SPRY TI-RPC Guide

SPRY TI-RPC Guide, Copyright December, 1993

Introduction

The SPRY TI-RPC (Transport Independent Remote Procedure Calls) for Windows Sockets provides a standard API for developing client and server applications that use remote procedure call technology. TI-RPC implements a standard RPC protocol, making it well-suited for distributed computing environments.

The standard API and protocol supported by the TI-RPC for Windows Sockets is an implementation of the Sun Microsystems, Inc, Open Network Computing Transport Independent Remote Procedure Call (ONC+/TI-RPC) specification. ONC+/TI-RPC has received wide acceptance and is offered by a variety of vendors.

Using SPRY's TI-RPC for Windows Sockets high-level API, you can develop applications that use TCP/UDP and invoke no transport functions directly. The lower levels of the API provide increasing degrees of control over transport-specific functions which you can program if required.

TI-RPC for Windows Sockets will aid in the development of distributed applications. By using TI-RPC, a distributed application can take advantage of the following features:

- Machine independence
- Portable code
- Heterogeneous environment

What's Included

The TI-RPC Guide includes:

- Documentation on TIRPC procedure calls
- The TIRPC.LIB file

- The TIRPC.DLL file
- Header files for constant definition and function prototypes

Index to Routines

This manual uses the traditional UNIX "manual page" format to document commands. The following table lists RPC routines and the sections in which they are described:

RPC Routine	Manual Reference Page
auth_destroy	rpc_clnt_auth
authnone_create	rpc_clnt_auth
authsys_create	rpc_clnt_auth
authsys_create_default	rpc_clnt_auth
authunix_create	rpc_soc
authunix_create_default	rpc_soc
callrpc	rpc_soc
clnt_broadcast	rpc_soc
clnt_call	rpc_clnt_calls
clnt_control	rpc_clnt_create
clnt_create	rpc_clnt_create
clnt_destroy	rpc_clnt_create
clnt_dg_create	rpc_clnt_create
clnt_freeres	rpc_clnt_calls
clnt_geterr	rpc_clnt_calls
clnt_pcreateerror	rpc_clnt_create
clnt_perrno	rpc_clnt_calls
clnt_perror	rpc_clnt_calls
clnt_raw_create	rpc_clnt_create

clnt_spcreateerror	rpc_clnt_create
clnt_sperrno	rpc_clnt_calls
clnt_sperror	rpc_clnt_calls
clnt_tli_create	rpc_clnt_create
clnt_tp_create	rpc_clnt_create
clnt_udpcreate	rpc_soc
clnt_vc_create	rpc_clnt_create
clntraw_create	rpc_soc
clnttcp_create	rpc_soc
clntudp_bufcreate	rpc_soc
get_myaddress	rpc_soc
pmap_getmaps	rpc_soc
pmap_getport	rpc_soc
pmap_rmtcall	rpc_soc
pmap_set	rpc_soc
pmap_unset	rpc_soc
registerrpc	rpc_soc
rpc_broadcast	rpc_clnt_calls
rpc_broadcast_exp	rpc_clnt_calls
rpc_call	rpc_clnt_calls
rpc_reg	rpc_svc_calls
svc_create	rpc_svc_create
svc_destroy	rpc_svc_create
svc_dg_create	rpc_svc_create

svc_dg_enablecache	rpc_svc_calls
svc_fd_create	rpc_svc_create
svc_fds	rpc_soc
svc_freeargs	rpc_svc_reg
svc_getargs	rpc_svc_reg
svc_getcaller	rpc_soc
svc_getreq	rpc_soc
svc_getreqset	rpc_svc_calls
svc_getrpccaller	rpc_svc_calls
svc_raw_create	rpc_svc_create
svc_reg	rpc_svc_calls
svc_register	rpc_soc
svc_run	rpc_svc_reg
svc_sendreply	rpc_svc_reg
svc_tli_create	rpc_svc_create
svc_tp_create	rpc_svc_create
svc_unreg	rpc_svc_calls
svc_unregister	rpc_soc
svc_vc_create	rpc_svc_create
svcerr_auth	rpc_svc_err
svcerr_decode	rpc_svc_err
svcerr_noproc	rpc_svc_err
svcerr_noprog	rpc_svc_err
svcerr_progvers	rpc_svc_err

svcerr_systemerr	rpc_svc_err
svcerr_weakauth	rpc_svc_err
svcfd_create	rpc_soc
svcraw_create	rpc_soc
svctcp_create	rpc_soc
svcudp_bufcreate	rpc_soc
svcudp_create	rpc_soc
xdr_accepted_reply	rpc_xdr
xdr_authsys_parms	rpc_xdr
xdr_authunix_parms	rpc_soc
xdr_callhdr	rpc_xdr
xdr_callmsg	rpc_xdr
xdr_opaque_auth	rpc_xdr
xdr_rejected_reply	rpc_xdr
xdr_replymsg	rpc_xdr
xprt_register	rpc_svc_calls
xprt_unregister	rpc_svc_calls

RPC Synopsis

cl [**option** ...] **filename** ... -link [**linkoption** ...]

#include <rpc.h>

The RPC routines allow C language programs to make procedure calls on other machines across a network. First, the client sends a request to the server. On receipt of the request, the server calls a dispatch routine to perform the requested service, and then sends back a reply.

All RPC routines require the header <rpc.h>.

Using the Nettype String

Some of the high-level RPC interface routines (for example, clnt_create(), svc_create(), rpc_reg(), rpc_call()) take a **nettype** string as one of the parameters . This string defines a class of transports which can be used for a particular application.

nettype can be one of the following:

netpath Choose from the transports which have been indicated by their token names in the environment variable. If is unset or NULL, it defaults to visible. netpath is the default **nettype**.

visible Choose the transports which have the visible flag(v) set in the /etc/netconfig file.

circuit_v This is same as visible except that it chooses only the connection oriented transports (semantics tpi_cots or tpi_cots_ord) from the entries in the /etc/netconfig file.

datagram_v This is same as visible except that it chooses only the connectionless datagram transports (semantics tpi_clts) from the entries in the /etc/netconfig file.

circuit_n This is same as netpath except that it chooses only the connection oriented datagram transports (semantics tpi_cots or tpi_cots_ord).

datagram_n This is same as netpath except that it chooses only the connectionless datagram transports (semantics tpi_clts).

udp This refers to Internet UDP.

tcp This refers to Internet TCP.

If **nettype** is NULL, it defaults to netpath. The transports are tried in left to right order in the variable or in top to down order in the /etc/netconfig file.

RPC Data Structures

Some of the data structures used by the RPC package are shown below.

The AUTH Structure

```
union des_block {
        struct {
            u_int32 high;
            u_int32 low;
        } key;
        char c[8];
};
typedef union des_block des_block;
extern bool_t xdr_des_block();
/*


* Authentication info. Opaque to client.
*/
struct opaque_auth {
        enum_t  oa_flavor;      /* flavor of auth */
        caddr_t oa_base;        /* address of more
                                   auth stuff */
        u_int   oa_length;      /* not to exceed
                                   MAX_AUTH_BYTES */
};
/*
* Auth handle, interface to client side
   authenticators.
*/
typedef struct {
        struct  opaque_auth  ah_cred;
        struct  opaque_auth  ah_verf;
        union   des_block    ah_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;
} AUTH;
```

The CLIENT 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;         /* net identifier */
        char  *cl_tp;            /* device name */
} CLIENT;
```

The SVCXPRT Structure

```
enum xprt_stat {
        XPRT_DIED,
        XPRT_MOREREQS,
        XPRT_IDLE
};
/*
* Server side transport handle
*/
typedef struct {
        int  xp_fd;
                /* file descriptor for the server
                handle */
        u_short  xp_port;           /* 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 alloc. 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 identifier */
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_type
             /* transport/yp */
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;
```

The svc_reg Structure

```
struct svc_req {
      u_long  rq_prog;
                /* service program number */
      u_long  rq_vers;
                /* service protocol version */
      u_long  rq_proc;
                /* the desired procedure */
      struct opaque_auth rq_cred;
                /* raw creds from the wire */
      caddr_t  rq_clntcred;
                /* read only cooked cred */
      SVCXPRT  *rq_xprt;
                /* associated transport */
};
```

The XDR Structure

```
/*
* XDR operations.
* XDR_ENCODE causes the type to be encoded into
   the stream.
* XDR_DECODE causes the type to be extracted from
```

```
   the stream.
* XDR_FREE can be used to release the space
   allocated by an XDR_DECODE request.
*/
enum xdr_op {
      XDR_ENCODE=0,
      XDR_DECODE=1,
      XDR_FREE=2
};
/* The no. of bytes per unit of external data */
#define BYTES_PER_XDR_UNIT    (4)
#define RNDUP(x)  ((((x) + BYTES_PER_XDR_UNIT - 1)
   / BYTES_PER_XDR_UNIT) \ * BYTES_PER_XDR_UNIT)
/*
* A xdrproc_t exists for each data type which is
   to be encoded or decoded.
* The second argument to the xdrproc_t is a
   pointer to an opaque pointer.
* The opaque pointer generally points to a
   structure of the data type
* to be decoded.  If this points to 0, then the
   type routines should
* allocate dynamic storage of the appropriate size
   and return it.
* bool_t  (*xdrproc_t)(XDR *, caddr_t *);
*/
typedef  bool_t (*xdrproc_t)();
/*
* The XDR handle.
* Contains operation which is being applied to the
```

```
   stream,
* an operations vector for the particular
   implementation
*/
typedef struct {
      enum xdr_op x_op;
         /* operation; fast additional param */
      struct xdr_ops {
         bool_t  (*x_getlong)();
         /* get a long from underlying stream */
         bool_t  (*x_putlong)();
         /* put a long to the underlying stream*/
         bool_t  (*x_getbytes)();
         /* get some bytes from the underlying
            stream */
         bool_t  (*x_putbytes)();
         /* put some bytes to the underlying
            stream */
         u_int   (*x_getpostn)();
         /* returns bytes off from beginning */
         bool_t  (*x_setpostn)();
         /* lets you reposition the stream */
         long *  (*x_inline)();
         /* buf quick ptr to buffered data */
         void    (*x_destroy)();
         /* free privates of this xdr_stream */
      } *x_ops;
      caddr_t  x_public;
         /* users' data */
```

```
        caddr_t  x_private;
            /* pointer to private data */
        caddr_t  x_base;
            /* private used for position info */
        int  x_handy;
            /* extra private word */
} XDR;
```

NAME

rpc_clnt_auth, auth_destroy, authnone_create, authsys_create, authsys_create_default - library routines for client side remote procedure call authentication

DESCRIPTION

These routines are part of the RPC library that allows C language programs to make procedure calls on other machines across the network, with desired authentication.

These routines are normally called after creating the handle. The cl_auth field of the structure should be initialized by the structure returned by some of the following routines. The client's authentication information is passed to the server when the RPC call is made.

Routines

The following routines require that the header <rpc.h> be included (see rpc for the definition of the data structure).

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

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

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

auth destroy().

