_long**, **xdr_u_short**, and **xdr_wrapstring**.

NOTE: The routine **xdr_string** exists, but takes more than two parameters. It cannot, therefore, be used with **rpc_call** and **rpc_reg**, which only pass two parameters to their XDR routines. **xdr_wrapstring** has only two parameters, and is thus acceptable. It, in turn, calls **xdr_string**.

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

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

then **rpc_call** would be called as:

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

where **xdr_simple** is written as:

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



bool_t
xdr_simple(xdrsp, simplep)
```

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

An XDR routine returns nonzero (true in the C sense) if it completes successfully, and zero otherwise. A complete description of XDR is provided in the ``XDR/RPC protocol specification''. Note that the above routine could have been generated automatically by using the **rpcgen** compiler.

In addition to the built-in primitives, there are also some prefabricated building blocks: **xdr_array**, **xdr_bytes**, **xdr_opaque**, **xdr_pointer**, **xdr_reference**, **xdr_string**, **xdr_union**, and **xdr_vector**.

To send a variable array of integers, the array might be packaged as a structure like this:

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

and sent by an RPC call such as:

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

with **xdr_varintarr** defined as:

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

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

If the size of the array is known in advance, one can use **xdr_vector**, which serializes fixed-length arrays.

```
int intarr[SIZE];



bool_t
xdr_intarr(xdrsp, intarr)
     XDR *xdrsp;
     int intarr[];
{
     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 the **xdr_string** routine, which is the same as **xdr_bytes** without the length parameter. On serializing it gets the string length from **strlen**, and on deserializing it creates a null-terminated string.

This 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;



bool_t
xdr_finalexample(xdrsp, finalp)
     XDR *xdrsp;
     struct finalexample *finalp;
{
     if (!xdr_string(xdrsp, &finalp->string, MAXSTRLEN))
             return (FALSE);
     if (!xdr_reference(xdrsp, &finalp->simplep,
       sizeof(struct simple), xdr_simple))
             return (FALSE);
     return (TRUE);
}
```

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

UnixWare 7 Release 7.1.4 - 27 April 2004

The lower levels of RPC

In the examples given for programming at the simplied interface, RPC takes care of almost as many details as would the **rpcgen** compiler. RPC does so by choosing defaults for almost everything, including the transport protocol.

This section shows how to control these details by using lower levels of the RPC library. The reader is assumed to be familiar with the Transport Level Interface (XTI).

There are several reasons for using lower levels of RPC:

- A program may need to directly control the selection of the transport protocol, which at the simplified interface level, can be done only by use of the **NETPATH** variable.
- A program may need to allocate and free memory while serializing or deserializing with XDR routines. There are no facilities for doing so available at the higher level. (For details, see ``An example of performing memory allocation with XDR'').

The following sections illustrate programming at the lower levels of RPC.

``The top level'' describes RPC interfaces that allow for control of transport selection by type.

``The intermediate level'' section describes those interfaces that allow a programmer to choose a specific transport.

``The expert level'' section describes routines that:

- allow program control of client and server characteristics
- provide an interface to **rpcbind**

Finally, the section on ``The bottom level'' describes routines that control most details of transport options.

For detailed descriptions of RPC routines, see **rpc**(3rpc).

The top level

At the top level, the application can specify the type of transport that it wants to use, but not an individual transport. This level differs from the simplified interface to RPC in that the application is responsible for creating its own transport handles, on both the client and server sides.

Top level: client side

Assume we have the following header file:

```
/*
 * time_prot.h
 */



#include <rpc/types.h>



struct timev {
     int second;
     int minute;
     int hour;
};
typedef struct timev timev;



bool_t xdr_timev(xdrsp, resp)
     XDR *xdrsp;
     struct timev *resp;
{
     if (!xdr_int(xdrsp, &resp->second))
             return (FALSE);
     if (!xdr_int(xdrsp, &resp->minute))
             return (FALSE);
     if (!xdr_int(xdrsp, &resp->hour))
             return (FALSE);
     return (TRUE);
```

```
}



#define TIME_PROG ((u_long)76)
#define TIME_VERS ((u_long)1)
#define TIME_GET ((u_long)1)
```

The following code implements the client side of a trivial date service, written at the top level:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "time_prot.h"



#define TOTAL (30)



/*
 *  Caller of trivial date service
 *  usage: calltime hostname
 */
main(argc,argv)
     int argc;
     char *argv[];
{
     struct timeval timeout;
     CLIENT *client;
     enum clnt_stat stat;
     struct timev timev;
     char *nettype;



     if (argc != 2 && argc != 3) {
             fprintf(stderr,"usage: %s host [nettype]\n",argv[0]);
     }



     if (argc == 2)
             nettype = "netpath";    /* Default */
     else
             nettype = argv[2];
```

```
        client = clnt_create(argv[1], TIME_PROG, TIME_VERS,
          nettype);
        if (client == NULL) {
                clnt_pcreateerror("Couldn't create client");
                exit(1);
        }
        timeout.tv_sec = TOTAL;
        timeout.tv_usec = 0;
        stat = clnt_call(client, TIME_GET, xdr_void, NULL,
          xdr_timev, &timev, timeout);
        if (stat != RPC_SUCCESS) {
                clnt_perror(client, "Call failed");
                exit(1);
        }
        printf("%s: %02d:%02d:%02d GMT\n", nettype, timev.hour,
          timev.minute, timev.second);
        exit(0);
}
```

Note that, when this program is run, if **nettype** is not given on the command line, the code assigns it to point to the string ``"netpath"''. Whenever the routines in the RPC libraries encounter this string, they consult the **NETPATH** environment variable for the user's list of acceptable network identifiers.

If the client handle cannot be created, the reason for the failure can be printed using **clnt_pcreateerror**, or the error status can be obtained via the global variable **rpc_createerr**.

NOTE: In applications linked with the Threads Library, distinct instances of **rpc_createerr** and **errno** will be supported for each thread. The value of **rpc_createerr** for the current thread will be returned by the new function **get_rpc_createerr**(3rpc). You may set the value of **rpc_createerr** by calling the function **set_rpc_createerr**, which is also described on the **get_rpc_createerr**(3rpc) manual page. The symbol **rpc_createerr** will continue to be available in read-only form, but will generate a compile-time error if you try to assign it a value directly. You may continue to assign values to **errno** directly, however. See ``Multithreaded network programming'' for more information about using RPC with the Threads Library.

After the client handle is created, **clnt_call** is used to make the remote call. It takes as arguments the remote procedure number, an XDR filter for the input argument and the argument pointer, an XDR filter for the result and the result pointer, and the time-out period of the call. Normally, this last should not be **0**. In this particular example there are no arguments, and thus **xdr_void** has been specified.

