RPC

The RPC module implements the Remote Procedure Call protocol, running on top of UDP and the 'portmapper' service. It also provides the shared library for marshalling and unmarshalling arguments in XDR format.

Process Information

Prototype Name	pmap
Link Order	does not matter
Process Name	"pmap"

Data Definitions

The header file *rpc.h* contains data definitions for the following exported datatypes and structures:

RPC_CALLBACK	This defines the type of the callback function used to link remote procedure call returns to the corresponding application layer processing. The definition is **void** (*)(**int**, **ptr**, **ROME_MESSAGE ***) where the **int** and **ptr** parameters are specified on the call to *rpc_setup_for_reply* and the **ROME_MESSAGE** is the incoming message holding the procedure return values.
RPC_FILE	An RPC FILE represents a data path to a remote service. Associated with the file are the remote *program* and *version* being accessed. The *tid* and *tid_pending* fields contain transaction identifiers which identify individual requests. The structure also contains the user and group information used to authorise requests on this data, in the *auth_code*, *machine*, *uid*, *gid*, *gids* and *gidc* fields, and a list of pending requests rooted at *r_root*.
RPC_PROCCALL	This data structure is used to hold the data for an incoming remote procedure call within the portmapper function.
RPC_REQUEST	An RPC_REQUEST data structure hold a single instance of a remote procedure call. It contains a pointer to the transport layer dataflow in *down*, and the message containing the marshalled arguments in *out*. The *cont*, *context* and *state* fields contain the callback information

used to pass the return values back to the application. The structure also contains a timer token *ttoken* used to control retransmissions of requests for which no response has been received.

Process Operation

The module has only a main process which is used for the portmapper function. Currently the portmapper function is not implemented for incoming calls.

Shared Library Macros and Routines

Use of the remote-procedure-call library is a three-stage process. Firstly a dataflow must be initialised to the remote service, using *rpc_open.* This returns a handle to an RPC FILE structure on which requests can be sent. Individual requests are created using *rpc_start_call*, the per-call arguments are added using the *xdr_add* functions and the procedure is invoked by *rpc_end_call.* The RPC library is unpended, and the *rpc_setup_for_reply* routine is used to link a callback function to the outgoing requests. All RPC replies are handled by a common routine, *rpc_reply_handler*, which calls the appropriate callback function, at which point the results can be retrieved using the *xdr_get* functions. The dataflow is terminated by the *rpc_close* routine.

The library also supports the ‘indirect’ service call through the portmapper function. In this model, the remote procedure call is embedded in an outer call to a common ‘portmapper’ function on the remote machine. In this way, the individual port numbers for services do not need to be used.

rpc_add_gid

void *rpc_addgid*(
    RPC_FILE **rval*,
    uint *gid*)

The *rpc_addgid* routine adds the group identifier *gid* to the list of groups to be sent in the authorisation part of all subsequent RPCs on *rval*.

rpc_end_call

void rpc_end_call(
    RPC_FILE **val*,
    RPC_REQUEST **ret*,
    uchar **sp*)

The *rpc_end_call* routine completes and transmits a remote procedure call request. The request *ret* is sent on the data path *val.* The *sp* value points to the end of the marshalled arguments (for example as returned by the last call to an *xdr_add* routine).

rpc_close

void *rpc_close*(
    RPC_FILE **val*)

The *rpc_close* routine terminates the data path associated with the RPC_FILE *val* and frees the resources used by the path. This routine should not be called when a remote procedure call is still in progress on the datapath.

rpc_open

RPC_FILE **rpc_open*(
  ROME_URL **where*,
  uint *prog*,
  uint *version*)

The *rpc_open* routine creates a new datapath to *version* of the *prog* service on the remote machine *where*. If the *where* structure does not specify the transport protocol, UDP is selected by default. The URL must contain at least the remote IP address and remote port number for the requested service.

rpc_portmapper_callit

uchar **rpc_portmapper_callit*(
  RPC_FILE **map*,
  uint *service*,
  uint *version*,
  uint *proc*,
  RPC_REQUEST ***ret*)

The *rpc_portmapper_callit* function initialises a remote procedure call to make an indirect call through the portmapper ‘callit’ operation. *map* should be a remote procedure call file previously opened to the portmapper service on the remote machine. The routine initialises a procedure call to the *proc* procedure of version *version* of remote service *service*. The *ret* value is initialised to point to the request structure for this call. The routine returns a pointer which can be used to add call-specific arguments to the request.

rpc_portmapper_getport

int *rpc_portmapper_getport*(
  char **host*,
  uint *prog*,
  uint *version*,
  uint *protocol*,
  uint **port*)

The *rpc_portmapper_getport* routine uses the portmapper function on the remote machine named *host* to find the service port number for the remote procedure call program *prog*, version *version* using the protocol *protocol*. The protocol value should be one of *IP_PROTO_UDP* or *IP_PROTO_TCP*. The corresponding service port on the remote machine is returned in the *port* variable. The routine returns 0 if the *port* value is set, and a non-zero error code otherwise.

rpc_portmapper_open

RPC_FILE **rpc_portmapper_open*(
  char **host*)

