Addendum to CONVEX Network File Reference Set

Document No. 710-001699-201

November 1988

CONVEX Computer Corporation
Richardson, Texas

Printed in the United States of America

Replacement Instructions for *CONVEX Network File System Reference Set*

This addendum updates the *CONVEX Network File System Reference Set*, Document No. 710-001730-202, Second Edition, Rev. 1, as follows:

- This documentation supports CONVEX UNIX V7.0.
- This documentation corrects errors in the previous manual.
- The *CONVEX Network File System Reference Set* document number changes to Document No. 710-001730-203.
- The *CONVEX Network File System Reference Set* changes from Second Edition, Rev. 1 to Second Edition, Rev. 2.

Other changes are described in the Revision Information page. Please update your document as indicated in the following table.

Remove Pages	**Insert Pages**
title and copyright pages Revision Information iii through vi	title and copyright pages Revision Information iii through vi
NFS Overview	
title and copyright pages over-1-5 through over-1-6	title and copyright pages over-1-5 through over-1-6
RPC Programming Guide	
title and copyright pages iii through iv Chapter 2 Index (pages 1 through 3)	title and copyright pages iii through iv Chapter 2 Index (pages 1 through 3)
Master Index (pages 1 through 11) Reader's Forum	Master Index (pages 1 through 11) Reader's Forum

Place this sheet after the title and copyright pages for future reference.

CONVEX Network File System Reference Set

Document No. 710-001730-203

Second Edition, Rev. 2
November 1988

CONVEX Computer Corporation
Richardson, Texas

CONVEX Network File System
Reference Set
Order No. DSW-111
Second Edition, Rev. 2

Printed in the United States of America

Revision Information for
CONVEX Network File System
Reference Set

Edition	Document No.	Description
Second Rev. 2	710-001730-203	Released with CONVEX UNIX V7.0, November 1988. Includes the following changes: *NFS Overview* Chapter 1, corrected sample entries to the */etc/exports* file *RPC Programming Guide* Chapter 2, corrected sample program using *callrpc()*
Second Rev. 1	710-001730-202	Released with CONVEX UNIX V6.2, April 1988.
2.0	710-001730-201	Released with CONVEX UNIX V6.1, October 1987.
1.0	710-000630-000	Initial release with CONVEX UNIX V6.0, April 1987.

NFS Overview

List of Figures

NFS Protocol Spec.

YP Protocol Spec.

RPC Protocol Spec.

XDR Protocol Spec.

