Introduction to TI-RPC 2

This chapter provides an overview of TI-RPC, also known as Sun RPC. The information presented is most useful to someone new to RPC. (See also "Glossary" on page 331 for the definition of the terms used in this guide.)

What Is TI-RPC	*page 9*
TI-RPC Issues	*page 11*
Overview of Interface Routines	*page 13*
Network Selection	*page 16*
Transport Selection	*page 17*
Address Lookup Services	*page 18*

What Is TI-RPC

TI-RPC is a powerful technique for constructing distributed, client-server based applications. It is based on extending the notion of conventional, or local procedure calling, so that the called procedure need not exist in the same address space as the calling procedure. The two processes my be on the same system, or they may be on different systems with a network connecting them.

By using RPC, programmers of distributed applications avoid the details of the interface with the network. The transport independence of RPC isolates the application from the physical and logical elements of the data communications mechanism and allows the application to use a variety of transports.

2

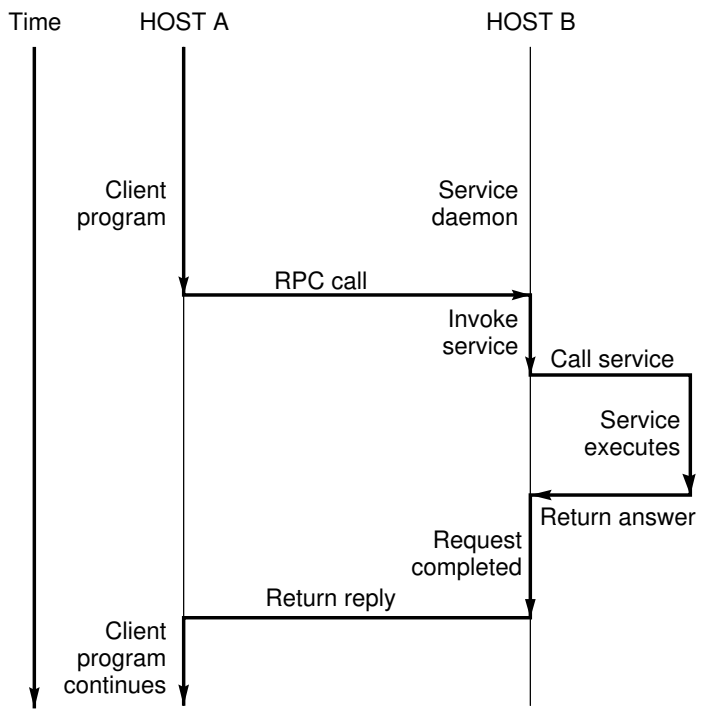

Figure 2-1 How RPC Works

An RPC is analogous to a function call. Like a function call, when an RPC is made, the calling arguments are passed to the remote procedure and the caller waits for a response to be returned from the remote procedure.

Figure 2-1 shows the flow of activity that takes place during an RPC call between two networked systems. The client makes a procedure call that sends a request to the server and waits. The thread is blocked from processing until either a reply is received, or it times out. When the request arrives, the server calls a dispatch routine that performs the requested service, and sends the reply to the client. After the RPC call is completed, the client program continues.

RPC specifically supports network applications. TI- RPC runs on available networking mechanisms such as TCP/IP. Other RPC standards are OSF DCE (based on Apollo's NCS system), Xerox Courier, and Netwise.

TI-RPC Issues

There are a number of issues that help to characterize a particular RPC implementation.

- How are parameters and results passed?
- How is binding carried out?
- How are transport protocols dealt with?
- What are the call semantics?
- What data representation is used?

Parameter Passing

TI-RPC allows a single parameter to be passed from client to server. If more than one parameter is required, the components can be combined into a structure which is counted as a single element. Information passed from server to client is passed as the function's return value. It is not possible to pass information back from server to client through the parameter list.

Binding

The client must know how to contact the service it is interested in. The two aspects are finding out which host the server is on, and then connecting to the actual server process. On each host, a service called `rpcbind` manages RPC services. TI-RPC uses the available host-naming services, such as the `hosts` file, NIS+, and DNS, to locate a host.