```
AUTH *authnone_create(void);
```

Create and return an RPC authentication handle that passes nonusable authentication information with each remote procedure call. This is the default authentication used by RPC.

```
AUTH *authsys_create(const char *host, const uid t
   uid, const gid_t gid, const int len, const gid
   t *aup gids);
```

Create and return an RPC authentication handle that contains authentication information. The parameter **host** is the name of the machine on which the information was created; **uid** is the user's user ID; **gid** is the user's current group ID; **len** and **aup gids** refer to a counted array of groups to which the user belongs.

```
AUTH *authsys_create_default(void);
```

Call authsys_create() with the appropriate parameters.

SEE ALSO

rpc, rpc_clnt_calls, rpc_clnt_create, secure_rpc

NAME

rpc_clnt_calls, clnt_call, clnt_freeres, clnt_geterr, clnt_perrno, clnt_perror, clnt_sperrno, clnt_sperror, rpc_broadcast, rpc_broadcast_exp, rpc_call - library routines for client side calls

DESCRIPTION

RPC library routines allow C language programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a request to the server. Upon receipt of the request, the server calls a dispatch routine to perform the requested service, and then sends back a reply.

The clnt_call(), rpc_call(), and rpc_broadcast() routines handle the client side of the procedure call. The remaining routines deal with error handling in the case of errors.

Some of the routines take a CLIENT handle as one of the parameters. A CLIENT handle can be created by an RPC creation routine such as clnt_create() (see rpc_clnt_create).

Routines

See rpc for the definition of the data structure.

```
#include <rpc.h>

enum clnt_stat clnt_call(CLIENT *clnt, const u
   long procnum, const xdrproc_t inproc, const
```

```
caddr_t in, const xdrproc t outproc, caddr t
out, const struct timeval tout);
```

A function macro that calls the remote procedure **proc- num** associated with the client handle, **clnt**, which is obtained with an RPC client creation routine such as clnt_create() (see rpc_clnt_create). The parameter **inproc** is the XDR function used to encode the procedure's parameters, and **outproc** is the XDR function used to decode the procedure's results; **in** is the address of the procedure's argument(s), and **out** is the address of where to place the result(s). **tout** is the time allowed for results to be returned, which is over-ridden by a default time-out set by clnt_create() or by clnt_create_vers() or by a time-out set explicitly through clnt_control(), see rpc_clnt_create.

If the remote call succeeds, the status returned is RPC_SUCCESS, otherwise an appropriate status is returned.

```
bool_t clnt_freeres(CLIENT *clnt, const xdrproc t
   outproc, caddr_t out);
```

A function macro that frees any data allocated by the RPC/XDR system when it decoded the results of an RPC call. The parameter **out** is the address of the results, and **outproc** is the XDR routine describing the results. This routine returns 1 if the results were successfully freed, and 0 otherwise.

```
void clnt_geterr(const CLIENT *clnt, struct rpc
   err *errp);
```

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

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

Print a message to standard error corresponding to the condition indicated by **stat**. A newline is appended. Normally used after a procedure call fails for a routine for which a client handle is not needed, for instance rpc_call().

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

Print a message to the standard error indicating why an RPC call failed; **clnt** is the handle used to do the call. The message is prepended with string **s** and a colon. A newline is appended. Normally used after a remote procedure call fails for a routine which requires a client handle, for instance clnt_call().

```
char *clnt_sperrno(const enum clnt_stat stat);
```

Take the same arguments as clnt_perrno(), but instead of sending a message to the standard error indicating why an RPC call failed, return a pointer to a string which contains the message.

clnt_sperrno() is normally used instead of clnt_perrno() when the program does not have a standard error (as a program running as a server quite likely does not), or if the programmer does not want the message to be output with printf(), or if a message format different than that supported by clnt_perrno() is to be used. Note: unlike clnt_sperror() and clnt_spcreaterror() (see rpc_clnt_create), clnt_sperrno() does not return pointer to static data so the result will not get overwritten on each call.

```
char *clnt_sperror(const CLIENT *clnt, const char
```

```
*s);
```

Like clnt_perror(), except that (like clnt_sperrno()) it returns a string instead of printing to standard error. However, clnt_sperror() does not append a newline at the end of the message.

Warning: returns pointer to static data that is overwritten on each call.

```
enum clnt_stat rpc_broadcast(const u_long prognum,
    const u_long versnum, const u long procnum,
    const xdrproc_t inproc, const caddr t in, const
    xdrproc_t outproc, caddr t out, const
    resultproc_t eachresult, const char *nettype);
```

Like rpc_call(), except the call message is broadcast to all the connectionless transports specified by **net- type**. If **nettype** is NULL, it defaults to "netpath" . Each time it receives a response, this routine calls eachresult(), whose form is:

bool_t eachresult(caddr_t **out**, const struct netbuf ***addr**, const struct netconfig ***netconf**);

where **out** is the same as **out** passed to rpc broadcast(), except that the remote procedure's output is decoded there; **addr** points to the address of the machine that sent the results, and **netconf** is the netconfig structure of the transport on which the remote server responded. If eachresult() returns 0, rpc_broadcast() waits for more replies; otherwise it returns with appropriate status.

Warning: broadcast file descriptors are limited in size to the maximum transfer size of that transport. For Ethernet, this value is 1500 bytes. rpc_broadcast() uses SB AUTH_SYS credentials by default (see rpc_clnt_auth).

```
enum clnt_stat rpc_broadcast_exp(const u_long
    prognum, const u_long versnum, const u long
    procnum, const xdrproc_t xargs, caddr t argsp,
```

```
    const xdrproc_t xresults, caddr t resultsp,
    const int inittime, const int waittime, const
    resultproc_t eachresult, const char *nettype);
    );
```

Like rpc_broadcast(), except that the initial timeout, **inittime** and the maximum timeout, **waittime** are specified in milliseconds.

inittime is the initial time that rpc broadcast exp() waits before resending the request. After the first resend, the re-transmission interval increases exponentially until it exceeds **waittime**.

```
enum clnt_stat rpc_call(const char *host, const u
    long prognum, const u_long versnum, const u
    long procnum, const xdrproc_t inproc, const
    char *in, const xdrproc_t outproc, char *out,
    const char *nettype);
```

Call the remote procedure associated with **prognum**, **versnum**, and **procnum** on the machine, **host**. The parameter **inproc** is used to encode the procedure's parameters, and **outproc** is used to decode the procedure's results; **in** is the address of the procedure's argument(s), and **out** is the address of where to place the result(s). **nettype** can be any of the values listed on rpc. This routine returns RPC_SUCCESS if it succeeds, or an appropriate status is returned. Use the clnt_perrno() routine to translate failure status into error messages.

Warning: rpc_call() uses the first available transport belonging to the class **nettype**, on which it can create a connection. You do not have control of timeouts or authentication using this routine.

SEE ALSO

rpc, rpc_clnt_auth, rpc_clnt_create

NAME

rpc_clnt_create, clnt_control, clnt_create, clnt_create_vers, clnt_destroy, clnt_dg_create, clnt_pcreateerror, clnt_raw_create, clnt_spcreateerror, clnt_tli_create, clnt_tp_create, clnt_vc_create, rpc_createerr - library routines for dealing with creation and manipulation of CLIENT handles

DESCRIPTION

RPC library routines allow C language programs to make procedure calls on other machines across the network. First a handle is created and then the client calls a procedure to send a request to the server. On receipt of the request, the server calls a dispatch routine to perform the requested service, and then sends a reply.

Routines

See rpc for the definition of the data structure.

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

```
bool_t clnt_control(CLIENT *clnt, const u int req,
   char *info);
```

A function macro to change or retrieve various information about a client object. **req** indicates the type of operation, and **info** is a pointer to the information. For both connectionless and connection-oriented transports, the supported values of **req** and their argument types and what they do are:

CLSET_TIMEOUT	struct timeval *set total timeout
CLGET_TIMEOUT	struct timeval *get total timeout

[Note] If you set the timeout using clnt_control(), the timeout argument passed by clnt_call() is ignored in all subsequent calls.

[Note] If you set the timeout value to 0 clnt_control() immediately returns an error (RPC_TIMEDOUT). Set timeout to 0 for batching calls.

CLGET_FD	int *	get the associated file descriptor
CLGET_SVC_ADDR		struct netbuf *get servers address
CLSET_FD_CLOSE		voidclose the file descriptor when destroying the client handle (see clnt_destroy())
CLSET_FD_NCLOSE		voiddo not close the file descriptor when destroyingthe client handle
CLGET_VERS	unsigned long	*get the RPC program's version number associated with the client handle
CLSET_VERS	unsigned long	*set the RPC program's version number associated with the client handle. This assumes that the RPC server for this new version is still listening at the address of the previous version.
CLGET_XID	unsigned long	*get the XID of the previous remote procedure call
CLSET_XID	unsigned long	*set the XID of the next remote procedure call

The following operations are valid for connectionless transports only:

CLSET_RETRY_TIMEOUT struct timeval

* set the retry timeout

CLGET_RETRY_TIMEOUT struct timeval

* get the retry timeout

The retry timeout is the time that RPC waits for the server to reply before retransmitting the request.

clnt_control() returns on success and on failure.

```
CLIENT *clnt_create(const char *host, const u long
   prognum, const u_long versnum, const char
   *nettype);
```

Generic client creation routine for program **prognum** and version **versnum**. **host** identifies the name of the remote host where the server is located. **nettype** indicates the class of transport protocol to use. The transports are tried in left to right order in variable or in top to bottom order in the netconfig database.

clnt_create() tries all the transports of the **nettype** class available from the environment variable and the netconfig database, and chooses the first successful one. A default timeout is set and can be modified using clnt_control(). This routine returns if it fails. The clnt_pcreaterror() routine can be used to print the reason for failure.

[Note] clnt_create() returns a valid client handle even if the particular version number supplied to clnt_create() is not registered with the rpcbind service. This mismatch will be discovered by a clnt_call later (see rpc_clnt_calls).

