NAME

rpc_rac, rac_drop, rac_poll, rac_recv, rac_send – remote asynchronous calls

SYNOPSIS

cc [*flag* ...] *file* ... **–lrac –lnsl** [*library* ...]

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

MT-LEVEL

Unsafe

DESCRIPTION

The remote asynchronous calls (RAC) package is a special interface to the RPC library that allows messages to be sent using the RPC protocol without blocking during the time between when the message is sent and the reply is received. To RPC servers, RAC messages are indistinguishable from RPC messages.

A client establishes an RPC session in the usual way (see **rpc_clnt_create**(3N)). A RAC message is sent using **rac_send( )**. This routine returns immediately, allowing the client to conduct other processing. When the client wants to determine whether the returned value from the call has been received, **rac_poll( )** is used. **rac_recv( )** is used to collect the returned value; it can also be used to block while waiting for the returned value to arrive. **rac_drop( )** is used to inform the RPC library that the client is no longer interested in the results of a particular RAC message.

#include <rpc/rpc.h>

void rac_drop(CLIENT **cl*, **void** **h***);**

> **rac_drop( )** should be called when the user is no longer interested in the result of a **rac_send( )** currently in progress. No message to the server is generated by this call, but any subsequent reply received for this handle will be silently dropped. It also frees any space occupied by the asynchronous call handle *h*.
>
> After a call to **rac_drop( )** the handle referred to by *h* is invalid. It may no longer be used in any asynchronous operation.

enum clnt_stat rac_poll(CLIENT **cl*, **void** **h***);**

> **rac_poll( )** returns the status of the call currently in progress on the <CLIENT, asynchronous handle> tuple referred to by *cl* and *h*.
>
> **rac_poll( )** return values are:
>
> **RPC_SUCCESS**
> A reply has been received and is available for reading by **rac_recv( )**.
>
> **RPC_INPROGRESS**
> No reply has been received. The call referred to by the given handle has not yet timed out.
>
> **RPC_TIMEDOUT**
> No reply has been received. The call referred to by the given handle has exceeded the maximum timeout value specified in **rac_send( )**.

RPC_STALERACHANDLE
Either the handle referred to by *h* is invalid or no call is currently in progress for the given <CLIENT, asynchronous handle> tuple.

RPC_CANTRECV
Either the file descriptor associated with the given CLIENT handle is bad, or an error occurred while attempting to receive a packet.

RPC_SYSTEMERROR
Space could not be allocated to receive a packet.

On unreliable transports, a call to **rac_poll( )** will trigger a retransmission when necessary (that is, if a **rac_send( )** is in progress, no reply has been received, the per-call timeout has expired, and the total timeout has not yet expired).

The return value for **rac_poll( )** is independent of the RPC return value in the reply packet. Although a combination of **clnt_control( )**'s CLGET_FD request and **poll**(2) may be used to extract the proper file descriptor and poll for packets, **rac_poll( )** is still useful since it will determine whether a reply is available for a specific <CLIENT, asynchronous handle> tuple.

enum clnt_stat rac_recv(CLIENT **cl*, **void** * *h***);**

rac_recv( ) retrieves the results of a previous asynchronous RPC call, placing them in the buffer indicated in the **rac_send( )** call and using the XDR decode function supplied there. It depends on the application to have ensured that a reply is present (using **rac_poll()**). If **rac_recv( )** is called before a reply has been received, it will block awaiting a reply.

All errors normally returned by the RPC client call functions may be returned here. In addition:

RPC_STALERACHANDLE
Either the handle referred to by *h* is invalid or no call is currently in progress for the given <CLIENT, asynchronous handle> tuple.

Additionally, if a packet is present and its status is not RPC_SUCCESS, it is possible that the client credentials need refreshing. In this case, RPC_AUTHERROR is returned and the client should attempt to resend the call.

When a reply has been received, **rac_recv( )** will invoke the XDR decode procedure specified in the **rac_send( )** call. After a call to **rac_recv( )**, the handle referred to by *h* is invalid. It may no longer be used in any asynchronous operation.

void ∗**rac_send(**CLIENT ∗*cl*, **unsigned long** *proc*, **xdrproc_t** *xargs*, **void** ∗*argsp*, **xdrproc_t** *xresults*, **void** ∗*resultsp*, **struct timeval** *timeout***)**;

rac_send( ) initiates (sends to the server) an RPC call to the specified procedure. It does not await a reply from the server. *argsp* is the address of the procedure's arguments, *resultsp* is the address in which to place the results, *xargs* and *xresults* are XDR functions used to encode and decode respectively. Note: *resultsp* must be a valid pointer when **rac_recv( )** is called. *timeout* should contain the total amount of time the application is willing to wait for a reply.

Upon success, an opaque handle, known as the asynchronous handle, is returned. This handle is to be used in subsequent asynchronous calls to poll for the status of the call (**rac_poll( )**), receive the returned results of the call (**rac_recv( )**), or cancel the call (**rac_drop( )**).

On failure, *(void ∗) 0* is returned.

In case of failure, the application may retrieve the RPC failure code by calling **clnt_geterr( )** immediately after a **rac_send( )** failure (see **rpc**(3N)). Possible errors include both transient problems (such as transport failures) and permanent ones (such as XDR encoding failures).

Multiple **rac_send**s on the same client handle are permitted, but may introduce unpredictable perturbations to the current timeout and retry model used by the RPC library.

The interface imposes a limit on the amount of time a call may be in progress before it is considered to have failed. This method was chosen over limitations on the number of retries because of a desire for transport independence.

SEE ALSO

poll(2), **rpc**(3N), **rpc_clnt_create**(3N), **rpc_clnt_calls**(3N), **xdr**(3N)

WARNINGS

The RAC interface is not the recommended interface for having multiple RPC requests outstanding. The preferred method of accomplishing this in the Solaris environment is to use synchronous RPC calls with threads. The RAC interface is provided as a service to developers interested in porting RPC applications to Solaris 2.0. Use of this interface will degrade the performance of normal synchronous RPC calls (see **rpc_clnt_calls**(3N)). For these reasons, use of this interface is disparaged.

The library **librac** must be linked before **libnsl** to use RAC. If the libraries are not linked in the correct order, then the results are indeterminate.

NOTES

These interfaces are unsafe in multithreaded applications. Unsafe interfaces should be called only from the main thread.