Transport Protocol

The transport protocol specifies how the call message and the reply message are transmitted between client and server. TS-RPC used TCP and UDP as transport protocols, but the current version of TI-RPC is transport independent; that is, it works with any transport protocol supported by Solaris 2.x.

Call Semantics

Call semantics has to do with what the client can assume about the execution of the remote procedure, in particular, how many times the procedure was executed. This is important in dealing with error conditions. The three alternatives are *exactly once*, *at most once*, and *at least once*. ONC+ provides *at least once* semantics. Procedures called remotely are *idempotent*: they should return the same result each time they are called, even if it is several times.

Data Representation

Data representation describes the format used for parameters and results as they are passed between processes. For RPC to function on a variety of system architectures requires a standard data representation. TI-RPC uses external data representation (XDR). XDR is a machine-independent data description and encoding protocol. Using XDR, RPC can handle arbitrary data structures, regardless of different hosts' byte orders or structure layout conventions. For a detailed discussion of XDR, see Appendix C, "XDR Protocol Specification," and Appendix A, "XDR Technical Note."

Program, Version, and Procedure Numbers

A remote procedure is uniquely identified by the triple:

(*program number, version number, procedure number*)

The *program* number identifies a group of related remote procedures, each of which has a unique procedure number.

A program may consist of one or more *versions*. Each version consists of a collection of procedures which are available to be called remotely. Version numbers enable multiple versions of an RPC protocol to be available simultaneously.

Each version contains a a number of procedures that can be called remotely. Each procedure has a *procedure* number.

"Program and Procedure Numbers" on page 210," lists the range of values and their significance and tells you how to have a program number assigned to your RPC program. A list of mappings of RPC service name to program number is available in the `rpc` netowrk database, `/etc/rpc`.

Overview of Interface Routines

RPC has multiple levels of application interface to its services. These levels provide different degrees of control balanced with different amounts of interface code to implement. In order of increasing control and complexity. This section gives a summary of the routines available at each level.

Simplified Interface Routines

The simplified interfaces are used to make remote procedure calls to routines on other machines, and specify only the type of transport to use. The routines at this level are used for most applications. Descriptions and code samples can be found in the section, "Simplified Interface" on page 60.

Table 2-1 RPC Routines—Simplified Level

Routine	Function
`rpc_reg()`	Registers a procedure as an RPC program on all transports of the specified type.
`rpc_call()`	Remote calls the specified procedure on the specified remote host.
`rpc_broadcast()`	Broadcasts a call message across all transports of the specified type.

Standard Interface Routines

The standard interfaces are divided into *top level*, *intermediate level*, *expert level*, and *bottom level*. These interfaces give a developer much greater control over communication parameters such as the transport being used, how long to wait beforeresponding to errors and retransmitting requests, and so on.

Top Level Routines

At the top level, the interface is still simple, but the program has to create a client handle before making a call or create a server handle before receiving calls. If you want the application to run on all transports, use this interface. Use of these routines and code samples can be found in "Top Level Interface" on page 68.

Table 2-2 RPC Routines—Top Level

Routine	Description
`clnt_create()`	Generic client creation. The program tells `clnt_create()` where the server is located and the type of transport to use.
`clnt_create_timed()`	Similar to `clnt_create()` but lets the programmer specify the maximum time allowed for each type of transport tried during the creation attempt.
`svc_create()`	Creates server handles for all transports of the specified type. The program tells `svc_create()` which dispatch function to use.
`clnt_call()`	Client calls a procedure to send a request to the server.

Intermediate Level Routines

The intermediate level interface of RPC lets you control details. Programs written at these lower levels are more complicated but run more efficiently. The intermediate level enables you to specify the transport to use. Use of these routines and code samples can be found in "Intermediate Level Interface" on page 73.

Table 2-3 RPC Routines—Intermediate Level

Routine	Description
`clnt_tp_create()`	Creates a client handle for the specified transport.
`clnt_tp_create_timed()`	Similar to `clnt_tp_create()` but lets the programmer specify the maximum time allowed.
`svc_tp_create()`	Creates a server handle for the specified transport.
`clnt_call()`	Client calls a procedure to send a request to the server.