```
CLIENT *clnt_create_vers(const char *host, const u
    long prognum, u_long *vers outp, const u long
    vers low, const u_long vers high, char
    *nettype);
```

Generic client creation routine which is similar to clnt_create() but which also checks for the version availability. **host** identifies the name of the remote host where the server is located. **nettype** indicates the class transport protocols to be used. If the routine is successful it returns a client handle created for the highest version between **vers low** and **vers high** that is supported by the server. **vers outp** is set to this value. That is, after a successful return **vers low** <= ***vers outp** <= **vers high**. If no version between **vers low** and **vers high** is supported by the server then the routine fails and returns NULL. A default timeout is set and can be modified using clnt_control(). This routine returns if it fails. The clnt_pcreaterror() routine can be used to print the reason for failure.

[Note] clnt_create() returns a valid client handle even if the particular version number supplied to clnt_create() is not registered with the rpcbind service. This mismatch will be discovered by a clnt_call later (see rpc_clnt_calls). However, clnt_create_vers() does this for you and returns a valid handle only if a version within the range supplied is supported by the server.

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

A function macro that destroys the client's RPC handle. Destruction usually involves deallocation of private data structures, including **clnt** itself. Use of **clnt** is undefined after calling clnt_destroy(). If the RPC library opened the associated file descriptor, or was set using clnt_control(), the file descriptor will be closed.

The caller should call auth_destroy(**clnt->cl auth**) (before calling clnt_destroy()) to destroy the associated AUTH structure (see rpc_clnt_auth).

```
CLIENT *clnt_dg_create(const int fildes, const
   struct netbuf *svcaddr, const u long prognum,
   const u_long versnum, const u int sendsz, const
   u_int recvsz);
```

This routine creates an RPC client for the remote program **prognum** and version **versnum**; the client uses a connectionless transport. The remote program is located at address **svcaddr**. The parameter **fildes** is an open and bound file descriptor. This routine will resend the call message in intervals of 15 seconds until a response is received or until the call times out. The total time for the call to time out is specified by clnt_call() (see clnt_call() in rpc_clnt_calls). The retry time out and the total time out periods can be changed using clnt_control(). The user may set the size of the send and receive buffers with the parameters **sendsz** and **recvsz**; values of 0 choose suitable defaults. This routine returns if it fails.

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

Print a message to standard error indicating why a client RPC handle could not be created. The message is prepended with the string **s** and a colon, and appended with a newline.

```
CLIENT *clnt_raw_create(const u_long prognum,
   const u_long versnum);
```

This routine creates an RPC client handle for the remote program **prognum** and version **versnum**. The transport used to pass messages to the service is a buffer within the process's address space, so the corresponding RPC server should live in the same address space; (see svc_raw_create() in rpc_svc_create). This allows simulation of RPC and measurement of RPC overheads, such as round trip times, without any kernel or networking interference. This routine returns if it fails. clnt_raw_create() should be called after svc_raw_create().

```
char *clnt_spcreateerror(const char *s);
```

Like clnt_pcreateerror(), except that it returns a string instead of printing to the standard error. A newline is not appended to the message in this case.

Warning: Returns a pointer to static data that is overwritten on each call.

```
CLIENT *clnt_tli_create(const int fildes, const
    struct netconfig *netconf, const struct netbuf
    *svcaddr, const long prognum, const u_long
    versnum, const u int sendsz, const u_int
    recvsz);
```

This routine creates an RPC client handle for the remote program **prognum** and version **versnum**. The remote program is located at address **svcaddr**. If **svcaddr** is and it is connection-oriented, it is assumed that the file descriptor is connected. For connectionless transports, if **svcaddr** is NULL, error is set. **fildes** is a file descriptor which may be open, bound and connected. If it is RPC_ANYFD, it opens a file descriptor on the transport specified by **netconf**. If **fildes** is and **netconf** is NULL, a error is set. If **fildes** is unbound, then it will attempt to bind the descriptor. The user may specify the size of the buffers with the parameters **sendsz** and **recvsz**; values of 0 choose suitable defaults.

Depending upon the type of the transport (connection-oriented or connectionless), clnt_tli_create() calls appropriate client creation routines. This routine returns if it fails. The clnt_pcreaterror() routine can be used to print the reason for failure. The remote rpcbind service is not consulted for the address of the remote service.

```
CLIENT *clnt_tp_create(const char *host, const u
    long prognum, const u_long versnum, const
    struct netconfig *netconf);
```

Like clnt_create() except clnt_tp_create() tries only one transport specified through **netconf**.

clnt_tp_create() creates a client handle for the program **prognum**, the version **versnum**, and for the transport specified by **netconf**. Default options are set, which can be changed using clnt_control() calls. The remote rpcbind service on the host **host** is consulted for the address of the remote service. This routine returns if it fails. The clnt_pcreaterror() routine can be used to print the reason for failure.

```
CLIENT *clnt_vc_create(const int fildes, const
   struct netbuf *svcaddr, const u long prognum,
   const u_long versnum, const u int sendsz, const
   u_int recvsz);
```

This routine creates an RPC client for the remote program **prognum** and version **versnum**; the client uses a connection-oriented transport. The remote program is located at address **svcaddr**. The parameter **fildes** is an open and bound file descriptor. The user may specify the size of the send and receive buffers with the parameters **sendsz** and **recvsz**; values of 0 choose suitable defaults. This routine returns if it fails.

The address **svcaddr** should not be and should point to the actual address of the remote program. clnt_vc_create() does not consult the remote rpcbind service for this information.

```
struct rpc_createerr rpc_createerr;
```

A global variable whose value is set by any RPC client handle creation routine that fails. It is used by the routine clnt_pcreateerror() to print the reason for the failure.

SEE ALSO

rpc, rpc_clnt_auth, rpc_clnt_calls

NAME

rpc_svc_calls, svc_dg_enablecache, svc_exit, svc_fdset, svc_freeargs, svc_getargs, svc_getreq_common, svc_getreq_poll, svc_getreqset, svc_getrpccaller, svc_pollset, svc_run, svc_sendreply - library routines for RPC servers

DESCRIPTION

These routines are part of the RPC library which allows C language programs to make procedure calls on other machines across the network.

These routines are associated with the server side of the RPC mechanism. Some of them are called by the server side dispatch function, while others (such as svc_run()) are called when the server is initiated.

Routines

```
#include <rpc.h>

int svc_dg_enablecache(SVCXPRT *xprt, const
   unsigned long cache size);
```

This function allocates a duplicate request cache for the service endpoint **xprt**, large enough to hold **cache size** entries. Once enabled, there is no way to disable caching. This routine returns **0** if space necessary for a cache of the given size was successfully allocated, and **1** otherwise.

```
void svc_exit(void);
```

This function when called by any of the RPC server procedure or otherwise, causes svc_run() to return.

```
fd_set svc_fdset;
```

A global variable reflecting the RPC server's read file descriptor bit mask; it is suitable as a parameter to the select(3C) system call. This is only of interest if service implementors do not call svc_run(), but rather do their own asynchronous event processing. This variable is read-only (do not pass its address to select(3C)!), yet it may change after calls to svc_getreqset() or any creation routines.

```
bool_t svc_freeargs(const SVCXPRT *xprt, const
   xdrproc t inproc, caddr_t in);
```

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

```
bool_t svc_getargs(const SVCXPRT *xprt, const
   xdrproc t inproc, caddr_t in);
```

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

```
void svc_getreq_common(const int fd);
```

This routine is called to handle a request on the given file descriptor.

```
void svc_getreq_poll(struct pollfd *pfdp, const
    int pollretval);
```

This routine is only of interest if a service implementor does not call svc_run(), but instead implements custom asynchronous event processing. It is called when poll(2) has determined that an RPC request has arrived on some RPC file descriptors; **pollretval** is the return value from poll() and **pfdp** is the array of **pollfd** structures on which the poll() was done. It is assumed to be an array large enough to contain the maximal number of descriptors allowed.

```
void svc_getreqset(fd_set *rdfds);
```

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

```
struct netbuf *svc_getrpccaller(const SVCXPRT
    *xprt);
```

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

```
void svc_run(void);
```

This routine never returns. It waits for RPC requests to arrive, and calls the appropriate service procedure using svc_getreq_poll() when one arrives. This procedure is usually waiting for the poll() library call to return.

```
bool_t svc_sendreply(const SVCXPRT *xprt, const
   xdrproc_t outproc, const caddr t *out);
```

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

SEE ALSO

rpc, rpc_svc_create, rpc_svc_err

NAME

rpc_svc_create, svc_create, svc_destroy, svc_dg_create, svc_fd_create, svc_raw_create, svc_tli_create, svc_tp_create, svc_vc_create - library routines for the creation of server handles

DESCRIPTION

These routines are part of the RPC library which allows C language programs to make procedure calls on servers across the network. These routines deal with the creation of service handles. Once the handle is created, the server can be invoked by calling svc_run().

Routines

See rpc for the definition of the data structure.

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

```
int svc_create(const void (*dispatch)(struct
   svc_req * rqstp , SVCXPRT * transp ), const
   u_long prognum, const u_long versnum, const
   char *nettype);
```

svc_create() creates server handles for all the transports belonging to the class **nettype**.

nettype defines a class of transports which can be used for a particular application. The transports are tried in left to right order in variable or in top to bottom order in the netconfig database. If

nettype is NULL, it defaults to netpath.

svc_create() registers itself with the rpcbind service. **dispatch** is called when there is a remote procedure call for the given **prognum** and **vers- num**; this requires calling svc run() (see svc run() in rpc_svc_reg). If svc_create() succeeds, it returns the number of server handles it created, otherwise it returns 0 and an error message is logged.

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

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

```
SVCXPRT *svc_dg_create(const int fildes, const u
   int sendsz, const u_int recvsz);
```

This routine creates a connectionless RPC service handle, and returns a pointer to it. This routine returns if it fails, and an error message is logged. **sendsz** and **recvsz** are parameters used to specify the size of the buffers. If they are 0, suitable defaults are chosen. The file descriptor **fildes** should be open and bound. The server is not registered with rpcbind.

Warning: Since connectionless-based RPC messages can only hold limited amount of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

```
SVCXPRT *svc_fd_create(const int fildes, const u
   int sendsz, const u_int recvsz);
```

This routine creates a service on top of an open and bound file descriptor, and returns the handle to it. Typically, this descriptor is a

connected file descriptor for a connection-oriented transport. **sendsz** and **recvsz** indicate sizes for the send and receive buffers. If they are 0, reasonable defaults are chosen. This routine returns if it fails, and an error message is logged.