Top level: server side

This is the code for the time server:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "time_prot.h"



static void time_prog();



main(argc,argv)
     int argc;
     char *argv[];
{
     int transpnum;
     char *nettype;



     if (argc == 2)
             nettype = argv[1];
     else
             nettype = "netpath";    /* Default */
     transpnum = svc_create(time_prog, TIME_PROG, TIME_VERS,
       nettype);
     if (transpnum == 0) {
             fprintf(stderr,"%s: cannot create %s service.\n",
               argv[0], nettype);
             exit(1);
     }
     svc_run();
}



/*
 * The server dispatch function
 */
static void
time_prog(rqstp, transp)
     struct svc_req *rqstp;
     SVCXPRT *transp;
{
     struct timev rslt;
     long thetime;
```

```
        switch (rqstp->rq_proc) {
        case NULLPROC:
                svc_sendreply(transp, xdr_void, NULL);
                return;



        case TIME_GET:
                break;



        default:
                svcerr_noproc(transp);
                return;
        }



        thetime = time(0);
        rslt.second = thetime % 60;
        thetime /= 60;
        rslt.minute = thetime % 60;
        thetime /= 60;
        rslt.hour = thetime % 24;
        if (! svc_sendreply(transp, xdr_timev, &rslt)) {
                svcerr_systemerr(transp);
        }
    }
```

svc_create returns the number of transports on which it could create server handles. **time_prog** is the dispatch function called by **svc_run** whenever there is a request for its given program and version number.

Here the remote procedure takes no arguments. Had arguments been required,

```
svc_getargs(transport, XDR_filter, argument_pointer)
```

could have been used to deserialize (XDR decode) the arguments. In such cases, **svc_freeargs** should be used to free up the arguments after the actual call has been made. The server reply results are sent back to the client using **svc_sendreply**.

It is recommended that **rpcgen** be used to generate the dispatch function which can later be customized.

NOTE: When **rpcgen** is used to generate the dispatch function, **svc_sendreply** is called only after the actual procedure has returned, and hence it is essential to have **rslt** (in this example) declared as ``static'' within that actual procedure.

In this example, **rslt** is not declared as static because **svc_sendreply** is called from within the dispatch function.

The intermediate level

At the intermediate level, the application directly chooses the transport it wishes to use, factoring the value of **NETPATH** and the contents of */etc/netconfig* into the choice as it sees fit.

Intermediate level: client side

The following code implements the client side of the same time service shown above, but written to the intermediate level of the RPC package.

The program requires the user to enter the transport over which the call will be made:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <netconfig.h>                  /* For netconfig structure */
#include "time_prot.h"



#define TOTAL (30)



/*
 *  Caller of trivial date service
 *  usage: calltime hostname netid
 */
main(argc,argv)
     int argc;
     char *argv[];
{
     struct netconfig *nconf;



     /* Declarations from previous example */



     if (argc != 3) {
```

```
                fprintf(stderr,"usage: %s host netid\n",argv[0]);
        }
        nettype = argv[2];
        if ((nconf = getnetconfigent(nettype)) == NULL) {
                fprintf(stderr, "Bad netid type: %s\n", nettype);
                exit(1);
        }
        client = clnt_tp_create(argv[1], TIME_PROG, TIME_VERS,
          nconf);
        if (client == NULL) {
                clnt_pcreateerror("Could not create client");
                exit(1);
        }



        /* Same as previous example after this point */



    }
```

The **netconfig** structure can be obtained by a call to **getnetconfigent(nettype)**. (See **getnetconfig**(3N) for more details.)

At this level, the program must explicitly make all decisions about network-selection.

Intermediate level: server side

This is the corresponding server. The administrator who starts the service is required to name, on the command line, the transport over which the service is provided:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <netconfig.h>                  /* For netconfig structure */
#include "time_prot.h"



static void time_prog();



/* Service to supply Greenwich mean time */
/* usage: server netid */
```

```
main(argc,argv)
     int argc;
     char *argv[];
{
     SVCXPRT *transp;
     struct netconfig *nconf;



     if (argc == 1) {
             fprintf(stderr, "usage: server netid \n");
             exit(1);
     }



     if ((nconf = getnetconfigent(argv[1])) == NULL) {
             fprintf(stderr, "Could not find info on %s\n",
               argv[1]);
             exit(1);
     }



     transp = svc_tp_create(time_prog, TIME_PROG, TIME_VERS,
       nconf);



     if (transp == NULL) {
             fprintf(stderr,"%s: cannot create %s service.\n",
               argv[0], argv[1]);
             exit(1)
             }



     svc_run();
}



static void
```

```
time_prog(rqstp, transp)
     struct svc_req *rqstp;
     SVCXPRT *transp;
{



     /* Code identical to Top Level version */
}
```

UnixWare 7 Release 7.1.4 - 27 April 2004

The expert level

At the expert level, network selection is done exactly as at the intermediate level. The only difference here is in the level of control that the application has over the details of the transport's configuration. Control at this level is much greater. These examples illustrate that control, which is exercised using the **clnt_tli_create** and **svc_tli_create** routines.

Expert level: client side

This is the client side of some code that implements a version of **clntudp_create** (the client-side creation routine for the UDP transport) in terms of **clnt_tli_create**. The example shows how to do network selection based on the family of the transport one wishes to use.

clnt_tli_create is normally used to create a client handle when:

- the application wants to pass an open file descriptor, which may or may not be bound
- the programmer wants to feed the server's address to the client
- the programmer wants to specify the send and receive buffer size (here, 8800 bytes)

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <netconfig.h>
#include <netinet/in.h>
/*
 * In an earlier implementation of RPC, the only transports supported
 * were TCP/IP and UDP/IP.  Here they are shown based on the Berkeley
 * socket, but implemented on the top of XTI/Streams.
 */
CLIENT *
clntudp_create(raddr, prog, vers, wait, sockp)
     struct sockaddr_in *raddr;      /* Remote address */
     u_long prog;            /* Program number */
     u_long vers;            /* Version number */
     struct timeval wait;    /* Time to wait */
     int *sockp;             /* fd pointer */
{
     CLIENT *cl;             /* Client handle */
     int madefd = FALSE;     /* Is fd opened here */
```

```
        int fd = *sockp;                /* fd */
        struct t_bind *tbind;   /* bind address */
        struct netconfig *nconf;        /* netconfig structure */
        void *handlep;



        if ((handlep = setnetconfig()) == 0) { /* No transports available
*/
                rpc_createerr.cf_stat = RPC_UNKNOWNPROTO;
                return ((CLIENT *)NULL);
        }
        /*
         * Try all the transports till it gets one which is
         * connectionless, family is INET and name is UDP
         */
        while (nconf = getnetconfig(handlep)) {
                if ((nconf->nc_semantics == NC_TPI_CLTS) &&
                    (strcmp(nconf->nc_protofmly, NC_INET) == 0) &&
                    (strcmp(nconf->nc_proto, NC_UDP) == 0))
                        break;
        }
        if (nconf == NULL) {
                rpc_createerr.cf_stat = RPC_UNKNOWNPROTO;
                goto err;
        }
        if (fd == RPC_ANYSOCK) {
                fd = t_open(nconf->nc_device, O_RDWR, NULL);
                if (fd == -1) {
                        rpc_createerr.cf_stat = RPC_SYSTEMERROR;
                        goto err;
                }
                madefd = TRUE;  /* The fd was opened here */
        }
        if (raddr->sin_port == 0) {     /* remote addr unknown */
                ushort_t sport;



                /*
                 * rpcb_getport() is a user provided routine
                 * which will call rpcb_getaddr and translate
                 * the netbuf address to port number.
                 */
                sport = rpcb_getport(raddr, prog, vers, nconf);
                if (sport == 0) {
```

```
                        rpc_createerr.cf_stat = RPC_PROGUNAVAIL;
                        goto err;
                }
                raddr->sin_port = sport;
        }



        /* Transform sockaddr_in to netbuf */
        tbind = (struct t_bind *)t_alloc(fd, T_BIND, T_ADDR);
        if (tbind == NULL) {
                rpc_createerr.cf_stat = RPC_SYSTEMERROR;
                goto err;
        }
        (void) memcpy(tbind->addr.buf, (char *)raddr,
          (int)tbind->addr.maxlen);
        tbind->addr.len = tbind->addr.maxlen;



        /* Bind endpoint to a reserved address */
        (void) bind_resv(fd);
        cl = clnt_tli_create(fd, nconf, &(tbind->addr), prog,
          vers, 8800, 8800);
        (void) endnetconfig(handlep);   /* Close the netconfig file */
        (void) t_free((char *)tbind, T_BIND);
        if (cl) {
                *sockp = fd;
                if (madefd == TRUE) {
                        /* fd should be closed while destroying the
handle */
                        (void) CLNT_CONTROL(cl, CLSET_FD_CLOSE, NULL);
                }
                /* Set the retry time */
                (void) CLNT_CONTROL(cl, CLSET_RETRY_TIMEOUT, &wait);
                return (cl);
        }



   err:
        if (madefd == TRUE)
                (void) t_close(fd);
        (void) endnetconfig(handlep);
        return ((CLIENT *)NULL);
   }
```

