CS587: Simple RPC System

Prof. Patrick G. Bridges

Due: November 16, 2015, 5:00pm

Introduction

For this assignment, you will create a C runtime library for supporting simple remote procedure calls (SRPCs). The SRPC runtime must use UDP datagrams to communicate between clients and servers, support multiple clients and servers, a single server must be able to handle multiple clients, and a single client must handle multiple servers. Ther parameters for an individual RPC will fit into a single UDP datagram, so you don't need to worry about fragmentation.

SPRC Runtime API

The SRPC Runtime library implements 11 functions and 1 data structure that are used to execute RPCs:

- Initialization – `Srpc_ClientInit()`, `Srpc_ServerInit()`
- Argument Marshalling – `struct Srpc_Arg`, `Srpc_FreeArgs()`
- RPC Binding – `Srpc_Bind()`
- RPC Invocation – `Srpc_Call()`
- Cleanup/Shutdown – `Srpc_ClientExit()`, `Srpc_ServerExit()`
- Server Routines – `Srpc_Export()`,`Srpc_Server()`
- Miscellaneous – `Srpc_StatusMsg()`

Return Codes and Failures

Every SRPC routine returns an SRPC status code as defined in the attached file `srpc.h`. A server responds to client failure by cleaning up any state associated with the client. A client responds to server failure by returning `SRPC_ERR_TIMEOUT` if a server reply is never received. If a server reboots, then `SRPC_ERR_REBOOT` is returned. In either case, the server must ensure that it does not process requests sent before the failure, thus providing "at most once" semantics, in that an RPC is executed at most once on a server when there is a failure.

Initialization Routines

Two initialization routines are provided, one for clients and one for servers. It is possible for a single process to be both a client and a server simultaneously.

Routine

```
Srpc_Status
Srpc_ClientInit(
                unsigned int timeout, // milliseconds before retransmitting
                unsigned int retries); // max # retries before SRPC_TIMEOUT
```

Description

Initializes the client side of the SRPC system.

Returns

```
SRPC_ERR_OK
SRPC_ERR_ALREADY_INITIALIZED
```

Routine

```
Srpc_Status
Srpc_ServerInit(
                unsigned short port, // port on which to listen
                unsigned int timeout, // milliseconds before retransmitting
                unsigned int retries); // max # retries before SRPC_TIMEOUT
```

Description

Initializes the SRPC server side to receive SRPC requests on the indicated port.

Returns

```
SRPC_ERR_OK
SRPC_ERR_ALREADY_INITIALIZED
SRPC_ERR_PORT_INUSE
```

Marshalling Arguments

Arguments must be marshalled and unmarshalled to enable transmission using UDP datagrams, and to allow computers with different data formats to communicate. This marshalling and unmarshalling is usually done using stubs that are automatically generated, but the SRPC system avoids this complication by making the calling routine specify the types of the arguments to an RPC. When invoking an RPC the caller passes an array of Srpc_Arg data structures, one for each argument. The format of the Srpc_Arg is as follows:

```
typedef struct Srpc_Arg {
    Srpc_Type type; // Type of argument
    unsigned int size; // Size of arg, in bytes.
    void *value; // Value
} Srpc_Arg;
```

Srpc_Type is one of:

`SRPC_TYPE_NONE` – denotes end of arg array

`SRPC_TYPE_INT` – value field contains an integer (valid sizes 1,2,4)

`SRPC_TYPE_DATA` – value field contains a pointer to an array of bytes

The total size of the arguments must be less than `SRPC_MAX_ARG_SIZE`. This is defined to be small enough that the marshalled arguments plus any reasonable amount of RPC protocol headers will fit into a single UDP datagram.

Routine

```
void Srpc_FreeArgs(Srpc_Arg args[]); // Arg array to be freed.
```

Description

Frees all memory used by the args array. All value fields of type `SRPC_TYPE_DATA` are assumed to contain a memory address and are freed. The memory used by the args array itself is also freed.