```
SVCXPRT *svc_raw_create(void);
```

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

[Note] svc_run() should not be called when the raw interface is being used.

```
SVCXPRT *svc_tli_create(const int fildes, const
   struct netconfig *netconf, const struct t_bind
   *bindaddr, const u int sendsz, const u_int
   recvsz);
```

This routine creates an RPC server handle, and returns a pointer to it. **fildes** is the file descriptor on which the service is listening. If **fildes** is RPC ANYFD, it opens a file descriptor on the transport specified by **netconf**. If the file descriptor is unbound and **bindaddr** is non-null **fildes** is bound to the address specified by **bindaddr**, otherwise **fildes** is bound to a default address chosen by the transport. In the case where the default address is chosen, the number of outstanding connect requests is set to 8 for connectionoriented transports. The user may specify the size of the send and receive buffers with the parameters **sendsz** and **recvsz** ; values of 0 choose suitable defaults. This routine returns if it fails, and an error message is logged. The server is not registered with the rpcbind service.

```
SVCPRT *svc_tp_create(const void (*dispatch)
    (const struct svc_reg *, const SVCXPRT *),
    const u_long prognum, const u long versnum,
    const struct netconfig *netconf);
```

svc_tp_create() creates a server handle for the network specified by **netconf**, and registers itself with the rpcbind service. **dispatch** is called when there is a remote procedure call for the given **prognum** and **versnum**; this requires calling svc run(). svc tp create() returns the service handle if it succeeds, otherwise a is returned and an error message is logged.

```
SVCXPRT *svc_vc_create(const int fildes, const u
    int sendsz, const u_int recvsz);
```

This routine creates a connection-oriented RPC service and returns a pointer to it. This routine returns if it fails, and an error message is logged. The users may specify the size of the send and receive buffers with the parameters **sendsz** and **recvsz**; values of 0 choose suitable defaults. The file descriptor **fildes** should be open and bound. The server is not registered with the rpcbind service.

SEE ALSO

rpc, rpc_svc_calls, rpc_svc_err, rpc_svc_reg

NAME

rpc_svc_err, svcerr_auth, svcerr_decode, svcerr_noproc, svcerr_noprog, svcerr_progvers, svcerr_systemerr, svcerr_weakauth - library routines for server side remote procedure call errors

DESCRIPTION

These routines are part of the RPC library which allows C language programs to make procedure calls on other machines across the network.

These routines can be called by the server side dispatch function if there is any error in the transaction with the client.

Routines

See rpc for the definition of the data structure.

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

```
void svcerr_auth(const SVCXPRT *xprt, const enum
   auth stat why);
```

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

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

Called by a service dispatch routine that cannot successfully decode the remote parameters (see svc_getargs() in rpc_svc_reg).

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

Called by a service dispatch routine that does not implement the procedure number that the caller requests.

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

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

```
void svcerr_progvers(const SVCXPRT *xprt, u_long
   low vers, u_long high vers);
```

Called when the desired version of a program is not registered with the RPC package. **low vers** is the lowest version number, and **high vers** is the highest version number. Service implementors usually do not need this routine.

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

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

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

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

AUTH_TOOWEAK).

SEE ALSO

rpc, rpc_svc_calls, rpc_svc_create, rpc_svc_reg

NAME

rpc_svc_reg, rpc_reg, svc_reg, svc_unreg, svc_auth_reg, xprt_register, xprt_unregister - library routines for registering servers

DESCRIPTION

These routines are a part of the RPC library which allows the RPC servers to register themselves with rpcbind() and associate the given program and version number with the dispatch function. When the RPC server receives a RPC request, the library invokes the dispatch routine with the appropriate arguments.

Routines

See rpc for the definition of the data structure.

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

```
bool_t rpc_reg(const u_long prognum, const u long
   versnum, const u_long procnum, const char
   *(*procname), const xdrproc_t inproc, const
   xdrproc_t outproc, const char *nettype);
```

Register program **prognum**, procedure **procname**, and version **versnum** with the RPC service package. If a request arrives for program **prognum**, version **versnum**, and procedure **procnum**, **procname** is called with a pointer to its parameter(s); **procname** should return a pointer to its static **result**(**s**); **inproc** is the XDR

function used to decode the parameters while **outproc** is the XDR function used to encode the results. Procedures are registered on all available transports of the class **nettype**. See rpc. This routine returns 0 if the registration succeeded, -1 otherwise.

```
int svc_reg(const SVCXPRT *xprt, const u long
   prognum, const u_long versnum, const void
   (*dispatch), const struct netconfig *netconf);
```

Associates **prognum** and **versnum** with the service dispatch procedure, **dispatch**. If **netconf** is NULL, the service is not registered with the **rpcbind** service. If **netconf** is non-zero, then a mapping of the triple [**prognum**, **versnum**, **netconf->nc netid**] to **xprt->xp ltaddr** is established with the local rpcbind service.

The svc_reg() routine returns 1 if it succeeds, and 0 otherwise.

```
void svc_unreg(const u_long prognum, const u long
   versnum);
```

Remove from the rpcbind service, all mappings of the triple [**prognum**, **versnum**, **all-transports**] to network address and all mappings within the RPC service package of the double [**prognum**, **versnum**] to dispatch routines.

```
int svc_auth_reg(const int cred flavor, const enum
   auth stat (*handler));
```

Registers the service authentication routine **handler** with the dispatch mechanism so that it can be invoked to authenticate RPC requests received with authentication type **cred flavor**. This interface allows developers to add new authentication types to their RPC applications without needing to modify the libraries. Service implementors usually do not need this routine.

Typical service application would call svc_auth_reg() after registering the service and prior to calling svc_run(). When needed to process an RPC credential of type **cred flavor**, the **handler** procedure will be called with two parameters (struct svc_req ***rqst**, struct rpc_msg ***msg**) and is expected to return a valid enum auth_stat value. There is no provision to change or delete an authentication handler once registered.

The svc_auth_reg() routine returns 0 if the registration is successful, 1 if **cred flavor** already has an authentication handler registered for it, and -1 otherwise.

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

After RPC service transport handle **xprt** is created, it is registered with the RPC service package. This routine modifies the global variable svc_fdset (see rpc_svc_calls). Service implementors usually do not need this routine.

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

Before an RPC service transport handle **xprt** is destroyed, it unregisters itself with the RPC service package. This routine modifies the global variable svc_fdset (see rpc_svc_calls). Service implementors usually do not need this routine.

SEE ALSO

rpc, rpc_svc_calls, rpc_svc_create, rpc_svc_err, rpcbind

NAME

rpc_xdr, xdr_accepted_reply, xdr_authsys_parms, xdr_callhdr, xdr_callmsg, xdr_opaque_auth, xdr_rejected_reply, xdr_replymsg - XDR library routines for remote procedure calls

DESCRIPTION

These routines are used for describing the RPC messages in XDR language. They should normally be used by those who do not want to use the RPC package directly. These routines return TRUE if they succeed, FALSE otherwise.

Routines

See rpc for the definition of the data structure.

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

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

Used to translate between RPC reply messages and their external representation. It includes the status of the RPC call in the XDR language format. In the case of success, it also includes the call results.

```
bool_t xdr_authsys_parms(XDR *xdrs, struct authsys
```

```
    parms *aupp);
```

Used for describing UNIX operating system credentials. It includes machine-name, uid, gid list, etc.

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

Used for describing RPC call header messages. It encodes the static part of the call message header in the XDR language format. It includes information such as transaction ID, RPC version number, program and version number.

```
bool_t xdr_callmsg(XDR *xdrs, struct rpc msg
    *cmsg);
```

Used for describing RPC call messages. This includes all the RPC call information such as transaction ID, RPC version number, program number, version number, authentication information, etc. This is normally used by servers to determine information about the client RPC call.

```
bool_t xdr_opaque_auth(XDR *xdrs, struct opaque
    auth *ap);
```

Used for describing RPC opaque authentication information messages.

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

Used for describing RPC reply messages. It encodes the rejected RPC message in the XDR language format. The message could be rejected either because of version number mis-match or because of authentication errors.

```
bool_t xdr_replymsg(XDR *xdrs, const struct rpc
   msg *rmsg);
```

Used for describing RPC reply messages. It translates between the RPC reply message and its external representation. This reply could be either an acceptance, rejection or NULL.

SEE ALSO

rpc, xdr

NAME

rpcbind, rpcb_getmaps, rpcb_getaddr, rpcb_gettime, rpcb_rmtcall, rpcb_set, rpcb_unset - library routines for RPC bind service

DESCRIPTION

These routines allow client C programs to make procedure calls to the RPC binder service. rpcbind maintains a list of mappings between programs and their universal addresses.

Routines

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

```
struct rpcblist *rpcb_getmaps(const struct
   netconfig *netconf, const char *host);
```

An interface to the rpcbind service, which returns a list of the current RPC program-to-address mappings on **host**. It uses the transport specified through **netconf** to contact the remote rpcbind service on **host**. This routine will return NULL, if the remote rpcbind could not be contacted.

```
bool_t rpcb_getaddr(const u_long prognum, const u
   long versnum, const struct netconfig *netconf,
   struct netbuf *svcaddr, const char *host);
```

An interface to the rpcbind service, which finds the address of the service on **host** that is registered with program number **prognum**,

version **versnum**, and speaks the transport protocol associated with **netconf**. The address found is returned in **svcaddr**. **svcaddr** should be preallocated. This routine returns if it succeeds. A return value of means that the mapping does not exist or that the RPC system failed to contact the remote rpcbind service. In the latter case, the global variable rpc_createerr (see rpc_clnt_create) contains the RPC status.

```
bool_t rpcb_gettime(const char *host, time t
    *timep);
```

This routine returns the time on **host** in **timep**. If **host** is NULL, rpcb gettime() returns the time on its own machine. This routine returns if it succeeds, if it fails. rpcb_gettime() can be used to synchronize the time between the client and the remote server. This routine is particularly useful for secure RPC.

```
enum clnt_stat rpcb_rmtcall(const struct netconfig
    *netconf, const char *host, const u long
    prognum, const u_long versnum, const u long
    procnum, const xdrproc_t inproc, const caddr t
    in, const xdrproc_t outproc, caddr t out, const
    struct timeval tout, struct netbuf  *svcaddr);
```

An interface to the rpcbind service, which instructs rpcbind on **host** to make an RPC call on your behalf to a procedure on that host. The netconfig structure should correspond to a connectionless transport. The parameter ***svcaddr** will be modified to the server's address if the procedure succeeds (see rpc_call() and clnt_call() in rpc_clnt_calls for the definitions of other parameters).

This procedure should normally be used for a ping and nothing else. This routine allows programs to do lookup and call, all in one step.

[Note] Even if the server is not running rpcbind does not return any error messages to the caller. In such a case, the caller times out.

[Note] rpcb_rmtcall() is only available for connectionless transports.