The network selection is done using the library functions **setnetconfig**, **getnetconfig**, and **endnetconfig**. (Note that **endnetconfig** is not called until after the call to **clnt_tli_create**, near the end of the example.)

clntudp_create can be passed an open **fd**, but if not `(fd == RPC_ANYSOCK)`, it will open its own using the **netconfig** structure for UDP.

If the remote address is not known, `(raddr->sin_port == 0)`, then it is obtained from the remote **rpcbind**. Note the call to **bind_resv**, which is a user-supplied function that serves to bind a transport endpoint to a reserved address. This call is necessary because there is no notion of a reserved address in RPC under XTI, as there is in both TCP and UDP. The implementation of this routine is of no interest here, because it is entirely transport specific. What is of interest is the scaffolding necessary to call it.

After the client handle has been created, the programmer can suitably customize it using calls to **clnt_control**. Here, the RPC library closes the file descriptor while destroying the handle (as it usually does with a call to **clnt_destroy** when it opens the **fd** itself) and sets the retry timeout period.

Expert level: server side

Below is the corresponding server code. It implements **svcudp_create** in terms of **svc_tli_create**, and calls the user provided **bind_resv** to bind the transport endpoint to a reserved address.

svc_tli_create is normally used when the application needs a fine degree of control, and especially if it is necessary to:

- pass an open file descriptor to the application
- pass the user's bind address
- set the send and receive buffer sizes (here being set to 8800 bytes)

The **fd** argument may be unbound when passed in. If it is, then it is bound to a given address, and the address is stored in a handle. If the bind address is set to **NULL**, and if the **fd** is initially unbound, it will be bound to any suitable address.

NOTE: It is the responsibility of the programmer to use **rpcb_set** to register the service with **rpcbind**.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <netconfig.h>
#include <netinet/in.h>
```

```
   /*
    * On the server side
    */
   SVCXPRT *
   svcudp_create(fd)
        register int fd;
   {
        struct netconfig *nconf;
        SVCXPRT *svc;
        int madefd = FALSE;
        int port;
        void *handlep;



        if ((handlep = setnetconfig()) == 0) { /* No transports available
*/
                nc_perror("server");
                return ((SVCXPRT *)NULL);
        }
        /*
         * Try all the transports till it gets one which is
         * a connection less, family is INET and name is UDP
         */
        while (nconf = getnetconfig(handlep)) {
                if ((nconf->nc_semantics == NC_TPI_CLTS) &&
                    (strcmp(nconf->nc_protofmly, NC_INET) == 0) &&
                    (strcmp(nconf->nc_proto, NC_UDP) == 0))
                        break;
        }
        if (nconf == NULL) {
                endnetconfig(handlep);
                fprintf(stderr, "UDP transport not available\n");
                return ((SVCXPRT *)NULL);
        }
        if (fd == RPC_ANYFD) {
                fd = t_open(nconf->nc_device, O_RDWR, NULL);
                if (fd == -1) {
                        (void) endnetconfig();
                        (void) fprintf(stderr,
                        "svcudp_create: could not open connection\n");
                        return ((SVCXPRT *)NULL);
                }
                madefd = TRUE;
```

```
        }



        /*
         * Bind Endpoint to a reserved address
         */
        port = bind_resv(fd);
        svc = svc_tli_create(fd, nconf, (struct t_bind *)NULL,
          8800, 8800);
        (void) endnetconfig(handlep);
        if (svc == (SVCXPRT *)NULL) {
                if (madefd)
                        (void) t_close(fd);
                return ((SVCXPRT *)NULL);
        }
        if (port == -1)
                /* Specifically set xp_port now */
                svc->xp_port =
                  ((struct sockaddr_in *)svc->xp_ltaddr.buf)->sin_port;
        else
                svc->xp_port = port;
        return (svc);
}
```

The network selection here is done in a similar way as in **clntudp_create**.

svcudp_create is set up to receive an open **fd**, but if it does not, it will open one itself using the selected **netconfig** structure.

bind_resv is a user-provided function that binds the **fd** to a reserved port if the caller is an RPC administrator.

The bottom level

At the bottom-level interface to RPC, the application can control all options, transport-related and otherwise. **clnt_tli_create**, and the other expert-level RPC interface routines are implemented on top of these bottom-level routines.

The programmer should not normally be using these low-level routines.

These routines are responsible for creating their own data structures, their own buffer management, the creation of their own RPC headers, and so on.

Callers of these routines (such as the expert level routine **clnt_tli_create**) are responsible for initializing the `cl_netid` and `cl_tp` fields within the client handle. The bottom level routines **clnt_dg_create** and **clnt_vc_create** are themselves responsible for populating the `clnt_ops` and `cl_private` fields.

For a created handle, `cl_netid` is the network identifier (for example, **udp**) of the transport and `cl_tp` is the device name of that transport (for example, */dev/udp*).

Bottom level: client side

The example here shows the use of local variables to control the exact details of the calls to **clnt_vc_create** and **clnt_dg_create**. Thus, these routines allow control of the transport to the lowest level:

