

NAME

rpc - library routines for remote procedure calls

SYNOPSIS AND DESCRIPTION

These routines allow C programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a data packet to the server. Upon receipt of the packet, 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.

To take use of these routines, include the header file `<rpc/rpc.h>`.

The prototypes below make use of the following types:

```
typedef int bool_t;
typedef bool_t (*xdrproc_t) (XDR *, void *, ...);
typedef bool_t (*resultproc_t) (caddr_t resp,
struct sockaddr_in *raddr);
```

See the header files for the declarations of the *AUTH*, *CLIENT*, *SVCXPRT*, and *XDR* types.

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

A 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 *authunix_create(char *host, int uid, int gid,
int len, int *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. It is easy to impersonate a user.

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

Calls **authunix_create()** with the appropriate parameters.

```
int callrpc(char *host, unsigned long prognum,
unsigned long versnum, unsigned 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 *in* is the address of the procedure's argument(s), and *out* is the address of where to place the result(s); *inproc* is used to encode the procedure's parameters, and *outproc* is used to decode the procedure's results. This routine returns zero if it succeeds, or the value of **enum clnt_stat** cast to an integer if it fails. The routine **clnt_perr no()** is handy for translating failure statuses into messages.

Warning: calling remote procedures with this routine uses UDP/IP as a transport; see **clntudp_create()** for restrictions. You do not have control of timeouts or authentication using this routine.

```
enum clnt_stat clnt_broadcast(unsigned long prognum,
unsigned long versnum, unsigned 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 it 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 zero, `clnt_broadcast()` waits for more replies; otherwise it returns with appropriate status.

Warning: broadcast sockets are limited in size to the maximum transfer unit of the data link. For ethernet, this value is 1500 bytes.

```
enum clnt_stat clnt_call(CLIENT *clnt, unsigned long procnum,  
xdrproc_t inproc, char *in,  
xdrproc_t outproc, char *out,  
struct timeval tout);
```

A macro that calls the remote procedure `procnum` associated with the client handle, `clnt`, which is obtained with an RPC client creation routine such as `clnt_create()`. The parameter `in` is the address of the procedure's argument(s), and `out` is the address of where to place the result(s); `inproc` is used to encode the procedure's parameters, and `outproc` is used to decode the procedure's results; `tout` is the time allowed for results to come back.

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

A 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 socket, it will close it also. Otherwise, the socket remains open.

```
CLIENT *clnt_create(char *host, unsigned long prog,  
unsigned long vers, char *proto);
```

Generic client creation routine. `host` identifies the name of the remote host where the server is located. `proto` indicates which kind of transport protocol to use. The currently supported values for this field are "udp" and "tcp". Default timeouts are set, but can be modified using `clnt_control()`.

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

```
bool_t clnt_control(CLIENT *cl, int req, char *info);
```

A macro used 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 UDP and TCP, 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 parameter passed to `clnt_call()` will be ignored in all future calls.

```
CLGET_SERVER_ADDR struct sockaddr_in // get server's address
```

The following operations are valid for UDP 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 "UDP RPC" waits for the server to reply before retransmitting the request.

```
clnt_freeres(CLIENT *clnt, xdrproc_t outproc, char *out);
```

A 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 one if the results were successfully freed, and zero otherwise.

void `clnt_geterr(CLIENT *clnt, struct rpc_err *errp);`

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

void `clnt_pcreateerror(char *s);`

Print a message to standard error indicating why a client RPC handle could not be created. The message is prepended with string *s* and a colon. Used when a `clnt_create()`, `clntraw_create()`, `clnttcp_create()`, or `clntudp_create()` call fails.

void `clnt_perrno(enum clnt_stat stat);`

Print a message to standard error corresponding to the condition indicated by *stat*. Used after `call_rpc()`.

clnt_perror(CLIENT *clnt, char *s);

Print a message to 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. Used after `clnt_call()`.

char *clnt_screateerror(char *s);

Like `clnt_pcreateerror()`, except that it returns a string instead of printing to the standard error.

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

char *clnt_sperrno(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. The string ends with a NEWLINE.

`clnt_sperrno()` is used instead of `clnt_perrno()` if 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(3)`, or if a message format different than that supported by `clnt_perrno()` is to be used. Note: unlike `clnt_sperror()` and `clnt_screateerror()`, `clnt_sperrno()` returns pointer to static data, but the result will not get overwritten on each call.

char *clnt_sperror(CLIENT *rpch, char *s);

Like `clnt_perror()`, except that (like `clnt_sperrno()`) it returns a string instead of printing to standard error.

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

CLIENT *clntraw_create(unsigned long prognum, unsigned long versnum);

This routine creates a toy 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 `svcrw_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.