```
bool_t rpcb_set(const u_long prognum, const u long
    versnum, const struct netconfig *netconf, const
    struct netbuf *svcaddr);
```

An interface to the rpcbind service, which establishes a mapping between the triple [**prognum**, **versnum**, **netconf->nc netid**] and **svcaddr** on the machine's rpcbind service. The value of **nc netid** must correspond to a network identifier that is defined by the netconfig database. This routine returns if it succeeds, otherwise. (See also svc_reg() in rpc_svc_calls). If there already exists such an entry with rpcbind, rpcb_set() will fail.

```
bool_t rpcb_unset(const u_long prognum, const u
    long versnum, const struct netconfig *netconf);
```

An interface to the rpcbind service, which destroys the mapping between the triple [**prognum**, **versnum**, **netconf->nc netid**] and the address on the machine's rpcbind service. If **netconf** is NULL, rpcb unset() destroys all mapping between the triple [**prognum**, **versnum**, **all-transports**] and the addresses on the machine's rpcbind service. This routine returns if it succeeds, otherwise. Only the owner of the service or the super-user can destroy the mapping. (See also svc_unreg() in rpc_svc_calls).

SEE ALSO

rpcbind(1M), rpcinfo(1M), rpc_clnt_calls, rpc_svc_calls

xdr xdr

NAME

xdr - library routines for external data representation

DESCRIPTION

XDR routines allow C programmers to describe arbitrary data structures in a machine-independent fashion. Data for remote procedure calls (RPC) are transmitted using these routines.

Index to Routines

The following table lists XDR routines and the manual reference pages on which they are described:

XDR Routine	Manual Reference Page
xdr_array	xdr_complex
xdr_bool	xdr_simple
xdr_bytes	xdr_complex
xdr_char	xdr_simple
xdr_destroy	xdr_create
xdr_double	xdr_simple
xdr_enum	xdr_simple
xdr_float	xdr_simple
xdr_free	xdr_simple
xdr_getpos	xdr_admin

xdr_hyper	xdr_simple
xdr_inline	xdr_admin
xdr_int	xdr_simple
xdr_long	xdr_simple
xdr_longlong_t	xdr_simple
xdr_opaque	xdr_complex
xdr_pointer	xdr_complex
xdr_quadruple	xdr_simple
xdr_reference	xdr_complex
xdr_setpos	xdr_admin
xdr_short	xdr_simple
xdr_string	xdr_complex
xdr_u_char	xdr_simple
xdr_u_hyper	xdr_simple
xdr_u_int	xdr_simple
xdr_u_long	xdr_simple
xdr_u_longlong_t	xdr_simple
xdr_u_short	xdr_simple
xdr_union	xdr_complex
xdr_vector	xdr_complex
xdr_void	xdr_simple
xdr_wrapstring	xdr_complex
xdrmem_create	xdr_create
xdrrec_create	xdr_create

xdrrec_endofrecord	xdr_admin
xdrrec_eof	xdr_admin
xdrrec_skiprecord	xdr_admin
xdrstdio_create	xdr_create

SEE ALSO

rpc, xdr_admin, xdr_complex, xdr_create, xdr_simple

NAME

xdr_admin, xdr_getpos, xdr_inline, xdrrec_endofrecord, xdrrec_eof, xdrrec_skiprecord, xdr_setpos, xdr_sizeof library routines for external data representation

DESCRIPTION

XDR library routines allow C programmers to describe arbitrary data structures in a machine-independent fashion. Protocols such as remote procedure calls (RPC) use these routines to describe the format of the data.

These routines deal specifically with the management of the XDR stream.

Routines

See rpc for the definition of the data structure. Note that any buffers passed to the XDR routines must be properly aligned. It is suggested that malloc(3C) be used to allocate these buffers or that the programmer insure that the buffer address is divisible evenly by four.

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

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

A macro that invokes the get-position routine associated with the XDR stream, **xdrs**. The routine returns an unsigned integer, which

indicates the position of the XDR byte stream. A desirable feature of XDR streams is that simple arithmetic works with this number, although the XDR stream instances need not guarantee this. Therefore, applications written for portability should not depend on this feature.

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

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

Warning: xdr_inline() may return (0) if it cannot allocate a contiguous piece of a buffer. Therefore the behavior may vary among stream instances; it exists for the sake of efficiency, and applications written for portability should not depend on this feature.

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

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

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

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

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

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

```
bool_t xdr_setpos(XDR *xdrs, const u int pos);
```

A macro that invokes the set position routine associated with the XDR stream **xdrs**. The parameter **pos** is a position value obtained from xdr_getpos(). This routine returns if the XDR stream was repositioned, and otherwise.

Warning: It is difficult to reposition some types of XDR streams, so this routine may fail with one type of stream and succeed with another. Therefore, applications written for portability should not depend on this feature.

```
unsigned long xdr_sizeof(xdrproc_t func, void
   *data);
```

This routine returns the number of bytes required to encode **data** using the XDR filter function **func**, excluding potential overhead such as RPC headers or record markers. O is returned on error. This information might be used to select between transport protocols, or to determine the buffer size for various lower levels of RPC client and server creation routines, or to allocate storage when XDR is used outside of the RPC subsystem.

SEE ALSO

malloc(3C), rpc, xdr_complex, xdr_create, xdr_simple

NAME

xdr_complex, xdr_array, xdr_bytes, xdr_opaque, xdr_pointer, xdr_reference, xdr_string, xdr_union, xdr_vector, xdr_wrapstring - library routines for external data representation

DESCRIPTION

XDR library routines allow C programmers to describe complex data structures in a machine-independent fashion. Protocols such as remote procedure calls (RPC) use these routines to describe the format of the data. These routines are the XDR library routines for complex data structures. They require the creation of XDR stream (see xdr_create).

Routines

See rpc for the definition of the data structure. Note that any buffers passed to the XDR routines must be properly aligned. It is suggested that malloc() be used to allocate these buffers or that the programmer insure that the buffer address is divisible evenly by four.

```
#include <rpc.h>

bool_t xdr_array(XDR *xdrs, caddr t *arrp, u int
   *sizep, const u_int maxsize, const u int
```

```
    elsize, const xdrproc_t elproc);
```

xdr_array() translates between variable-length arrays and their corresponding external representations. The parameter **arrp** is the address of the pointer to the array, while **sizep** is the address of the element count of the array; this element count cannot exceed **maxsize**. The parameter **elsize** is the size of each of the array's elements, and **elproc** is an XDR routine that translates between the array elements' C form and their external representation. If ***aarp** is null when decoding, xdr_array() allocates memory and ***aarp** points to it. This routine returns if it succeeds, otherwise.

```
bool_t xdr_bytes(XDR *xdrs, char **sp, u int
    *sizep, const u_int maxsize);
```

xdr_bytes() translates between counted byte strings and their external representations. The parameter **sp** is the address of the string pointer. The length of the string is located at address **sizep**; strings cannot be longer than **maxsize**. If ***sp** is null when decoding, xdr_bytes() allocates memory and ***sp** points to it. This routine returns if it succeeds, otherwise.

```
bool_t xdr_opaque(XDR *xdrs, caddr t cp, const u
    int cnt);
```

xdr_opaque() translates between fixed size opaque data and its external representation. The parameter **cp** is the address of the opaque object, and **cnt** is its size in bytes. This routine returns if it succeeds, otherwise.

```
bool_t xdr_pointer(XDR *xdrs, char **objpp, u int
    objsize, const xdrproc_t xdrobj);
```

Like xdr_reference() except that it serializes pointers, whereas xdr_reference() does not. Thus, xdr_pointer() can represent

recursive data structures, such as binary trees or linked lists. If ***objpp** is null when decoding, xdr_pointer() allocates memory and ***objpp** points to it.

```
bool_t xdr_reference(XDR *xdrs, caddr t *pp, u int
    size, const xdrproc_t proc);
```

xdr_reference() provides pointer chasing within structures. The parameter **pp** is the address of the pointer; **size** is the sizeof the structure that ***pp** points to; and **proc** is an XDR procedure that translates the structure between its C form and its external representation. If ***pp** is null when decoding, xdr reference() allocates memory and ***pp** points to it. This routine returns 1 if it succeeds, 0 otherwise.

Warning: This routine does not understand pointers. Use xdr_pointer() instead.

```
bool_t xdr_string(XDR *xdrs, char **sp, const u
    int maxsize);
```

xdr_string() translates between C strings and their corresponding external representations. Strings cannot be longer than **maxsize**. Note: **sp** is the address of the string's pointer. If ***sp** is null when decoding, xdr_string() allocates memory and ***sp** points to it. This routine returns Note: xdr_string() can be used to send an empty string (""), but not a NULL string. if it succeeds, otherwise.

```
bool_t xdr_union(XDR *xdrs, enum t *dscmp, char
    *unp, const struct xdr_discrim *choices, const
    xdrproc_t (*defaultarm);
```

xdr_union() translates between a discriminated C union and its corresponding external representation. It first translates the discriminant of the union located at **dscmp**. This discriminant is always an enum t. Next the union located at **unp** is translated. The

parameter **choices** is a pointer to an array of xdr discrim structures. Each structure contains an ordered pair of [**value**, **proc**]. If the union's discriminant is equal to the associated **value**, then the **proc** is called to translate the union. The end of the xdr_discrim structure array is denoted by a routine of value NULL. If the discriminant is not found in the **choices** array, then the **defaultarm** procedure is called (if it is not NULL). Returns if it succeeds, otherwise.