```
switch (tinfo.servtype) {
        case T_COTS:
        case T_COTS_ORD:
                cl = clnt_vc_create(fd, svcaddr, prog, vers,
                  sendsz, recvsz);
                break;
        case T_CLTS:
                cl = clnt_dg_create(fd, svcaddr, prog, vers,
                  sendsz, recvsz);
                break;
        default:
                goto err;
}
```

Bottom level: server side

And, again, on the server side:

```
        /* call transport specific function. */



        switch(tinfo.servtype) {
                case T_COTS_ORD:
                case T_COTS:
                        xprt = svc_vc_create(fd, sendsz, recvsz);
                        break;



                case T_CLTS:
                        xprt = svc_dg_create(fd, sendsz, recvsz);
                        break;
                default:
                         goto err;
           }
```

UnixWare 7 Release 7.1.4 - 27 April 2004

Low-level data structures

For reference, here are the client- and server-side RPC handles, as well as an authentication structure.

```
/*
 * Client rpc handle.
 * Created by individual implementations
 * Client is responsible for initializing auth
 */



typedef struct {
    AUTH    *cl_auth;                /* authenticator */
    struct clnt_ops {
        enum clnt_stat  (*cl_call)(); /* call remote procedure */
        void        (*cl_abort)();    /* abort a call */
        void        (*cl_geterr)();   /* get specific error code */
        bool_t      (*cl_freeres)();  /* frees results */
        void        (*cl_destroy)();  /* destroy this structure */
        bool_t      (*cl_control)();  /* the ioctl() of rpc */
    } *cl_ops;
    caddr_t         cl_private;       /* private stuff */
    char            *cl_netid;        /* network token */
    char            *cl_tp;           /* device name */
} CLIENT;
```

The client-side handle contains an authentication structure. For a client program authenticate itself, it must initialize the `cl_auth` field to an appropriate authentication structure:

```
/*
 * Auth handle, interface to client side authenticators.
 */
typedef struct {
    struct  opaque_auth     ah_cred; /* credentials */
    struct  opaque_auth     ah_verf; /* verifier */
    union   des_block       ah_key;  /* DES key */
    struct auth_ops {
        void    (*ah_nextverf)();
        int     (*ah_marshal)();    /* nextverf & serialize */
        int     (*ah_validate)();   /* validate verifier */
```

```
        int     (*ah_refresh)();    /* refresh credentials */
        void    (*ah_destroy)();    /* destroy this structure */
    } *ah_ops;
    caddr_t ah_private;                             /* Private data */
} AUTH;
```

Within the **AUTH** structure, `ah_cred` contains the caller's credentials, and `ah_verf` contains the information necessary to verify those credentials. (See ``Authentication" for more details.)

This is the server-side transport handle:

```
/*
 * Server side transport handle
 */



typedef struct {
    int         xp_fd;          /* associated file descriptor */
    ushort_t    xp_port;        /* associated port number (obsolete) */
    struct xp_ops {
        bool_t      (*xp_recv)();    /* receive incoming requests */
        enum xprt_stat (*xp_stat)(); /* get transport status */
        bool_t      (*xp_getargs)(); /* get arguments */
        bool_t      (*xp_reply)();   /* send reply */
        bool_t      (*xp_freeargs)();/* free mem allocated for args */
        void    (*xp_destroy)();     /* destroy this struct */
    } *xp_ops;
    int         xp_addrlen;     /* length of remote addr. Obsolete */
    char        *xp_tp;         /* transport provider device name */
    char        *xp_netid;      /* network token */
    struct netbuf   xp_ltaddr;  /* local transport address */
    struct netbuf   xp_rtaddr;  /* remote transport address */
    char        xp_raddr[16];   /* remote address. Obsolete */
    struct opaque_auth xp_verf; /* raw response verifier */
    caddr_t     xp_p1;          /* private: for use by svc ops */
    caddr_t     xp_p2;          /* private: for use by svc ops */
    caddr_t     xp_p3;          /* private: for use by svc lib */
} SVCXPRT;
```

`xp_fd` is the file descriptor associated with the handle. Two or more server handles can share the same file descriptor.

`xp_netid` is the network identifier (for example, *udp*) of the transport on which this handle was created and `xp_tp` is the device name associated with that transport.

`xp_ltaddr` is the server's own bind address, while `xp_rtaddr` is the address of the remote caller and hence may change from call to call.

`xp_netid`, `xp_tp` and `xp_ltaddr` are initialized by **svc_tli_create** and other expert-level routines.

The rest of the fields are initialized by the bottom-level server routines **svc_dg_create** and **svc_vc_create**.

Low-level program testing using raw RPC

There are two pseudo-RPC interface routines provided to support program testing. These routines, **clnt_raw_create** and **svc_raw_create**, do not involve the use of any real transport. They exist to help the developer debug and test the non-communications-oriented aspects of an application before running it over a real network.

This is an example of their use:

```
/*
 * A simple program to increment a number by 1
 */
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpc/raw.h>



struct timeval TIMEOUT = {0, 0};
static void server();



main (argc, argv)
     int argc;
     char **argv;
{
     CLIENT *cl;
     SVCXPRT *svc;
     int num = 0, ans;



     if (argc == 2)
             num = atoi(argv[1]);
     svc = svc_raw_create();
     if (svc == NULL) {
             fprintf(stderr, "Could not create server handle\n");
             exit(1);
     }
     svc_reg(svc, 200000, 1, server, 0);
```

```
        cl = clnt_raw_create(200000, 1);
        if (cl == NULL) {
                clnt_pcreateerror("raw");
                exit(1);
        }
        if (clnt_call(cl, 1, xdr_int, &num, xdr_int, &ans,
          TIMEOUT) != RPC_SUCCESS) {
                clnt_perror(cl, "raw");
                exit(1);
        }
        printf("Client: number returned %d\n", ans);
        exit(0) ;
}



static void
server(rqstp, transp)
        struct svc_req *rqstp;
        SVCXPRT *transp;
{
        int num;



        switch(rqstp->rq_proc) {
        case 0:
                if (svc_sendreply(transp, xdr_void, 0) == NULL) {
                        fprintf(stderr, "error in null proc\n");
                        exit (1);
                }
                return;
        case 1:
                break;
        default:
                svcerr_noproc(transp);
                return;
        }
        if (!svc_getargs(transp, xdr_int, &num)) {
                svcerr_decode(transp);
                return;
        }
        num++;
        if (svc_sendreply(transp, xdr_int, &num) == NULL) {
                fprintf(stderr, "error in sending answer\n");
                exit (1);
```