**CLIENT *clnttcp_create(struct sockaddr_in *addr,
unsigned long prognum, unsigned long versnum,
int *sockp, unsigned int sendsz, unsigned 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 zero, then it is set to the actual port that the remote program is listening on (the remote `portmap` service is consulted for this information). The parameter *sockp* is a socket; if it is `RPC_ANYSOCK`, then this routine opens a new one and sets *sockp*. 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 zero choose suitable defaults. This routine returns NULL if it fails.

```
CLIENT *clntudp_create(struct sockaddr_in *addr,  
unsigned long prognum, unsigned long versnum,  
struct timeval wait, int *sockp);
```

This routine creates an RPC client 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 zero, then it is set to actual port that the remote program is listening on (the remote **portmap** service is consulted for this information). The parameter *sockp* is a socket; if it is **RPC_ANYSOCK**, then this routine opens a new one and sets *sockp*. 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()**.

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

```
CLIENT *clntudp_bufcreate(struct sockaddr_in *addr,  
unsigned long prognum, unsigned long versnum,  
struct timeval wait, int *sockp,  
unsigned int sendsize, unsigned int recosize);
```

This routine creates an RPC client for the remote program *prognum*, on *versnum*; the client uses UDP/IP as a transport. The remote program is located at Internet address *addr*. If *addr->sin_port* is zero, then it is set to actual port that the remote program is listening on (the remote **portmap** service is consulted for this information). The parameter *sockp* is a socket; if it is **RPC_ANYSOCK**, then this routine opens a new one and sets *sockp*. 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()**.

This allows the user to specify the maximum packet size for sending and receiving UDP-based RPC messages.

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

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

```
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.