```
bool_t xdr_vector(XDR *xdrs, char *arrp, const u
   int size, const u_int elsize, const xdrproc t
   elproc);
```

xdr_vector() translates between fixed-length arrays and their corresponding external representations. The parameter **arrp** is the address of the pointer to the array, while **size** is the element count of the array. The parameter **elsize** is the sizeof each of the array's elements, and **elproc** is an XDR routine that translates between the array elements' C form and their external representation. This routine returns if it succeeds, otherwise.

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

A routine that calls xdr_string(**xdrs**, **sp**, **maxuint**); where **maxuint** is the maximum value of an unsigned integer.

Many routines, such as xdr_array(), xdr_pointer(), and xdr_vector() take a function pointer of type xdrproc_t(), which takes two arguments. xdr_string(), one of the most frequently used routines, requires three arguments, while xdr_wrapstring() only requires two. For these routines, xdr_wrapstring() is desirable. This routine returns if it succeeds, otherwise.

SEE ALSO

rpc, xdr_admin, xdr_create, xdr_simple

NAME

xdr_create, xdr_destroy, xdrmem_create, xdrrec_create, xdrstdio_create - library routines for external data representation stream creation

DESCRIPTION

XDR library routines allow C programmers to describe arbitrary data structures in a machine-independent fashion. Protocols such as remote procedure calls (RPC) use these routines to describe the format of the data.

These routines deal with the creation of XDR streams. XDR streams have to be created before any data can be translated into XDR format.

Routines

See rpc for the definition of the XDR, CLIENT, and data structures. Note that any buffers passed to the XDR routines must be properly aligned. It is suggested that malloc() be used to allocate these buffers or that the programmer insure that the buffer address is divisible evenly by four.

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

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

A macro that invokes the destroy routine associated with the XDR stream, **xdrs**. Destruction usually involves freeing private data

structures associated with the stream. Using **xdrs** after invoking xdr_destroy() is undefined.

```
void xdrmem_create(XDR *xdrs, const caddr t addr,
    const u_int size, const enum xdr op op);
```

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

```
void xdrrec_create(XDR *xdrs, const u int sendsz,
    const u_int recvsz, const caddr t handle, const
    int (*readit)(const void *read handle, char
    *buf, const int len), const int (*writeit)
    (const void *write handle, const char *buf,
    const int len));
```

This routine initializes the read-oriented XDR stream object pointed to by **xdrs**. The stream's data is written to a buffer of size **sendsz**; a value of 0 indicates the system should use a suitable default. The stream's data is read from a buffer of size **recvsz**; it too can be set to a suitable default by passing a 0 value.

When a stream's output buffer is full, **writeit** is called. Similarly, when a stream's input buffer is empty, **readit** is called. The behavior of these two routines is similar to the system calls read() and write(), except that an appropriate handle (**read handle** or **write handle**) is passed to the former routines as the first parameter instead of a file descriptor. Note: the XDR stream's **op** field must be set by the caller.

Warning: This XDR stream implements an intermediate record stream. Therefore there are additional bytes in the stream to provide record boundary information.

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

This routine initializes the XDR stream object pointed to by **xdrs**. The XDR stream data is written to, or read from, the standard I/O stream **file**. The parameter **op** determines the direction of the XDR stream (either XDR_ENCODE, XDR_DECODE, or XDR_FREE).

Warning: The destroy routine associated with such XDR streams calls fflush() on the **file** stream, but never fclose().

Failure of any of these functions can be detected by first initializing the **x ops** field in the XDR structure (**xdrs->x ops**) to NULL before calling the xdr* create() function. After the return from the xdr*_create() function, if the **x ops** field is still NULL , the call has failed. If the **x ops** field contains some other value, the call can be assumed to have succeeded.

SEE ALSO

rpc, xdr_admin, xdr_complex, xdr_simple

NAME

xdr_simple, xdr_bool, xdr_char, xdr_double, xdr_enum, xdr_float, xdr_free, xdr_hyper, xdr_int, xdr_long, xdr_longlong_t, xdr_quadruple, xdr_short, xdr_u_char, xdr_u_hyper, xdr_u_int, xdr_u_long, xdr_u_longlong_t, xdr_u_short, xdr_void - library routines for external data representation

DESCRIPTION

XDR library routines allow C programmers to describe simple data structures in a machine-independent fashion. Protocols such as remote procedure calls (RPC) use these routines to describe the format of the data.

These routines require the creation of XDR streams (see xdr_create).

Routines

See rpc for the definition of the data structure. Note that any buffers passed to the XDR routines must be properly aligned. It is suggested that malloc() be used to allocate these buffers or that the programmer insure that the buffer address is divisible evenly by four.

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

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

xdr_bool() translates between booleans (C integers) and their external representations. When encoding data, this filter produces values of either 1 or 0. This routine returns if it succeeds, otherwise.

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

xdr_char() translates between C characters and their external representations. This routine returns if it succeeds, otherwise. Note: encoded characters are not packed, and occupy 4 bytes each. For arrays of characters, it is worthwhile to consider xdr_bytes(), xdr_opaque(), or xdr_string() (see xdr_complex).

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

xdr_double() translates between C double precision numbers and their external representations. This routine returns if it succeeds, otherwise.

```
bool_t xdr_enum(XDR *xdrs, enum t *ep);
```

xdr_enum() translates between C enums (actually integers) and their external representations. This routine returns if it succeeds, otherwise.

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

xdr_float() translates between C floats and their external representations. This routine returns if it succeeds, otherwise.

```
void xdr_free(xdrproc_t proc, char *objp);
```

Generic freeing routine. The first argument is the XDR routine for the object being freed. The second argument is a pointer to the object itself. Note: the pointer passed to this routine is not freed, but what it points to is freed (recursively, depending on the XDR routine).

```
bool_t xdr_hyper(XDR *xdrs, longlong t *llp);
```

xdr_hyper() translates between ANSI C long long integers and their external representations. This routine returns if it succeeds, otherwise.

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

xdr_int() translates between C integers and their external representations. This routine returns if it succeeds, otherwise.

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

xdr_long() translates between C long integers and their external representations. This routine returns if it succeeds, otherwise.

```
bool_t xdr_longlong_t(XDR *xdrs, longlong t *llp);
```

xdr_longlong_t() translates between ANSI C long long integers and their external representations. This routine returns if it succeeds, otherwise. This routine is identical to xdr_hyper().

```
bool_t xdr_quadruple(XDR *xdrs, long double *pq);
```

xdr_quadruple() translates between IEEE quadruple precision floating point numbers and their external representations. This

routine returns if it succeeds, otherwise.

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

xdr_short() translates between C short integers and their external representations. This routine returns if it succeeds, otherwise.

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

xdr_u_char() translates between unsigned C characters and their external representations. This routine returns if it succeeds, otherwise.

```
bool_t xdr_u_hyper(XDR *xdrs, u longlong t *ullp);
```

xdr_u_hyper() translates between unsigned ANSI C long long integers and their external representations. This routine returns if it succeeds, otherwise.

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

A filter primitive that translates between a C unsigned integer and its external representation. This routine returns TRUE if it succeeds, FALSE otherwise.

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

xdr_u_long() translates between C unsigned long integers and their external representations. This routine returns if it succeeds, otherwise.

```
bool_t xdr_u_longlong_t(XDR *xdrs, u longlong t
```

```
    *ullp);
```

xdr_u_longlong_t() translates between unsigned ANSI C long long integers and their external representations. This routine returns if it succeeds, otherwise. This routine is identical to xdr_u_hyper().

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

xdr_u_short() translates between C unsigned short integers and their external representations. This routine returns if it succeeds, otherwise.

```
bool_t xdr_void(void);
```

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

SEE ALSO

rpc, xdr_admin, xdr_complex, xdr_create

NAME

rpc_soc, authunix_create, authunix_create_default, callrpc, clnt_broadcast, clntraw_create, clnttcp_create, clntudp_bufcreate, clntudp_create, get_myaddress, pmap_getmaps, pmap_getport, pmap_rmtcall, pmap_set, pmap_unset, registerrpc, svc_fds, svc_getcaller, svc_getreq, svc_register, svc_unregister, svcfd_create, svcraw_create, svctcp_create, svcudp_bufcreate, svcudp_create, xdr_authunix_parms obsolete library routines for RPC

DESCRIPTION

RPC routines allow C programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a request to the server. Upon receipt of the request, the server calls a dispatch routine to perform the requested service, and then sends back a reply. Finally, the procedure call returns to the client.

The routines described in this manual page have been superseded by other routines. The preferred routine is given after the description of the routine. New programs should use the preferred routines, as support for the older interfaces may be dropped in future releases.

File Descriptors

Transport independent RPC uses TLI as its transport interface instead of sockets.

Some of the routines described in this section (such as clnttcp_create()) take a pointer to a file descriptor as one of the

parameters. If the user wants the file descriptor to be a socket, then the application will have to be linked with both librpcsoc and libnsl. If the user passed as the file descriptor, and the application is linked with libnsl only, then the routine will return a TLI file descriptor and not a socket.

Routines

The following routines require that the header <rpc.h> be included. The symbol should be defined so that the appropriate function declarations for the old interfaces are included through the header files.

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

```
AUTH * authunix_create(char *host, int uid , int
   gid, int grouplen, int gidlistp);
```

Create and return an RPC authentication handle that contains authentication information. The parameter **host** is the name of the machine on which the information was created; **uid** is the user's user ID; **gid** is the user's current group ID; **grouplen** and **gidlistp** refer to a counted array of groups to which the user belongs.

Warning: It is not very difficult to impersonate a user.

Warning: This routine exists for backward compatibility only, and is obsoleted by authsys_create() (see rpc_clnt_auth).

```
AUTH * authunix_create_default(void)
```

Call authunix_create() with the appropriate parameters.

Warning: This routine exists for backward compatibility only, and is obsoleted by

authsys_create_default() (see rpc_clnt_auth).

```
callrpc(char *host, u long prognum, u long
    versnum, u long procnum, xdrproc t inproc, char
    *in, xdrproc t outproc, char *out);
```

Call the remote procedure associated with **prognum**, **versnum**, and **procnum** on the machine, **host**. The parameter **inproc** is used to encode the procedure's parameters, and **outproc** is used to decode the procedure's results; **in** is the address of the procedure's argument, and **out** is the address of where to place the result(s). This routine returns 0 if it succeeds, or the value of enum clnt_stat cast to an integer if it fails. The routine clnt_perrno() (see rpc_clnt_calls) is handy for translating failure statuses into messages.

Warning: You do not have control of timeouts or authentication using this routine. This routine exists for backward compatibility only, and is obsoleted by rpc_call() (see rpc_clnt_calls).