```
        }
        return;
}
```

Note the following points:

- The server is not registered with **rpcbind**, and **svc_run** is not called. The last parameter to **svc_reg** is 0, which means that it will not register with **rpcbind**.
- All the RPC calls occur within the same thread of control.
- It is necessary that the server be created before the client.
- **svc_raw_create** takes no parameters.
- The server dispatch routine is the same as it is for normal RPC servers.

Advanced RPC programming techniques

This section addresses areas of occasional interest to programmers using the lower level interfaces of the RPC package. The topics discussed are:

Topic	RPC capability
Server processing	If calling **svc_run** is not feasible, a server can call the dispatcher directly
Broadcast RPC	Details of the broadcast mechanism are described
Batching	Efficiency is gained if a series of calls can be batched
Authentication	Two methods in common use are described
Port monitors	Details are provided for interfacing with the **inetd** and **listen** port monitors
Threads	How to write RPC programs with multiple threads of execution

Server processing

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**.

If the other activity involves waiting on a file descriptor, however, the **svc_run** call will not work. Given the file descriptors of the transport endpoints associated with the programs being waited on, a process can have its own `select` that waits on both the RPC file descriptors and its own file descriptors.

For example, consider the code below, which is for an earlier version of **svc_run**. Note that **svc_fdset** is a bit mask of all the file descriptors that RPC is using for services. The mask can change every time any RPC library routine is called, because descriptors are constantly being opened and closed:

```
void
svc_run ()
{
     fd_set readfds;
     extern int errno;



     for (;;) {
             readfds = svc_fdset;
             switch (select(_rpc_dtbsize(), &readfds,
               (fd_set *)0, (fd_set *)0, (struct timeval *)0)) {



             case -1:
                     if (errno == EINTR) {
                             continue;
                     }
                     /*
                      * log an error: svc_run: select failed
                      */
                     return;
             case 0:
                     continue;
             default:
                     svc_getreqset(&readfds);
             }
```

```
        }
    }
```

The current version of **svc_run** calls **svc_getreq_poll**. A process can bypass **svc_run** and call **svc_getreqset** (the dispatcher) or **svc_getreq_poll** directly. The **svc_getreqset** and **svc_getreq_poll** routines in turn call **svc_getreq_common**. These functions are described on the **rpc_svc_reg**(3rpc) manual page.

CAUTION: The **svc_run** and **svc_getreq_poll** routines are not thread-safe and should not be called from multiple threads of execution. See ``Writing multithreaded RPC procedures'' and also ``Multithreaded network programming'' for more information about using RPC with the Threads Library.

UnixWare 7 Release 7.1.4 - 27 April 2004

Broadcast RPC

rpcbind is a daemon that converts RPC program numbers into network addresses comprehensible to any transport provider. **rpcbind** supports broadcast RPC. These are the main differences between broadcast RPC and normal RPC calls:

- Normal RPC expects one answer, whereas broadcast RPC expects many answers (one or more answer from each responding machine).
- Broadcast RPC can only be performed on connectionless protocols that support broadcasting, such as UDP.
- The implementation of broadcast RPC treats all unsuccessful responses as garbage by filtering them out. Thus, if there is a version mismatch between the broadcaster and a remote service, the user of broadcast RPC never knows.
- All broadcast messages are sent to **rpcbind**'s network address. Thus, only services that register themselves with **rpcbind** are accessible via the broadcast RPC mechanism.
- The size of broadcast requests is limited to the MTU (Maximum Transfer Unit) of the local network. For Ethernet, the MTU is 1500 bytes.

The following illustrates how **rpc_broadcast** is used and describes its arguments:

```
#include <rpc/clnt.h>
#include <rpc/rpcb_clnt.h>
     . . .
enum clnt_stat       clnt_stat;
     . . .
clnt_stat = rpc_broadcast(prognum, versnum, procnum,
  inproc, in, outproc, out, eachresult, nettype)
     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 */
     resultproc_t   eachresult;/* call with each result gotten */
     char      *nettype;       /* transport type */
```

The procedure **eachresult** is called each time a valid result is obtained. It returns a boolean that specifies whether the user wants more responses.

```
bool_t done;
     . . .
done = eachresult(resultsp, raddr, nconf)
     caddr_t resultsp;
     struct netbuf *addr;     /* Addr of responding machine */
     struct netconfig *nconf; /* Transport which responded */
```

If **done** is **TRUE**, then broadcasting stops and **rpc_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

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

RPC messages can be placed in a ``pipeline" of calls to a desired server; this is called batching. Batching assumes that:

- each RPC call in the pipeline requires no response from the server, and the server does not send a response message
- the pipeline of calls is transported on a reliable byte stream transport such as TCP/IP

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**(2) 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 to flush the pipeline.

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

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



void windowdispatch();



main ()
{
     int num;
```

```
        num = svc_create(windowdispatch, WINDOWPROG,
          WINDOWVERS, "tcp");
        if (num == 0){
                fprintf(stderr, "can't create an RPC server\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 an error occurred
                         */
                        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 that specifies whether the procedure should respond.

To take advantage of batching (using the code above), the client must make RPC calls on a reliable byte stream transport. The calls must have the following attributes:

- the XDR routine for the result must be zero (**NULL**)
- the RPC call's timeout must be zero

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

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



main(argc, argv)
        int argc;
        char **argv;
{
        struct timeval total_timeout;
        register CLIENT *client;
        enum clnt_stat clnt_stat;
```

```
        char buf[1000], *s = buf;



        if ((client = clnt_create(argv[1], WINDOWPROG,
          WINDOWVERS, "tcp")) == NULL) {
                clnt_pcreateerror("clnt_create");
                exit(1);
        }
        total_timeout.tv_sec = 0;
        total_timeout.tv_usec = 0;
        while (scanf("%s", s) != EOF) {
                clnt_call(client, RENDERSTRING_BATCHED,
                        xdr_wrapstring, &s, xdr_void, NULL,
total_timeout);
        }



        /* 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.

Batching performance

The example of batching was completed to render all the lines in a 2000 line file. The rendering service did nothing but throw the lines away.

The example was run in four configurations, with the following results:

Configuration	RPC	Time

Machine to itself	Regular	50 seconds
Machine to itself	Batched	16 seconds
Machine to another	Regular	52 seconds
Machine to another	Batched	10 seconds

Running **fscanf** on the same file only requires six seconds. These timings show the advantage of protocols that allow for overlapped execution.

Authentication

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

Every RPC call is subjected to a style of authentication by the RPC package on the server. Similarly, the RPC client package generates and sends authentication parameters suitable for the style of authentication in effect. The default authentication style is **AUTH_NONE** (none).

Just as different transports can be used when creating RPC clients and servers, different forms of authentication can be associated with RPC clients.

The authentication subsystem of the RPC package is open ended. That is, numerous styles of authentication are easy to support; programmers can design their own authentication style and easily configure the RPC package to support it.

In addition to **AUTH_NONE**, the RPC package already supports the following authentication styles:

AUTH_SYS
: An authentication style based on traditional UNIX operating system process permissions authentication.

AUTH_SHORT
: An alternate form of **AUTH_SYS** used by some servers for efficiency. Client programs using **AUTH_SYS** authentication should be prepared to receive **AUTH_SHORT** response verifiers from some servers. See ``Authentication protocols'' for details.

AUTH_DES
: An authentication style based on DES encryption techniques.

AUTH_NONE: client side

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

