University of Pittsburgh

Department of Computer Science

Project report

CS 2510 Computer Operating Systems

October 2015

Simple Remote Procedure Call (SRPC)

Mohammad Hasanzadeh Mofrad

hasanzadeh@cs.pitt.edu

Abstract

The Remote Procedure Call (RPC) is a technology for creating distributed client/server applications. As an interprocess communication technique, RPC enables client program to request a service from server program that is located in another machine from another network. In this report, the systematic and implementation details of a Simple RPC (SRPC) is discussed. The SRPC is armed with a directory service which is responsible for storing servers' endpoint addresses and verifying clients' endpoint queries. The directory service act as a central information repository for servers and clients while addressing the scalability issue of client/server architecture. The current SRPC implementation utilizes a well-defined message passing convention for parameter marshaling along with upper and lower levels data structures for efficient integration of various logical components of SRPC. The SRPC uses a simple 2 steps register/resolve/verify/check processes for server registration, server resolution, client verification and client checking. Also, it implements a common marshaling interface that is purely data type driven and optimizes the buffer size and number of send/receive attempts.

Keywords: Client/Server architecture, Remote Procedure Call (RPC)

1 Introduction

Remote Procedure Call (RPC) is a communication tool for executing a local procedure in a remote address space. The common metaphor of RPC is to make a function call to one remote machine without explicitly letting the user know it is a remote function call. Thus, the RPCs are much like Local Procedure Calls (LPCs) to users except for their implicit nature that is almost hidden from user programs.

The Simple Remote Procedure Call (SRPC) is a limited implementation of legacy RPC optimized for integer and character data types. The current SRPC implementation is combined with a directory service component for addressing the scalability issues of client/server architecture. Figure 1 shows the SRPC framework with its ordered message passing mechanism.

As shown in Figure 1, server contains two parts including *server application* which executes the local function call and *server stub* which receives the incoming function calls. Also, client contains two parts including *client application* which makes the remote function call and *client stub* which sends the function call to server stub. Moreover, directory service is responsible for server registration and resolution for server and client, respectively.

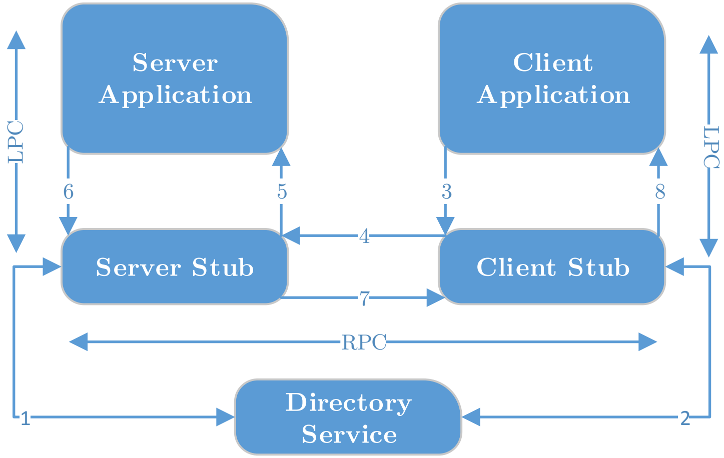

Figure 1 SRPC framework.

The rest of this report is organized as follows: Section 2 introduces the SRPC framework. Section 3 discusses about the process of stub generation. Section 4 presents the compilation details. Finally, Section 6 concludes this report.

2 Simple Remote Procedure Call (SRPC)

The message passing conventions and design issues of SRPC will be introduced in this section. This encompasses a detailed description of each SRPC's components including directory service, server stub and client stub. Figure 2 is the main diagram of message passing sequence of SRPC. This figure will be discussed in the following section.