```
enum clnt_stat clnt_broadcast(u_long prognum, u
    long versnum, u long procnum, xdrproc t inproc,
    char *in, xdrproc t outproc, char *out,
    resultproc t eachresult);
```

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

eachresult(char ***out**, struct sockaddr in ***addr**);

where **out** is the same as **out** passed to clnt_broadcast(), except that the remote procedure's output is decoded there; **addr** points to the address of the machine that sent the results. If eachresult() returns 0 clnt_broadcast() waits for more replies; otherwise it returns with appropriate status. If eachresult() is NULL, clnt_broadcast() returns

without waiting for any replies.

Warning: Broadcast packets are limited in size to the maximum transfer unit of the transports involved. For Ethernet, the callers argument size is approximately 1500 bytes. Since the call message is sent to all connected networks, it may potentially lead to broadcast storms. clnt_broadcast() uses SB AUTH_SYS credentials by default (see rpc_clnt_auth).

Warning: This routine exists for backward compatibility only, and is obsoleted by rpc_broadcast() (see rpc_clnt_calls).

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

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

Warning: This routine exists for backward compatibility only, and has the same functionality as clnt_raw_create() (see rpc_clnt_create), which obsoletes it.

```
CLIENT * clnttcp_create(struct sockaddr_in *addr,
    u long prognum, u long versnum, int *fdp, u int
    *sendsz, u int *recvsz);
```

This routine creates an RPC client for the remote program **prognum**, version **versnum**; the client uses TCP/IP as a transport. The remote program is located at Internet address **addr**. If **addr->sin port** is 0, then it is set to the actual port that the remote program is listening on (the remote rpcbind service is consulted for

this information). The parameter ***fdp** is a file descriptor, which may be open and bound; if it is RPC_ANYSOCK, then this routine opens a new one and sets ***fdp**. Refer to the File Descriptor section for more information. Since TCP-based RPC uses buffered I/O, the user may specify the size of the send and receive buffers with the parameters **sendsz** and **recvsz**; values of 0 choose suitable defaults. This routine returns NULL if it fails.

Warning: This routine exists for backward compatibility only. clnt_create(), clnt_tli_create(), or clnt_vc_create() (see rpc_clnt_create) should be used instead.

```
CLIENT * clntudp_bufcreate(struct sockaddr_in
    *addr, u long prognum, u long versnum, struct
    timeval wait, int *fdp, u_int sendsz, u int
    recvsz);
```

Create a client handle for the remote program **prognum**, on **versnum**; the client uses UDP/IP as the transport. The remote program is located at the Internet address **addr**. If **addr->sin port** is 0, it is set to port on which the remote program is listening on (the remote rpcbind service is consulted for this information). The parameter ***fdp** is a file descriptor, which may be open and bound; if it is RPC_ANYSOCK, then this routine opens a new one and sets ***fdp**. Refer to the File Descriptor section for more information. The UDP transport resends the call message in intervals of **wait** time until a response is received or until the call times out. The total time for the call to time out is specified by clnt_call() (see rpc_clnt_calls). If successful it returns a client handle, otherwise it returns NULL. The error can be printed using the clnt_pcreateerror() (see rpc_clnt_create) routine.

The user can specify the maximum packet size for sending and receiving by using **sendsz** and **recvsz** arguments for UDP-based RPC messages.

Warning: If **addr->sin port** is 0 and the requested version number **versnum** is not registered with the remote portmap service, it returns a handle if at least a version number for the given program number is registered. The version mismatch is discovered by a clnt_call() later (see rpc_clnt_calls).

Warning: This routine exists for backward compatibility only. clnt_tli_create() or clnt_dg_create() (see rpc_clnt_create) should be used instead.

```
CLIENT * clntudp_create(struct sockaddr_in *addr,
   u long progrnum, u long versnum, struct timeval
   wait, int *fdp);
```

This routine creates an RPC client handle for the remote program **prognum**, version **versnum**; the client uses UDP/IP as a transport. The remote program is located at Internet address **addr**. If **addr->sin port** is 0, then it is set to actual port that the remote program is listening on (the remote rpcbind service is consulted for this information). The parameter ***fdp** is a file descriptor, which may be open and bound; if it is RPC_ANYSOCK, then this routine opens a new one and sets ***fdp**. Refer to the File Descriptor section for more information. The UDP transport resends the call message in intervals of wait time until a response is received or until the call times out. The total time for the call to time out is specified by clnt_call() (see rpc_clnt_calls). clntudp_create() returns a client handle on success, otherwise it returns NULL. The error can be printed using the clnt_pcreateerror() (see rpc_clnt_create) routine.

Warning: Since UDP-based RPC messages can only hold up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

Warning: This routine exists for backward compatibility only. clnt_create(), clnt_tli_create(), or clnt_dg_create() (see rpc_clnt_create) should be used instead.

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

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

Warning: This routine is only intended for use with the RPC library. It returns the local system's address in a form compatible with the RPC library, and should not be taken as the system's actual IP address. In fact, the ***addr** buffer's host address part is actually zeroed. This address may have only local significance and should NOT be assumed to be an address that can be used to connect to the local system by remote systems or processes.

Warning: This routine remains for backward compatibility only. The routine netdir_getbyname() (see netdir) should be used with the name to retrieve the local system's network address as a **netbuf** structure.

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

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

Warning: This routine exists for backward compatibility only, enhanced functionality is provided by rpcb_getmaps() (see rpcbind).

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

A user interface to the portmap service, which returns the port number on which waits a service that supports program **prognum**, version **versnum**, and speaks the transport protocol associated with **protocol**. The value of **protocol** is most likely IPPROTO UDP or IPPROTO TCP. A return value of 0 means that the mapping does

not exist or that the RPC system failured to contact the remote portmap service. In the latter case, the global variable rpc_createerr contains the RPC status.

Warning: This routine exists for backward compatibility only, enhanced functionality is provided by rpcb_getaddr() (see rpcbind).

```
enum clnt_stat pmap_rmtcall(struct sockaddr_in
    *addr, u long prognum, u long versnum, u long
    procnum, char *in, xdrproct_t inproc, char
    *out, xdrproct t outproc, struct timeval tout,
    u long *portp);
```

Request that the portmap on the host at IP address ***addr** make an RPC on the behalf of the caller to a procedure on that host. ***portp** is modified to the program's port number if the procedure succeeds. The definitions of other parameters are discussed in callrpc() and clnt_call() (see rpc_clnt_calls).

[Note] This procedure is only available for the UDP transport.

Warning: If the requested remote procedure is not registered with the remote portmap then no error response is returned and the call times out. Also, no authentication is done.

Warning: This routine exists for backward compatibility only, enhanced functionality is provided by rpcb_rmtcall() (see rpcbind).

```
bool_t pmap_set(u_long prognum, u long versnum, u
    long protocol, u short port);
```

A user interface to the portmap service, that establishes a mapping between the triple [**prognum**, **versnum**, **protocol**] and **port** on the machine's portmap service. The value of **protocol** may be or IPPROTO TCP. Formerly, the routine failed if the requested **port** was found to be in use. Now, the routine only fails if it finds that

port is still bound. If **port is not bound**, **the routine** completes the requested registration. This routine returns 1 if it succeeds, 0 otherwise. Automatically done by svc_register().

Warning: This routine exists for backward compatibility only, enhanced functionality is provided by rpcb_set() (see rpcbind).

```
bool_t pmap_unset(u_long prognum, u long versnum);
```

A user interface to the portmap service, which destroys all mapping between the triple [**prognum**, **versnum**, **allprotocols**] and **port** on the machine's portmap service. This routine returns one if it succeeds, 0 otherwise.

Warning: This routine exists for backward compatibility only, enhanced functionality is provided by rpcb_unset() (see rpcbind).

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

This routine returns the network address, represented as a struct sockaddr_in, of the caller of a procedure associated with the RPC service transport handle, **xprt**.

Warning: This routine exists for backward compatibility only, and is obsolete. The preferred interface is svc_getrpccaller() (see rpc_svc_reg), which returns the address as a struct netbuf.

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

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

This routine is similar to svc_getreqset() but is limited to 32 descriptors.

Warning: This interface is obsoleted by svc_getreqset().

```
SVCXPRT * svcfd_create(int fd, u int sendsz, u int
    recvsz);
```

Create a service on top of any open and bound descriptor. Typically, this descriptor is a connected file descriptor for a stream protocol. Refer to the File Descriptor section for more information. **sendsz** and **recvsz** indicate sizes for the send and receive buffers. If they are 0, a reasonable default is chosen.

Warning: This interface is obsoleted by svc_fd_create() (see rpc_svc_create).

```
SVCXPRT * svcraw_create(void);
```

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

Warning: This routine exists for backward compatibility only, and has the same functionality of svc_raw_create() (see rpc_svc_create), which obsoletes it.

```
SVCXPRT * svctcp_create(int fd, u int sendsz, u
    int recvsz);
```

This routine creates a TCP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the file descriptor **fd**, which may be RPC_ANYSOCK, in which case a new file descriptor is created. If the file descriptor is not bound to a local TCP port, then this routine binds it to an arbitrary port. Refer

to the File Descriptor section for more information. Upon completion, **xprt->xp fd** is the transport's file descriptor, and **xprt->xp port** is the transport's port number. This routine returns NULL if it fails. Since TCP-based RPC uses buffered I/O, users may specify the size of buffers; values of 0 choose suitable defaults.

Warning: This routine exists for backward compatibility only. svc_create(), svc_tli_create(), or svc_vc_create() (see rpc_svc_create) should be used instead.

```
SVCXPRT * svcudp_bufcreate(int fd, u int sendsz, u
    int recvsz);
```

This routine creates a UDP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the file descriptor **fd**. If **fd** is RPC_ANYSOCK , then a new file descriptor is created. If the file descriptor is not bound to a local UDP port, then this routine binds it to an arbitrary port. Upon completion, **xprt->xp fd** is the transport's file descriptor, and **xprt->xp port** is the transport's port number. Refer to the File Descriptor section for more information. This routine returns NULL if it fails.

The user specifies the maximum packet size for sending and receiving UDP-based RPC messages by using the **sendsz** and **recvsz** parameters.

Warning: This routine exists for backward compatibility only. svc_tli_create(), or svc_dg_create() (see rpc_svc_create) should be used instead.

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

This routine creates a UDP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the file descriptor **fd**, which may be RPC_ANYSOCK , in which case a new file descriptor is created. If the file descriptor is not bound to a local UDP port, then this routine binds it to an arbitrary port. Upon

completion, **xprt->xp fd** is the transport's file descriptor, and **xprt->xp port** is the transport's port number. This routine returns NULL if it fails.

Warning: Since UDP-based RPC messages can only hold up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

Warning: This routine exists for backward compatibility only. svc_create(), svc_tli_create(), or svc_dg_create() (see rpc_svc_create) should be used instead.