```
clnt = clnt_create(host, prognum, versnum, nettype)
```

the appropriate transport instance defaults the associated authentication handle to be

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

NOTE: If the programmer creates a new style of authentication, the programmer is responsible for destroying it with **auth_destroy(clnt->cl_auth)**. This should always be done, to conserve memory.

AUTH_NONE: server side

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

```
/*
 * An RPC Service request
 */
struct svc_req {
        u_long  rq_prog;                        /* service program number
*/
        u_long  rq_vers;                        /* service protocol vers
num */
        u_long  rq_proc;                        /* desired procedure
number */
        struct opaque_auth rq_cred;             /* raw credentials from
wire */
        caddr_t rq_clntcred;            /* credentials (read only) */
        SVCXPRT *rq_xprt;                       /* associated transport */
};
```

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

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

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

- The request's `rq_cred` is well formed. Thus the service implementor may inspect the request's

`rq_cred.oa_flavor` to determine the 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 supported by the RPC package.

- 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 **AUTH_NONE**, **AUTH_SYS**, **AUTH_SHORT** and **AUTH_DES** styles are currently supported, so (currently) `rq_clntcred` could be cast only as a pointer to an **authsys_parms**, **short_hand_verf**, or **authdes_cred** structure. If `rq_clntcred` is **NULL**, the service implementor may wish to inspect the other (opaque) fields of `rq_cred` if the service knows about a new type of authentication that the RPC package does not know about.

AUTH_SYS authentication

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

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

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

```
/*
 * AUTH_SYS style credentials.
 */
struct authsys_parms {
     u_long  aup_time;               /* credentials creation time */
     char    *aup_machname;  /* host name where client is */
     uid_t   aup_uid;                /* client's effective uid */
     gid_t   aup_gid;                /* client's current group id */
     u_int   aup_len;                /* element length of aup_gids */
     gid_t   *aup_gids;      /* array of groups user is in */
};
```

These fields are set by **authsys_create_default** by invoking the appropriate system calls.

The following shows the server for a remote procedure, *RUSERPROC_**n***, that computes the number of users on the network. As a trivial demonstration of authentication usage, this server checks **AUTH_SYS** credentials and does not service requests from callers whose user ID is 16:

```
nuser(rqstp, transp)
     struct svc_req *rqstp;
     SVCXPRT *transp;
{
```

```
struct authsys_parms *SYS_cred;
uid_t 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_SYS:
        SYS_cred =
                (struct authsys_parms *)rqstp->rq_clntcred;
        uid = SYS_cred->aup_uid;
        break;
case AUTH_NONE:
default:
        svcerr_weakauth(transp);
        return;
}
switch (rqstp->rq_proc) {
case RUSERSPROC_n:
        /*
         * 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:

- It is customary not to check the authentication parameters associated with the **NULLPROC** (procedure number zero).
- The server should call **svcerr_weakauth** if the authentication parameter's type is not suitable for the service.
- The service protocol itself should return status for access denied; in this example, the protocol does not have such a status, so the service primitive **svcerr_systemerr** is called instead.

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

AUTH_DES authentication

AUTH_DES authentication is recommended for programs that require more security than that offered by the **AUTH_SYS** style of authentication.

AUTH_SYS authentication is easy to defeat. For example, instead of using **authsys_create_default**, a program could call **authsys_create**, and then change the RPC authentication handle to give itself any desired user ID and hostname.

The details of the **AUTH_DES** authentication protocol are complicated and are not explained here. See the ``AUTH_DES authentication'' for more information.

For **AUTH_DES** authentication to work, the **keyserv**(1Mbnu) daemon must be running on both the server and client machines. The users on these machines need public/secret key pairs assigned by the RPC administrator in the **publickey**(4bnu) database. And, they need to have decrypted their secret keys using the **keylogin**(1bnu) command.

AUTH_DES: client side

If a client wishes to use **AUTH_DES** authentication, it must set its authentication handle appropriately. This is

an example:

```
cl->cl_auth = authdes_seccreate(servername, 60, server, NULL);
```

The first argument is the network name or ``netname'' of the owner of the server process. Typically, server processes are root processes and their netname can be derived using the following call:

```
char servername[MAXNETNAMELEN];



host2netname(servername, rhostname, NULL);
```

rhostname is the hostname of the machine on which the server process is running. *host2netname* populates *servername* to contain this root process's netname. If the server process was run by a regular user, you could use the call *user2netname* instead. This is an example for a server process with the same user ID as the client:

```
char servername[MAXNETNAMELEN];



user2netname(servername, getuid(), NULL);
```

The last argument to both of these calls, *user2netname* and *host2netname*, is the name of the naming domain where the server is located. The **NULL** used here means ``use the local domain name.''

The second argument to **authdes_seccreate** is a lifetime for the credential. Here it is set to sixty seconds which means that the credential will expire 60 seconds from now. If some mischievous program tries to reuse the credential, the server RPC subsystem will recognize that it has expired and will not grant any requests. If the same mischievous program tries to reuse the credential within the sixty second lifetime, it will still be rejected, because the server RPC subsystem remembers credentials it has seen in the near past, and will not grant requests to duplicates.

The third argument to **authdes_seccreate** is the name of the host to synchronize with. For **AUTH_DES** authentication to work, the server and client must agree on the time. Here we pass the hostname of the server itself, so the client and server will both be using the same time: the server's time. The argument can be **NULL**, which means ``don't bother synchronizing.'' A program should pass **NULL** only if sure the client and server are already synchronized.

The final argument to **authdes_seccreate** is the address of a DES encryption key to use for encrypting timestamps and data. If this argument is **NULL**, as it is in this example, a random key will be chosen. The client may find out the encryption key being used by consulting the `ah_key` field of the authentication handle.