Expert Level Routines

The expert level contains a larger set of routines with which to specify transport-related parameters. Use of these routines and code samples can be found in“Expert Level Interface” on page 75.

Table 2-4 RPC Routines—Expert Level

Routine	Description
`clnt_tli_create()`	Creates a client handle for the specified transport.
`svc_tli_create()`	Creates a server handle for the specified transport.
`rpcb_set()`	Calls `rpcbind` to set a map between an RPC service and a network address.
`rpcb_unset()`	Deletes a mapping set by `rpcb_set`.
`rpcb_getaddr()`	Calls rpcbind to get the transport addresses of specified RPC services.
`svc_reg()`	Associates the specified program and version number pair with the specified dispatch routine.
`svc_unreg()`	Deletes an association set by `svc_reg`.
`clnt_call()`	Client calls a procedure to send a request to the server.

Bottom Level Routines

The bottom level contains routines used for full control of transport options. “Bottom Level Interface” on page 80, describes these routines.

Table 2-5 RPC Routines—Bottom Level

Routine	Description
`clnt_dg_create()`	Creates an RPC client handle for the specified remote program, using a connectionless transport.
`svc_dg_create()`	Creates an RPC server handle, using a connectionless transport.
`clnt_vc_create()`	Creates an RPC client handle for the specified remote program, using a connection-oriented transport.
`svc_vc_create()`	Creates an RPC server handle, using a connection-oriented transport.
`clnt_call()`	Client calls a procedure to send a request to the server.

2

Network Selection

You can write programs to run on a specific transport or transport type, or to operate on a system- or user-chosen transport. There are two mechanisms for network selection, the `/etc/netconfig` database and the environmental variable `NETPATH`. These mechanisms allow a fine degree of control over network selection: a user can specify a preferred transport, and if it can, an application uses it. If the specified transport is inappropriate, the application automatically tries others with the right characteristics.

`/etc/netconfig` lists the transports available to the host and identifies them by type. `NETPATH` is optional and allows a user to specify a transport or selection of transports from the list in `/etc/netconfig`. By setting the `NETPATH`, the user specifies the order in which the application tries the available transports. If `NETPATH` is not set, the system defaults to all visible transports specified in `/etc/netconfig`, in the order they appear in that file.

For more details on network selection, refer to the *Transport Interfaces Programming Guide* or see the `getnetconfig(3N)` and `netconfig`(4) manpages.

RPC divides selectable transports into the following types:

Table 2-6 `nettype` Parameters

Value	Meaning
`NULL`	Same as selecting `netpath`.
`visible`	Uses the transports chosen with the visible flag ('v') set in their `/etc/netconfig` entries.
`circuit_v`	Same as `visible`, but restricted to connection-oriented transports. Transports are selected in the order listed in `/etc/netconfig`.
`datagram_v`	Same as `visible`, but restricted to connectionless transports.
`circuit_n`	Uses the connection-oriented transports chosen in the order defined in `NETPATH`.
`datagram_n`	Uses the connectionless transports chosen in the order defined in `NETPATH`.
`udp`	Specifies Internet user datagram protocol (UDP).
`tcp`	Specifies Internet transport control protocol (TCP).

Transport Selection

RPC services are supported on both circuit-oriented and datagram transports. The selection of the transport depends on the requirements of the application.

A datagram transport is the transport of choice if the application has all of the following characteristics:

- Calls to the procedures do not change the state of the procedure or of associated data.
- The size of both the arguments and results is smaller than the transport packet size.
- The server is required to handle hundreds of clients. A datagram server does not keep any state data on clients, so it can potentially handle many clients. A circuit-oriented server keeps state data on each open client connection, so the number of clients is limited by the host resources.

A circuit-oriented transport is the transport of choice if the application has any of the following characteristics:

- The application can tolerate or amortize the higher cost of connection setup compared to datagram transports.
- Calls to the procedures can change the state of the procedure or of associated data.
- The size of either the arguments or the results exceed the maximum size of a datagram packet.