```
registerrpc(u_long prognum, u long versnum, u long
   procnum, char *(*procname)(), xdrproc t inproc,
   xdrproc t outproc);
```

Register program **prognum**, procedure **procname**, and version **versnum** with the RPC service package. If a request arrives for program **prognum**, version **versnum**, and procedure **procnum**, **procname** is called with a pointer to its parameter(s); **procname** should return a pointer to its static result(s); **inproc** is used to decode the parameters while **outproc** is used to encode the results. This routine returns 0 if the registration succeeded, -1 otherwise.

svc_run() must be called after all the services are registered.

Warning: This routine exists for backward compatibility only, and is obsoleted by rpc_reg() (see rpc_svc_calls).

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

Associates **prognum** and **versnum** with the service dispatch procedure, **dispatch**. If **protocol** is 0, the service is not registered with the portmap service. If **protocol** is non-zero, then a mapping of the triple [**prognum**, **versnum**, **protocol**] to **xprt->xp port** is

established with the local portmap service (generally **protocol** is 0, IPPROTO UDP or IPPROTO TCP). The procedure **dispatch** has the following form:

dispatch(struct svc_req *__request__, SVCXPRT *__xprt__);

The svc_register() routine returns one if it succeeds, and 0 otherwise.

Warning: This routine exists for backward compatibility only; enhanced functionality is provided by svc_reg() (see rpc_svc_calls).

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

Remove all mapping of the double [**prognum**, **versnum**] to dispatch routines, and of the triple [**prognum**, **versnum**, **all-protocols**] to port number from portmap.

Warning: This routine exists for backward compatibility, enhanced functionality is provided by svc_unreg() (see rpc_svc_calls).

```
xdr_authunix_parms(XDR *xdrs, struct authunix
    parms *aupp);
```

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

Warning: This routine exists for backward compatibility only, and is obsoleted by xdr_authsys_parms() (see rpc_xdr).

SEE ALSO

rpc, rpc_clnt_auth, rpc_clnt_create, rpc_clnt_calls, rpc_svc_calls, rpc_svc_create, rpc_svc_err, rpc_svc_reg, rpcbind

NAME

rpc_init, rpc_init_custom, rpc_set_syslog, rpc_set_user - Windows extensions to TIRPC

DESCRIPTION

These routine-like macros are specific to the port of TIRPC to MS-Windows. They are not in the standard TIRPC distribution. Programmers must call either rpc_init() or rpc_init_custom() before using any other TIRPC routines. rpc_init() is the simplest routine which sets up TIRPC to use a lot defaults. If a programmer is unhappy with any of these defaults, he or she may be interested in the other extension routines.

Routines

```
#include <rpc.h>

#include <rpcinit.h>

void FAR * PASCAL FAR rpc_init(HANDLE hInst)
```

This macro initializes TIRPC using a lot of quite sensible defaults. The file <rpcinit.h> MUST be included in order to use rpc_init() and should only be included in the module calling rpc_init(). The single parameter (hInst) is the instance handle of the Windows application. The defaults are set up as follows:

allocation: malloc and free

error printing: fputs to stderr

uid, gid, gid group set, pid: all zero

XDR over stdio: uses fwrite, fread, ftell, fseek and fflush

The value NULL is returned if initialization failed, non-NULL otherwise.

```
#include <rpc.h>


void FAR * PASCAL FAR  rpc_init_custom(
       void FAR *(PASCAL FAR *the_alloc)(unsigned
          long size, void FAR *handle),
       void (PASCAL FAR *the_free)(void FAR
          *ptr,void FAR *handle),
       void FAR *alloc_handle,
       void (PASCAL FAR *the_print)(const char FAR
          *str, void FAR *handle),
       void FAR *print_handle,
       int (PASCAL FAR *the_fread)(void FAR *ptr,
          size_t size, size_t nmemb, FILE FAR
          *stream),
       int (PASCAL FAR *the_fwrite)(const void FAR
          *ptr, size_t size, size_t nmemb, FILE FAR
          *stream), long (PASCAL FAR
          *the_ftell)(FILE FAR *stream), int
          (PASCAL FAR *the_fseek)(FILE FAR *stream,
          long offset, int whence),
       int (PASCAL FAR *the_fflush)(FILE FAR
          *stream));
```

This macro allows the programmer to set up some different defaults

upon initialization. the_alloc and the_free are pointers to the programmer's allocation routines. They are callback routines which are called by TIRPC from time to time as it needs to allocate or free memory. Both parameters take a handle argument which TIRPC uses to pass back to the application the alloc_handle parameter. Similarly, the_print and print_handle are used for times when TIRPC needs to output a string. Finally, the_fread, the_fwrite, the_ftell, the_fseek and the_fflush are used by the XDR/stdio feature of TIRPC. They follow the same API as the ANSI stdio routines, except that that stream parameter is not required to actually be of type FILE *, but can be any pointer type (cast to FILE * to avoid compiler errors).

Except for the allocation parameters, any of the other defaults may be set to NULL which will disable the feature.

The value NULL is returned if initialization failed, non-NULL otherwise.

```
#include <rpc.h>
void PASCAL FAR
   rpc_set_syslog(syslog_callback_proc
      the_callback);
typedef void (PASCAL FAR*syslog_callback_proc)
   (int priority,const char FAR *msg, void FAR
   *handle);
```

Use rpc_set_syslog if you want syslog messages generated by TIRPC to be handled differently than just getting printed using the default print function. The handle argument to the callback syslog procedure is the same as the print_handle passed to rpc_init_custom() or NULL if using rpc_init(). Note that only priorities LOG_ERR and LOG_DEBUG are ever used by TIRPC.

```
#include <rpc.h>
void PASCAL FAR rpc_set_user(
      int the_pid,
      int the_uid,
      int the_gid,
int the_grouplen,
int FAR *the_groups);
```

Use rpc_set_user if you want defaults for these values other than zero. The PID (process ID) is only used for generating RPC transaction IDs and is fairly unimportant. It can be any value, but ideally should be a number unique to each instance of an application to insure unique transaction IDs. The uid and gids are used for authentication purposes.

WARNING: Unlike Unix, Windows has no way to insure the uids and gids are authenticated via a password. Therefore, an application should try to authenticate these values before passing them to rpc_set_user and NOT allow a user to set them to any value.

TIRPC DLL Release Notes

Release 2

1. Interface

The interface is for the most part, the same as on Unix machines. The differences are the following:

a. You must initialize the library before using it.

b. Some #include files have different names.

c. No support for DES or Kerberos authentication.

d. netdir is hardcoded to support UDP and TCP only.

More information on this is described later in this section.

2. Bugs

There is at least one problem with multiple applications using the same DLL (TIRPC.DLL). If one application is blocked on a WINSOCK call, and then is terminated, the DLL does not get unloaded. This results in errors when other TIRPC applications are started. You normally have to quit Windows to get around this problem. This bug is unlikely to be fixed in the near future, and it is strongly recommended that an application wait for any blocking RPC call to terminate before exiting.

3. Initialization

To initialize the library, your code should call rpc_init() with a single argument of an instance handle (received from the Windows function WinMain()). This call sets up default callback routines for use with the TIRPC.DLL.

The default will be that allocation will be via malloc/free, error printing (clnt_perror(), etc.) will use fputs to stderr, stdio XDR (xdrstdiocreate()) will used fwrite/fread/ftell/fseek/fflush, uid, gid and pid will be zero, and no syslogs will be output. To set

the uid, gid and pid, use rpc_set_user(). To set the syslog output routine, use rpc_set_syslog(). To change the memory allocator or the stdio routines used by XDR stdio, call rpc_init_custom() instead of rpc_init(). Note that if you use the default rpc_init(), you should #include <rpcinit.h> in the file which calls rpc_init(). The include of rpcinit.h will cause the default allocation and stdio routines to be defined in that module. You are only required to set the memory allocation functions. If the other functions are set to NULL, then they will be disabled. For example, setting the error print function to NULL will disable error printing. Also, setting the stdio functions to NULL will disable xdr over stdio.

Note that each of the callback routines has a "handle" argument which is simply something that will be passed back to the application. There is an allocation handle for alloc/free, a print handle for error printing and a stdio handle (a FILE *) for xdr over stdio. Note that there is no stdio handle passed to rpc_init_custom(). The stdio handle is instead passed to xdrstdio_create(). It does not actually have to be a standard FILE pointer, but can be your own creation with the appropriate callbacks.

4. Callback routines

All callback routines pointers passed to any TIRPC function should be of the type created by the Windows function MakeProcInstance(). This includes the callback routines used in rpc_init_custom() described above, and most xdr routines. The callback functions should be of type FAR PASCAL and exported in a .DEF file (or automatically via the _export keyword).

Note that the Unix rpcgen program will not generate correct code for use with Windows because of restrictions Windows places upon callback routines. The Unix rpcgen output can be hand modified easily enough, however, by finding all references to generated xdr functions used as callback functions, replacing them with pointers returned from MakeProcInstance() and making the original referenced functions be "FAR PASCAL _export".

5. Include files

The assumption with the include files is that you have defined the rpc directory to be part of your compiler include path. This means that instead of including <rpc/rpc.h> as you would under Unix, you would include just plain <rpc.h> instead. If you need to include any other include files besides rpc.h, you should check to see if they have been renamed to work under DOS 8.3 names. An easy way to do this is to look at the commented out "#pragma ident" in each include file which contains the original Unix name for the include.

6. Netdir/TLI

The netdir, netconfig and netpath and tli routines are all made available to the client, in addition to the standard RPC routines. They are not a complete implementation of netdir, so use them at your own risk.

7. Implementation issues

Because RPC contains state, steps were taken to insure that this state was always contained in the application rather than the DLL. The technique for doing this is to pass a state parameter to all stateful routines. For each stateful RPC routine, a macro is defined to call the stateless DLL routine with the parameters plus an added state parameter. This should be transparent to the caller, but will be noticable in any way in which a macro is incompatible with a function, such as in linking or passing the address of a function. You will notice that most of the routines in the DLL start with the prefix "sl", such as "slclnt_create". This means that clnt_create() was stateful. A stateless version (slcnt_create) is provided in the DLL, and a macro (clnt_create) is provided to pass the state to the stateless routine.

8. Changes since Release 1

a. Calling conventions have been changed from C to PASCAL. Because of this change, you will have to recompile all code,

MakeProcInstance() for any callback routines and insure that the callback routines are FAR PASCAL and exported.

b. The rpc_init() macro now takes an argument, which is the instance handle of the application (passed in WinMain() function).

c. A bug has been fixed with respect to timeouts. Timeouts were restricted to 32 seconds previously because they were stored in 16-bit quantities. Now, you may use very large timeouts since they are stored as 32-bit quantities.