AUTH_DES: server side

The server side is simpler than the client side. This is the previous example rewritten to use the **AUTH_DES** style instead of **AUTH_SYS**:

```
#include <rpc/rpc.h>
     . . .
     . . .
nuser(rqstp, transp)
     struct svc_req *rqstp;
     SVCXPRT *transp;
{
     struct authdes_cred *DES_cred;
     uid_t uid;
     gid_t gid;
     int gidlen;
     gid_t gidlist[10];
     /*
      * we don't care about authentication for null proc
      */



     if (rqstp->rq_proc == NULLPROC) {
             /* same as before */
     }



     /*
      * now get the uid
      */
     switch (rqstp->rq_cred.oa_flavor) {
     case AUTH_DES:
             DES_cred =
               (struct authdes_cred *) rqstp->rq_clntcred;
             if (! netname2user(DES_cred->adc_fullname.name,
               &uid, &gid, &gidlen, gidlist)) {
                     fprintf(stderr, "unknown user: %s\n",
                       DES_cred->adc_fullname.name);
                     svcerr_systemerr(transp);
                     return;
             }
             break;
     case AUTH_NONE:
```

```
        default:
                svcerr_weakauth(transp);
                return;
        }



        /*
         * The rest is the same as before
         */
```

Note the use of the routine **netname2user**, the inverse of **user2netname:** it takes a network ID and converts to a local system ID. **netname2user** also supplies the group IDs, not used in this example, but which may be useful to other programs.

UnixWare 7 Release 7.1.4 - 27 April 2004

Using port monitors

An RPC server can be started from port monitors such as **inetd**(1Mtcp) and **listen**(1M). These port monitors listen for requests for the services and spawn servers in response to those requests. The forked server process is passed file descriptor 0, on which the request has been accepted. For **inetd**, after the server has serviced the request, it may exit immediately or wait a given interval of time for another service request to come in.

NOTE: For **listen**, a service may exit (after servicing a single request) or poll file descriptor 0 (waiting for additional requests). The listener must be administered to expect the correct behavior for each of its services. **listen** will always spawn a new process rather than give a request to a waiting server process.

The following routine can be used to create a service:

```
transp = svc_tli_create(0, nconf, NULL, 0, 0)
```

nconf is the **netconfig** structure of the transport on which the request came in.

Because the port monitors have already registered the service with **rpcbind**, there is no need for the service to register itself. Nevertheless, it must call **svc_reg**:

```
svc_reg(transp, PROGNUM, VERSNUM, dispatch, NULL)
```

The **netconfig** structure here is **NULL**.

NOTE: Programmers should study **rpcgen**-generated server stubs to better see the sequence in which these routines are called.

For connection-oriented transports, the following routine provides a lower level interface:

```
transp = svc_fd_create(0, recvsize, sendsize);
```

The file descriptor passed here is 0. The user may set the value of ***recvsize*** or ***sendsize*** to any appropriate buffer size. If they use a 0 in either case, a system default size will be chosen. This routine should be used by application servers that do not do any listening of their own, that is, servers that simply do their job and return.

Using inetd

The format of entries in */etc/inetd.conf* for RPC services is as follows:

```
rpc_prog/vers socket_type rpcproto flags uid pathname args
```

Here:

rpc_prog
is the symbolic name of the program as it appears in **rpc**(4tcp).

vers
is the version number

socket_type
is one of **dgram** for connectionless transport, or **stream** for virtual circuit transport

proto
is transport protocol, such as **udp** or **tcp** and must correspond to the specified ***socket_type*** (for example, **udp** for **dgram**, and **tcp** for **stream**)

flags
is one of **wait** or **nowait**

uid
is a user ID that must exist in */etc/passwd*

pathname
is the full pathname of the server daemon

args
are arguments to be passed to the daemon when it is invoked

For example:

```
mountd dgram rpc/udp wait root /usr/sbin/rpc.mountd rpc.mount
```

For more information, see **inetd.conf**(4tcp).

Using the listener

We will assume here that the reader already knows the details of setting up the listener process and of using **pmadm**. The following shows how to use **pmadm** to add RPC services:

```
pmadm -a -p pm_tag -s svctag -i id -v ver \
   -m "`nlsadmin -c command -D -R prog:vers`"
```

Here **-a** means add a service, **-p *pm_tag*** specifies a tag associated with the port monitor providing access to the service, **-s *svctag*** is the server's identifying code, **-i *id*** is the */etc/passwd* user ID assigned to service ***svctag***, **-v *ver*** is the version number for the port monitor's database file and **-m** specifies the **nlsadmin** command for invoking the service. **nlsadmin** may have additional arguments. For example, to add version 1 of a remote program server named *rusersd* the **pmadm** command might be:

**pmadm -a -p tcp -s rusers -i root -v `nlsadmin -V` **
**-m "`nlsadmin -c /usr/lib/netsvc/rusers/rpc.rusersd **
-D -R 100002:1`"

Here, the command is given *root* permissions, and is made available over TCP transports. If the service is to be a daemon process, the **-c** command must be replaced by a **-p *pipe*** where ***pipe*** is a FIFO or named STREAM.

NOTE: Because of the complexity of the arguments and options that can follow the **pmadm -a** invocation, it may be convenient to use a command script to add RPC services.

After adding a service, the listener must be reinitialized before the service will be available. This is accomplished by forcing the listener to re-read its database, as follows (note that **rpcbind** must be running):

sacadm -x -p *pmtag*

For additional information, see ``The Service Access Facility'' and the **listen**(1M), **pmadm**(1M), and **sacadm**(1M) manual pages.

Writing multithreaded RPC procedures

In UnixWare 7, transport-independent RPC functions may safely be called from multiple threads of execution. The following sections describe some considerations for the programmer of multithreaded applications (i.e., applications linked with the Threads Library, */usr/lib/libthread.so*). See ``Multithreaded network programming'' for more information about using RPC with the Threads Library.

Client handle usage

Client handles created by one thread are shared by all other threads, but only one RPC request is allowed at a time per thread. A process can invoke any number of RPC calls simultaneously, but it must use a different handle for each call. In other words, explicit synchronization is required for several threads to use the same handle. Note that each client handle contains its own authentication information and maintains it in a different way, depending on the type of authentication. This fact forces a process to hold separate authentication information for every handle, even though sibling threads have a single credential structure.

Server handle usage

Server handles are also shared by all threads, and only one request may be outstanding at a time per handle. Any handle can be registered or unregistered with the system, even when a request associated with the handle is being processed. Destroying it, however, may result in undefined behavior by the server. Server-side services, or dispatch routines, can also be added during **svc_run** or user-supplied alternatives to **svc_run**.

Servicing multiple RPC requests

In UnixWare 7, there is a multithreaded counterpart to **svc_run**, namely **svc_run_parallel,** which dynamically creates and deletes threads to service incoming RPC requests. Currently **svc_run_parallel** supports only connectionless transport providers. **svc_run_parallel** calls **svc_getreq_poll_parallel,** a thread-safe version of **svc_getreq_poll**. The **svc_getreq_poll_parallel** routine in turn calls **svc_getreq_common**. These routines are described on the **rpc_svc_reg**(3rpc) manual page.

NOTE: Although **svc_getreq_common** is thread-safe, a different file descriptor must be specified in each concurrent call to **svc_getreq_common**.