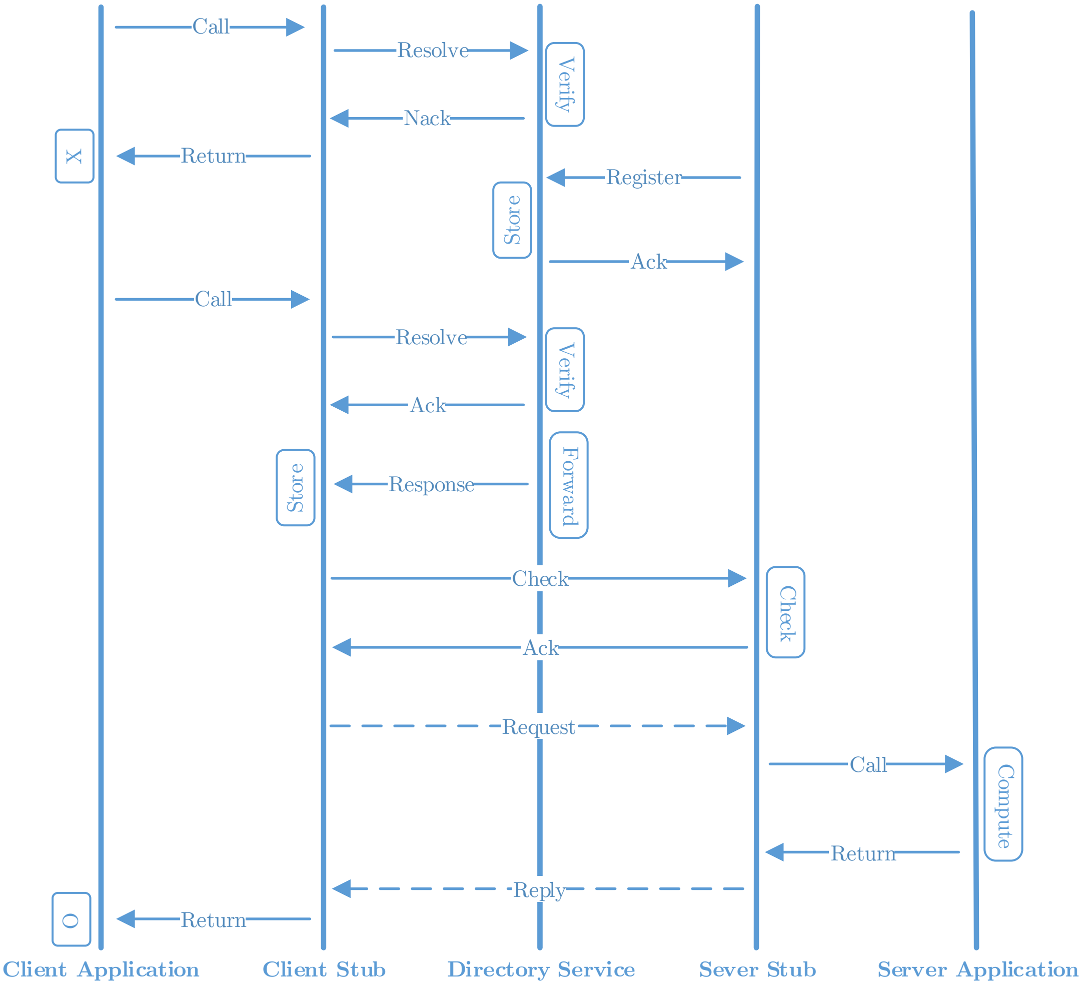

Figure 2 Request/response flow of SRPC.

2.1 Message Passing Flow

An overall request/response flow of SRPC with regard to Figure 2 is as follows:

1. After booting up, directory service waits for server registration or client resolution.
2. Serve stub registers its program to directory service.
3. Directory service stores program description along with server endpoint address.
4. Client application executes a function as a LPC without knowing it is a RPC.
5. Client stub receives the function call.
6. The client stub send a resolve request to directory service endpoint.
7. If a server with the same program version has been registered to the directory service, the directory service provides the client stub with its endpoint address.
8. Client stub marshals the procedure ID for checking in server stub.
9. After the checking is done, the server stub is ready to receive incoming parameters from client stub.
10. Client stub sends the parameters' buffer to server stub and server stub unmarshals the parameters.

11. Server stub initiates a LPC to server application for the received procedure ID and its corresponding parameters.
12. The server application executes the procedure and returns the results to the server stub.
13. The server stub packs the results into the buffer and marshals them to the client stub.
14. The client stub receives the buffer and unmarshals the results.
15. The client stub returns the results to the client application.

2.2 Upper Level Data Structures

The SRPC contains 4 logical components:

1. *Directory service* a.k.a directory which provides the directory service functionalities for client and server stubs.
2. *Server stub* a.k.a server which is the core component for handling client requests. It receives the computing requests of client stub, unmarshals them and make a LPC to server application for computing and sending back the results to the client stub.
3. *Client stub* a.k.a client which receives the user requests and marshals them to the server stub for further computation. Also, the client receives the computation results from the server, unmarshals them and returns them to user.
4. *Client* a.k.a user which generates the computing requests to the client stub.