The *rpc_portmapper_open* opens a data path to the portmapper service on the specified *host*. The routine returns a handle to am **RPC_FILE** that can be used to generate remote procedure call requests to the portmapper, or *NULL* if the dataflow cannot be established

rpc_reply_handler

> **void** *rpc_reply_handler*(
>   **ROME_MESSAGE** **mptr*)

The *rpc_reply_handler* routine is a generic message handler for all replies to remote procedure calls. The routine calls the continuation function associated with this message (as set by *rpc_setup_for_reply*) then returns the data block and frees the resources associated with the message. This routine is suitable for placing in a list of handlers passed to the *rome_generic_handler* routine.

rpc_setuid

> **void** *rpc_setuid*(
>   **RPC_FILE** **val*,
>   **uint** *uid*,
>   **uint** *gid*,
>   **char** **machine*)

The *rpc_setuid* routine sets the parameters for UNIX-style authorisation on all subsequent requests sent on the data path *val*. The *uid* and *gid* values are a user number and group number authorising the requests and the *machine* parameter is a string identifying the local machine.

rpc_setup_for_reply

> **ROME_MESSAGE** **rpc_setup_for_reply*(
>   **RPC_REQUEST** **rval*,
>   **RPC_CALLBACK** *cproc*,
>   **int** *state*,
>   **ptr** *cx*)

The *rpc_setup_for_reply* routine prepares the RPC layer to process the response to an outgoing request represented by *rval*. The returned value is a pointer to a message which will contain the reply (so that the application may wait for that specific message). When that message is passed back into the RPC layer through the *rpc_reply_handler* routine, the callback procedure *cproc* will be called with *state* and *cx* as its integer and pointer arguments respectively.

rpc_setup_service

> **RPC_FILE** **rpc_setup_service*(
>   **char** **hostname*,
>   **uint** *prog*,
>   **uint** *version*)

The *rpc_setup-service* routine returns an RPC file connected to the correct *version* of the RPC service numbered *prog* on the machine hostname. The routines uses the port mapper to locate the port on which the service is running. If the service cannot be accessed the routine returns *NULL*.

rpc_start_call

> **uchar** **rpc_start_call*(
>   **RPC_FILE** **rval*,
>   **uint** *procno*,

uint *size*,
RPC_REQUEST ***ret*)

The *rpc_start_call* routine prepares a new remote procedure call on the data path *rval,* which should be set up to a service on a remote machine. *procno* is the remote procedure being invoked on that service and *size* is the maximum size of the request will be generated. The routine places a pointer to the **REQUEST** structure for this call in the *ret* variable, and returns a pointer to the start of the argument area, into which data may be placed using the *xdr_add* routines.

rpc_start_reply

uchar **rpc_start_reply*(
RPC_FILE **val*,
ROME_MESSAGE **mp*)

The *rpc_start_reply* routine handles the initial common part of result processing for all remote procedure call returns. *mp* contains a reply received over the *val* data path. The routine locates the corresponding request based on the transaction identifier and checks the returned data for error codes. If the reply represents a valid return from the remote procedure call, the routine returns a pointer to the start of the marshalled arguments, which can be extracted using the *xdr_get* routines. If there is any error in the reply, the routine returns *NULL.*

xdr_add_binary

uchar *xdr_add_binary(
uchar *where,
uchar *what,
uint len)

The *xdr_add_binary* routine adds *len* bytes of data starting at *what* into the argument structure of the remote procedure call, starting at *where.* The returned value is the pointer position after the end of the argument.

xdr_add_int

uchar *xdr_add_int(
uchar *where,
uint what)

The *xdr_add_int* routine adds the 32-bit integer *what* into the argument structure of the remote procedure call, starting at *where.* The returned value is the pointer position after the end of the argument.

xdr_add_string

uchar **xdr_add_string*(
uchar **where,*
char **value*)

The *xdr_add_string* macro adds the *NUL*-terminated string value to the argument list at the position specified by *where.* The macro calls *xdr_add_binary* for the length of the string, and returns the pointer to the next character position in the argument list.

xdr_get_binary

uchar **xdr_get_binary*(
  uchar **where*,
  uchar **to*,
  uint **len*)

The *xdr_get_binary* routine extracts a sequence of binary octets from the marshalled arguments, starting at *where*, into the array pointed to by *to*. *len* is set to the number of bytes in the array, and the routine returns a pointer to the next value in the argument list.

xdr_get_int

uchar **xdr_get_int*(
  uchar **where*,
  uint **what*)

The *xdr_get_int* routine extracts a 32-bit integer from the marshalled arguments, starting at *where*, into the value *what*. The routine returns a pointer to the next value in the argument list.

xdr_get_string

uchar **xdr_get_string*(
  uchar **where*,
  uchar **to*)

The *xdr_get_string* routine extracts character string from the marshalled arguments, starting at *where*, into the value *to*, which is terminsed by a *NUL*. The routine returns a pointer to the next value in the argument list.

xdr_skip_binary

uchar **xdr_skip_binary*(
  uchar **where*)

The *xdr_skip_binary* routine ignores a sequence of binary octets in the marshalled arguments, starting at *where*. The routine returns a pointer to the next value in the argument list.