```
unsigned short pmap_getport(struct sockaddr_in *addr,  
unsigned long prognum, unsigned long versnum,  
unsigned int protocol);
```

A user interface to the **portmap** service, which returns the port number on which waits a service that supports program number *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 zero means that the mapping does not exist or that the RPC system failed to contact the remote **portmap** service. In the latter case, the global variable *rpc_createerr* contains the RPC status.

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

A user interface to the **portmap** service, which instructs **portmap** on the host at IP address **addr* to make an RPC call on your behalf to a procedure on that host. The parameter **portp* will be modified to the program's port number if the procedure succeeds. The definitions of other parameters are discussed in **callrpc()** and **clnt_call()**. This procedure should be used for a “ping” and nothing else. See also **clnt_broadcast()**.

bool_t pmap_set(unsigned long prognum, unsigned long versnum, unsigned int protocol, unsigned short port);

A user interface to the **portmap** service, which establishes a mapping between the triple [*prognum, versnum, protocol*] and *port* on the machine's **portmap** service. The value of *protocol* is most likely **IPPROTO_UDP** or **IPPROTO_TCP**. This routine returns one if it succeeds, zero otherwise. Automatically done by **svc_r_egister()**.

bool_t pmap_unset(unsigned long prognum, unsigned long versnum);

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

int registerrpc(unsigned long prognum, unsigned long versnum, unsigned long procnum, char *(*procname)(char *), xdrproc_t inproc, xdrproc_t outproc);

Register procedure *procname* 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 zero if the registration succeeded, -1 otherwise.

Warning: remote procedures registered in this form are accessed using the UDP/IP transport; see **svcudp_create()** for restrictions.

struct rpc_createerr rpc_createerr;

A global variable whose value is set by any RPC client creation routine that does not succeed. Use the routine **clnt_pcreateerror()** to print the reason why.

void svc_destroy(SVCXPRT *xpirt);

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

fd_set svc_fdset;

A global variable reflecting the RPC service side's read file descriptor bit mask; it is suitable as a parameter to the **select(2)** system call. This is of interest only if a service implementor does their own asynchronous event processing, instead of calling **svc_run()**. This variable is read-only (do not pass its address to **select(2)!**), yet it may change after calls to **svc_getreqset()** or any creation routines.

int svc_fds;

Similar to **svc_fdset**, but limited to 32 descriptors. This interface is obsolete by **svc_fdset**.

svc_freeargs(SVCXPRT *xpirt, xdrproc_t inproc, char *in);

A macro that frees any data allocated by the RPC/XDR system when it decoded the arguments to a service procedure using **svc_getargs()**. This routine returns 1 if the results were successfully freed, and zero otherwise.

svc_getargs(SVCXPRT *xpirt, xdrproc_t inproc, char *in);

A macro that decodes the arguments of an RPC request associated with the RPC service transport handle, *xpirt*. 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 one if decoding succeeds, and zero otherwise.

struct sockaddr_in *svc_getcaller(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_getreqset(fd_set *rdfs);

This routine is of interest only if a service implementor does not call **svc_run()**, but instead implements custom asynchronous event processing. It is called when the **select(2)** system call has determined that an RPC request has arrived on some RPC socket(s); *rdfs* is the resultant read file descriptor bit mask. The routine returns when all sockets associated with the value of *rdfs* have been serviced.

void svc_getreq(int rdfs);

Similar to **svc_getreqset()**, but limited to 32 descriptors. This interface is obsoleted by **svc_getreqset()**.

bool_t svc_register(SVCXPRT *xprt, unsigned long prognum, unsigned long versnum, void (*dispatch)(svc_req *, SVCXPRT *), unsigned long protocol);

Associates *prognum* and *versnum* with the service dispatch procedure, *dispatch*. If *protocol* is zero, the service is not registered with the **portmap** service. If *protocol* is nonzero, then a mapping of the triple [*prognum,versnum,protocol*] to *xprt->xp_port* is established with the local **portmap** service (generally *protocol* is zero, **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 zero otherwise.

void svc_run(void);

This routine never returns. It waits for RPC requests to arrive, and calls the appropriate service procedure using **svc_getreq()** when one arrives. This procedure is usually waiting for a **select(2)** system call to return.

bool_t svc_sendreply(SVCXPRT *xprt, xdrproc_t outproc, char *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 one if it succeeds, zero otherwise.

void svc_unregister(unsigned long prognum, unsigned long versnum);

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

void svcerr_auth(SVCXPRT *xprt, 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(SVCXPRT *xprt);

Called by a service dispatch routine that cannot successfully decode its parameters. See also **svc_getargs()**.

void svcerr_noproc(SVCXPRT *xprt);

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

void svcerr_noprogram(SVCXPRT *xpirt);

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

void svcerr_progvers(SVCXPRT *xpirt);

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

void svcerr_systemerr(SVCXPRT *xpirt);

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(SVCXPRT *xpirt);

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

SVCXPRT *svcfld_create(int fd, unsigned int sendsize, unsigned int recvsize);

Create a service on top of any open descriptor. Typically, this descriptor is a connected socket for a stream protocol such as TCP. *sendsize* and *recvsize* indicate sizes for the send and receive buffers. If they are zero, a reasonable default is chosen.

SVCXPRT *svcrw_create(void);

This routine creates a toy 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 **clntrw_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.

SVCXPRT *svctcp_create(int sock, unsigned int send_buf_size, unsigned int recv_buf_size);

This routine creates a TCP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket *sock*, which may be **RPC_ANYSOCK**, in which case a new socket is created. If the socket is not bound to a local TCP port, then this routine binds it to an arbitrary port. Upon completion, *xpirt->xp_sock* is the transport's socket descriptor, and *xpirt->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 zero choose suitable defaults.

SVCXPRT *svcdp_bufcreate(int sock, unsigned int sendsize, unsigned int recvsize);

This routine creates a UDP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket *sock*, which may be **RPC_ANYSOCK**, in which case a new socket is created. If the socket is not bound to a local UDP port, then this routine binds it to an arbitrary port. Upon completion, *xpirt->xp_sock* is the transport's socket descriptor, and *xpirt->xp_port* is the transport's port number. This routine returns NULL if it fails.

This allows the user to specify the maximum packet size for sending and receiving UDP-based RPC messages.

SVCXPRT *svcdp_create(int sock);

This call is equivalent to *svcdp_bufcreate(sock,SZ,SZ)* for some default size *SZ*.

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

Used for encoding RPC reply messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

bool_t 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.

void xdr_callhdr(XDR *xdrs, struct rpc_msg *chdr);

Used for describing RPC call header messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

bool_t xdr_callmsg(XDR *xdrs, struct rpc_msg *cmsg);

Used for describing RPC call messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

bool_t xdr_opaque_auth(XDR *xdrs, struct opaque_auth *ap);

Used for describing RPC authentication information messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

bool_t xdr_pmap(XDR *xdrs, struct pmap *regs);

Used for describing parameters to various **portmap** procedures, externally. This routine is useful for users who wish to generate these parameters without using the **pmap** interface.

bool_t xdr_pmaplist(XDR *xdrs, struct pmaplist **rp);

Used for describing a list of port mappings, externally. This routine is useful for users who wish to generate these parameters without using the **pmap** interface.

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

Used for describing RPC reply messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

bool_t xdr_replymsg(XDR *xdrs, struct rpc_msg *rmsg);

Used for describing RPC reply messages. This routine is useful for users who wish to generate RPC style messages without using the RPC package.

void xpirt_register(SVCXPRT *xpirt);

After RPC service transport handles are created, they should register themselves with the RPC service package. This routine modifies the global variable *svc_fds*. Service implementors usually do not need this routine.

void xpirt_unregister(SVCXPRT *xpirt);

Before an RPC service transport handle is destroyed, it should unregister itself with the RPC service package. This routine modifies the global variable *svc_fds*. Service implementors usually do not need this routine.

SEE ALSO

[xdr\(3\)](#)

The following manuals:

- Remote Procedure Calls: Protocol Specification
- Remote Procedure Call Programming Guide
- rpcgen Programming Guide

RPC: Remote Procedure Call Protocol Specification, RFC 1050, Sun Microsystems, Inc., USC-ISI.

COLOPHON

This page is part of release 3.74 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at <http://www.kernel.org/doc/man-pages/>.