An example of registering multiple versions of a remote procedure

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 **ruser** program that returns an unsigned short rather than a long. If we name this version ***RUSERSVERS_SHORT***, then a server that wants to support both versions would do a double register. The same server handle would be used for both of these registrations.

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

Both versions can be handled by the same C procedure:

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



    switch (rqstp->rq_proc) {
    case NULLPROC:
        if (!svc_sendreply(transp, xdr_void, 0)) {
            fprintf(stderr, "can't reply to RPC call\n");
            return (1);
        }
        return (0);
    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;
    }
    return (0);
}
```

An example of a remote copy program using connection-oriented transports

This is an example that copies a file from one system to another. The initiator of the RPC **send** call takes its standard input and sends it to the server *receive*, which prints it on standard output. This also illustrates an XDR procedure that behaves differently on serialization than on deserialization.

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



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

Note that in the following two screens, the serializing and deserializing is done only by **xdr_bytes**.

```
/* 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();



        if (argc < 2) {
                fprintf(stderr, "usage: %s servername\n", argv[0]);
                exit(1);
        }
        if (callcots(argv[1], RCPPROG, RCPPROC, RCPVERS,
          xdr_rcp, stdin, xdr_void, 0) != 0) {
                exit(1);
        }
        exit(0);
}



callcots(host, prognum, procnum, versnum, inproc, in, outproc, out)
```

```
        char *host, *in, *out;
        xdrproc_t inproc, outproc;
{
        enum clnt_stat clnt_stat;
        register CLIENT *client;
        struct timeval total_timeout;



        if ((client = clnt_create(host, prognum, versnum,
          "circuit_v")) == NULL) {
                perror("clnt_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);
        if (clnt_stat != RPC_SUCCESS) {
                clnt_perror("callcots");
        }
        return ((int)clnt_stat);
}

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



main ()
{
        int rcp_service(), xdr_rcp();



        if (svc_create(rpc_service, RCPPROG, RCPVERS,
          "circuit_v") == 0) {
                fprintf("svc_create: 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:
             if (!svc_getargs(transp, xdr_rcp, stdout)) {
                     svcerr_decode(transp);
                     return (1);
             }
             if (!svc_sendreply(transp, xdr_void, 0)) {
                     fprintf(stderr, "can't reply\n");
                     return (1);
             }
             return (0);
     default:
             svcerr_noproc(transp);
             return (1);
     }
}
```

Note that on the server side no explicit action was taken after receiving the arguments. This is because **xdr_rcp** did all the necessary dirty work automatically.

An example of a server placing a call back to a client

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

To do an RPC callback, a program number is needed to make the RPC call. Because this will be a dynamically generated program number, it should be in the transient range, 0x40000000 - 0x5fffffff. In the following example, the routine **gettransient** returns a valid program number in the transient range, and registers it with **rpcbind**. The call to **rpcb_set** is a test and set operation, in that it indivisibly tests whether a program number has already been registered, and if it has not, then reserves it.

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



gettransient(vers, nconf, address)
     int vers;
     struct netconfig *nconf;
     struct netbuf *address;
{
     static int prog = 0x40000000;



     while (! rpcb_set(prog++, vers, nconf, address))
             continue;
     return (prog - 1);
}
```

The following program 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 around to receive a callback from the server at that program number. The server registers the program **EXAMPLEPROG**, so that it can receive the RPC call informing it of the callback program number. Then at some random time (on receiving an **ALRM** signal in this example), it sends a callback RPC call, using the program number it received earlier.

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



main(argc, argv)
     int argc;
     char **argv;
{
     SVCXPRT *xprt;
     struct netconfig *nconf;
     int prognum;
     enum clnt_stat stat;



     if (argc != 3) {
             fprintf(stderr, "usage: clnt host transport\n");
             exit (1);
     }
     nconf = getnetconfigent(argv[2]);
     if (nconf == NULL) {
             fprintf(stderr, "unknown transport\n");
             exit (1);
     }
     xprt = svc_tli_create(RPC_ANYFD, nconf,
       (struct t_bind *)NULL, 0, 0);
     if (xprt == (SVCXPRT *)NULL) {
             fprintf(stderr, "could not create server handle\n");
             exit (1);
     }
     prognum = gettransient(1, nconf, &xprt->xp_ltaddr);
     fprintf(stderr, "client gets prognum %d\n", prognum);
     if (svc_reg(xprt, prognum, 1, callback, NULL)
       == FALSE) {
             fprintf(stderr, "could not register service\n");
             exit(1);
     }
     stat = rpc_call(argv[1], EXAMPLEPROG, EXAMPLEVERS,
       EXAMPLEPROC_CALLBACK, xdr_int, &prognum, xdr_void,
```

```
        NULL, NULL);
      if (stat != RPC_SUCCESS) {
            clnt_perrno(stat);
            exit(1);
      }
      svc_run();
      exit(1);
}



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);
                  }
                  return (0);
      }
      return (2);
}
```

This example shows how **svc_tli_create** can be used when it is necessary to explicitly choose the program number by calling **rpcb_set** until it succeeds. (Here it was not required that a service be registered on a given transport, and the example could simply have used a ``generic" network type.) After creating the handle, **svc_reg** is called (with the last parameter given as **NULL**) to register the dispatch function with the dispatcher. Once the server side is ready, it then notifies the actual server of its dynamic program number with **rpc_call**. On success it then waits for requests from the remote server.

In the following example, the server makes an RPC call to the client on an **ALARM** signal, but only if the

client has passed the program number to the server. This server example illustrates the simplicity of the code when one is using **rpc_reg**.

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



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



main ()
{
     if (argc != 2) {
             fprint(stderr, "usage: server hostname\n");
             exit (1);
     }
     hostname = argv[1];
     rpc_reg(EXAMPLEPROG, EXAMPLEVERS, EXAMPLEPROC_CALLBACK,
       getnewprog, xdr_int, xdr_void, NULL);
     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;



     if (pnum == 0) {
            alarm (10);
            return;
     }
     ans = rpc_call(hostname, pnum, 1, 1, xdr_void, 0,
       xdr_void, 0, NULL);
     if (ans != RPC_SUCCESS) {
            fprintf(stderr, "server: ");
            clnt_perrno(ans);
     }
}
```

An example of performing memory allocation with XDR

XDR routines not only perform input and output, they also perform memory allocation.

The second parameter of **xdr_array** is a pointer to an array, rather than the array itself.

NOTE: This is true for most XDR routines. The indirection is necessary because these routines often allocate memory.

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);
```

To have XDR to do the allocation, this routine must be rewritten 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 should normally 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 earlier, 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 **rpc_call**, the serializing part is used.
- When called from **svc_getargs**, the deserializer is used.
- When called from **svc_freeargs**, the memory deallocator is used.

When building simple programs like those given as examples in this section, a programmer does not have to worry about the three modes.