Name-to-Address Translation

Each transport has an associated set of routines that translate between universal network addresses (string representations of transport addresses) and the local address representation. These universal addresses are passed around within the RPC system (for example, between `rpcbind` and a client). A run-time linkable library that contains the name-to-address translation routines is associated with each transport. Table 2-7 shows the main translation routines.

For more details on these routines, see the `netdir(3N)` manpage and the *Transport Interfaces Programming Guide*. Note that the `netconfig` structure in each case provides the context for name-to-address translations.

Table 2-7 Name-to-Address Translation Routines

`netdir_getbyname`	Translates from host/service pairs (e.g. `server1`, `rpcbind`) and a `netconfig` structure to a set of `netbuf` addresses. `netbuf`s are Transport Level Interface (TLI) structures that contain transport-specific addresses at run-time.
`netdir_getbyaddr`	Translates from `netbuf` addresses and a `netconfig` structure to host/service pairs.
`uaddr2taddr`	Translates from universal addresses and a `netconfig` structure to `netbuf` addresses.
`taddr2uaddr`	Translates from `netbuf` addresses and a `netconfig` structure to universal addresses.

Address Lookup Services

Transport services do not provide address-lookup services. They provide only message transfer across a network. A client program needs a way to obtain the address of its server program. In earlier system releases this service was performed by `portmap`. `rpcbind` replaces the `portmap` utility.

RPC makes no assumption about the structure of a network address. It deals with universal addresses specified only as null-terminated strings of `ASCII` characters. RPC translates universal addresses into local transport addresses by using routines specific to the transport. For more details on these routines, see the `netdir(3N)` and `rpcbind(3N)` manpages.

`rpcbind` provides the operations:

- Add a registration
- Delete a registration
- Get address of a specified program number, version number, and transport
- Get the complete registration list
- Perform a remote call for a client
- Return the time

Registering Addresses

`rpcbind` maps RPC services to their addresses, so its address must be known. The name-to-address translation routines must reserve a known address for each type of transport used. For example, in the Internet domain, `rpcbind` has port number 111 on both TCP and UDP. When `rpcbind` is started, it registers its location on each of the transports supported by the host. `rpcbind` is the only RPC service that must have a known address.

For each supported transport, `rpcbind` registers the addresses of RPC services and makes the addresses available to clients. A service makes its address available to clients by registering the address with the `rpcbind` daemon. The address of the service is then available to `rpcinfo`(1M) and to programs using library routines named in the `rpcbind(3N)` manpage. No client or server can assume the network address of an RPC service.

Client and server programs and client and server hosts are usually distinct but they need not be. A server program can also be a client program. When one server calls another `rpcbind` server it makes the call as a client.

To find a remote program's address, a client sends an RPC message to a host's `rpcbind` daemon. If the service is on the host, the daemon returns the address in an RPC reply message. The client program can then send RPC messages to the server's address. (A client program can minimize its calls to `rpcbind` by storing the network addresses of recently called remote programs.)

The `RPCBPROC_CALLIT` procedure of `rpcbind` lets a client make a remote procedure call without knowing the address of the server. The client passes the target procedure's program number, version number, procedure number, and calling arguments in an RPC call message. `rpcbind` looks up the target procedure's address in the address map and sends an RPC call message, including the arguments received from the client, to the target procedure.

When the target procedure returns results, `RPCBPROC_CALLIT` passes them to the client program. It also returns the target procedure's universal address so that the client can later call it directly.

The RPC library provides an interface to all `rpcbind` procedures. Some of the RPC library procedures also call `rpcbind` automatically for client and server programs. For details, see Appendix B, "RPC Protocol and Language Specification."

2

Reporting RPC Information

`rpcinfo` is a utility that reports current RPC information registered with `rpcbind`. `rpcinfo` (with either `rpcbind` or the `portmap` utility) reports the universal addresses and the transports for all registered RPC services on a specified host. It can call a specific version of a specific program on a specific host and report whether a response is received. It can also delete registrations. For details, see the `rpcinfo(1M)` manpage.