RPC Binding

Routine

```
Srpc_Status
Srpc_Bind(
          char *function, // Name of function to which to bind
          char *server, // Server's DNS name
          unsigned short port, // Server's port
          Srpc_Handle *handle) // (Out) handle for Srpc_Call
```

Description

Used to bind to a particular function on a particular SRPC server. A handle is returned that is used in subsequent calls to Srpc_Call.

Returns

`SRPC_ERR_OK`

`SRPC_ERR_UNITIALIZED`

`SRPC_ERR_ALREADY_BOUND`

`SRPC_ERR_NO_FUNCTION` – server does not export function

`SRPC_ERR_NO_SERVER` – server not found

`SRPC_ERR_TIMEOUT` – no response from server

`SRPC_ERR_REBOOT` – server rebooted

Rpc Invocation

Routine

```
Srpc_Status
Srpc_Call(
          Srpc_Handle handle, // Returned by Srpc_Bind
          Srpc_Arg inArgs[], // Input arguments
          Srpc_Status *status, // (Out) RPC status from server
          Srpc_Arg **outArgs) // (Out) Output arguments
```

Description

Used to invoke an RPC on the server. handle was previously returned by Srpc_Bind. The caller must eventually call Srpc_FreeArgs on outArgs.

Returns

`SRPC_ERR_OK`

`SRPC_ERR_UNITIALIZED`

`SRPC_ERR_ARGS_TOO_BIG`

`SRPC_ERR_INVALID_ARG_TYPE`

`SRPC_ERR_TIMEOUT` – no reply

`SRPC_ERR_REBOOT` – server rebooted

Server Routines

Routine

```
Srpc_Status
Srpc_Export(
            char *name, // Name to export
            Srpc_Function *function, // Function to invoke
            void *functionData); // Opaque data passed to function
typedef Srpc_Status (Srpc_Function)(
                                    void *functionData, // Opaque data passed by Srpc_Export
                                    Srpc_Arg inArgs[], // Input args
                                    Srpc_Arg **outArgs); // (Out) Output args
```

Description

Srpc_Export is used by the server to export a function to the clients. The function is of type Srpc_Function, as shown above. The function returns its output arguments, if any, in outArgs. The RPC system must eventually call Srpc_FreeArgs on the value returned in outArgs.

Returns

`SRPC_ERR_OK`

`SRPC_ERR_UNITIALIZED`

`SRPC_ERR_ALREADY_EXPORTED`

Routine

```
Srpc_Status
Srpc_Server(void)
```

Description

Begins servicing SRPC requests from the clients. Does not return until Srpc_ServerExit is called.

Returns

`SRPC_ERR_OK`

`SRPC_ERR_UNITIALIZED`

Cleanup

Routine

```
Srpc_Status
Srpc_ClientExit(void)
Srpc_Status
Srpc_ServerExit(void)
```

Description

Cleans up and shuts down the RPC system.

Returns

`SRPC_ERR_OK`

`SRPC_ERR_UNINITIALIZED`

Misc.

Routine

```
char *
Srpc_StatusMsg(Srpc_Status status)
```

Description

Returns a text message corresponding to the given valid status code, NULL otherwise.

Returns

Text string or NULL (returned text string are read-only and do not need to be `free()`-ed)

Logistics

Turn in the C files that implement your library as a compressed tarfile on UNM Learn, as you did on assignment 1. Along with your source code, you must turn in *must* a `README` file that explains the general structure of your source code. You must also provide a `Makefile` that compiles your source code into library file named `libsrpc.a` *without warnings.* Programs should be able to compile against your library by linking to the *unmodified* `srpc.h` file provided and linking against your library by adding '-lsrpc' to their compilation command.

A sample implementation of the library to experiment with as well as testcases to use to test your library implementation can be found in /nfs/faculty/bridges/public/cs587/libsrpc.a and /nfs/faculty/bridges/public/cs587/srpc-testcases.

Credits

Thanks to John Hartman and Mohan Rajagopolan at the University of Arizona for materials (including the assignment outline and include file) and advice related to this assignment.