RPC Programming Guide

Appendices

List of Tables

List of Figures

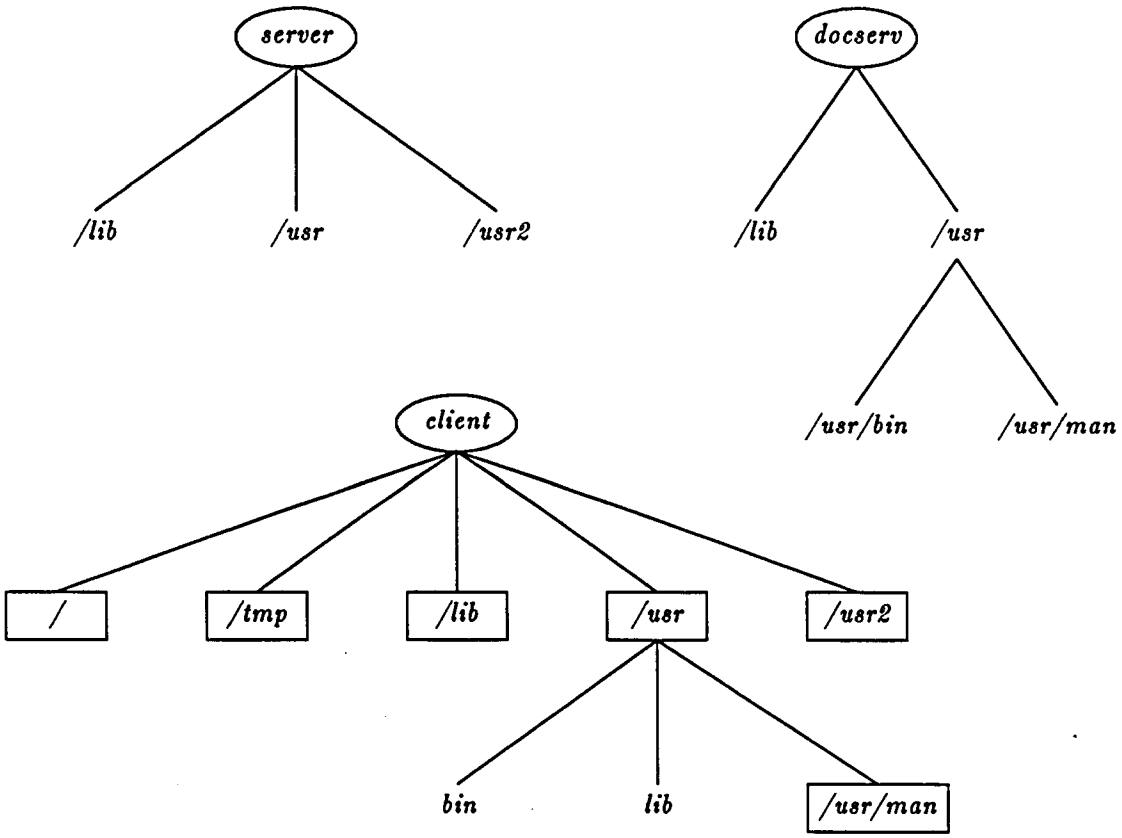
CONVEX Network File System Overview

November 1988

CONVEX Computer Corporation
Richardson, Texas

Printed in the United States of America

Figure 1–1: Mounting Directories



Exporting a Filesystem

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

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

```
/usr/proj    -access=cohort
```

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

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

Because both you and your colleague can change files on */usr/proj*, it is best to use the *rcs*(1) source code control system for concurrency control.

Note that you can enable or disable over-the-net superuser privileges for client machines on either a machine-by-machine basis, or a filesystem-by-filesystem basis. For details, see "Superuser Access to Remote Files" in the *CONVEX Network File System System Manager's Guide.*

Administering a Server Machine

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

```
/
/usr
/usr2
/usr/src      -access=staff
```

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

CONVEX Remote Procedure Call Programming Guide

November 1988

CONVEX Computer Corporation
Richardson, Texas

CONVEX Remote Procedure Call
Programming Guide

Printed in the United States of America

Table of Contents

Appendices

List of Tables

List of Figures

2

Higher Layers of *rpc*

Highest Layer

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

```
#include <stdio.h>

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

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

rpc library routines such as *rnusers()* are in the *rpc* services library *librpcsvc.a*. Thus, the program above should be compiled with:

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

This routine, and other *rpc* library routines, are documented in Section 3 of the *CONVEX UNIX Programmer's Manual*. Table 2-1 shows the *rpc* service library routines available to the C programmer.

Table 2-1: *rpc* Service Library Routines

Routine	*Description*
rnusers()	Return number of users on remote machine
rusers()	Return information about users on remote machine
havedisk()	Determine if remote machine has disk
rstat()	Get performance data from remote kernel
rwall()	Write to specified remote machines
getrpcport()	Get *rpc* port number
yppasswd()	Update user password in yellow pages

The other *rpc* services—*ether, mount, rquota,* and *spray*— are not available to the C programmer as library routines. They do, however, have *rpc* program numbers so they can be invoked with *callrpc(),* discussed in the next section.

Intermediate Layer

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

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

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

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

A program number, version number, and procedure number defines each *rpc* procedure. The program number defines a group of related remote procedures, each of which has a different procedure number. Each program also has a version number, so when a minor change is made to a remote service (adding a new procedure, for example), a new program number doesn't have to be assigned. When you want to call a procedure to find the number of remote users, you look up the appropriate program, version, and procedure numbers in a manual, similar to when you look up the name of memory allocator when you want to allocate memory. The simplest routine in the *rpc* library used to make remote procedure calls is *callrpc().* It has eight parameters. The first is the name of the remote machine. The next three parameters are the program, version, and

procedure numbers. The following two parameters define the argument of the *rpc* call, and the final two parameters are for the return value of the call. If it completes successfully, *callrpc()* returns zero, but nonzero otherwise. The exact meaning of the return codes is found in *<rpc/clnt.h>*, and is in fact an *enum clnt_stat* cast into an integer.

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

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

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

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

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

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

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

char *nuser();

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

The *registerrpc()* routine establishes what C procedure corresponds to each *rpc* procedure number.

The first three parameters, *RUSERPROG, RUSERSVERS,* and *RUSERSPROC_NUM,* are the program, version, and procedure numbers of the remote procedure to be registered; *nuser()* is the name of the C procedure implementing it; and *xdr_void* and *xdr_u_long* are the types of the input to and output from the procedure.

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

Warning: the UDP transport mechanism can only deal with arguments and results fewer than 8 Kbytes in length.

Assigning Program Numbers

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

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

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

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

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

Please include a complete protocol specification similar to those in this manual for *nfs* and *yp.* You will be given a unique program number in return.

Passing Arbitrary Data Types

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

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

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

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

call *callrpc()* as:

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

where *xdr_simple()* is written as:

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

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

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

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

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

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

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

and make an *rpc* call such as:

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

with *xdr_varintarr()* defined as:

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

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

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

```
int intarr[SIZE];

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

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

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

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

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

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

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

Index

Index

D

E

N

O

P

R

S

Y