In addition to these components, SRPC has a stub generation component that automates the process of reading input files and creating the directory, server and client components' output files.

2.2.1 Service Data Structure

Service struct (Figure 3) is a unified data structure that is used for storing the directory service and server stub information such as their IP address, port number and type ({Directory service: directory, Server stub: server}). Also, for server stub the program id, version and name is stored in this data structure for further usage. The *create*() procedure is responsible for inserting the directory and server information into the service data structure.

Service	Program id	char*
	Program version	char*
	Program name	char*
	IP address	char*
	Port number	char*
	Service type	char*

Figure 3 Service data structure for server and directory service.

2.2.2 Service Configuration Buffer

The configuration buffer is a common agreement between directory, server and client to send their endpoints' information along with service description (in the case of server). Please note that, before sending the configuration buffer, it is necessary to provide the calling party with the length of buffer. Figure 4 describes the buffer details.

Configuration buffer
Buffer length
A number
int

A

Configuration buffer				
IP address	Port number	Program id	Program version	Program name
XXX.XXX.XXX.XXX	1 – 65535	String Literal	String literal	String literal
char array[ ]				

B

Figure 4 Service configuration buffer. (A) The buffer length and (B) The configuration buffer.

2.3 Design Issues

Particular issues should be addressed while designing a client-server architecture that handles RPC. Natively the concept of RPC does not suggest any proper solution for *data marshaling/unmarshaling*, so while writing a RPC the data marshaling becomes the first challenge. The process of marshaling data over the network is carried out by utilizing TCP sockets. The send/receive buffers between client/server components have corresponding buffer sizes that are characterized in compile time with respect to procedure's parameters.

In SRPC, each component aims to fulfill its role in the most intelligence and efficient way. So, in addition to using a universal convention for data marshaling, all components use a same data structure for storing information about each other (Figure 3). By just reusing the concept of this data structure for different tasks in different components, this framework mitigates the complexity of request and response messages.

2.4 Directory Service Component

This component acts as the service registration gateway for servers and service resolution gateway for clients. It has a main loop for detecting the request type and storing/forwarding information from server/to client components. Figure 5 shows the pseudocode of directory component.

There is a common agreement between directory, servers and clients for register/resolve request and replies. Having a look at Figure 5, since directory service processes the server and client requests at the same time, there should be a mechanism to handle these requests separately. Also, the directory provides server and clients with an acknowledge message regarding their successful query. Moreover, in the case of a successful server resolution from the client, the directory should send back the server endpoint address to the client. Figure 6 and Figure 7 Show the prototype for handling the request/reply messages. Please note that both client and server components have equipped with proper procedures to handle the responses from the directory.

2.4.1 Multiple Server Registration

It is worth noting that, the challenge of registering multiple servers at a single directory is addressed in the implementation of this component by elaborating the service data structure (Figure 3) to an array of services for different servers. Also, directory could remove server instances by calling *remove*() in case of receiving a service withdrawal request.

Directory service pseudocode

```
create(directory)
while(True)
    if(There is a connection)
        if(server registration) // Server request
            if(verify(server))
                create(server)
                acknowledge(server) // Server response
            else
                nacknowledge(server) // Server response
        else if(server withdrawal) // Server request
                remove(server)
        else if(server resolution) // Client request
            create(client)
            if(verify(client))
                acknowledge(client) // Client response
                forward(client) // Client response
            else
                nacknowledge(client) // Client response
        else
            nacknowledge() // Invalid request
```

Figure 5 Directory service pseudocode.

Register/resolve request
Request type
{0, 1}
int

A

Register/resolve request
Request length
A number
int

B

Register/resolve request
Request parameters
String literal
char[ ]

C

Figure 6 Register/resolve requests from server/client to directory. (A) The request type is from the set of {0: server, 1:client}, (B) The request length is the length of request parameters and (C) The string that contains the server or client information with regard to Figure 3.

Register/resolve response
Response type
{0, 1}
int

A

Register/resolve response
Response length
A number
int

B

Register/resolve response
Response parameters
String literal
char[ ]

C

Figure 7 Register/resolve responses from directory to server/client. (A) The response type is from the set of {0: ack, 1:nack} client/server request, (B) The response length is the length of response parameters for client and (C) The string that contains the server information with regard to Figure 3.

2.4.2 Server and Client Program Verification

From Figure 5, the directory equipped with a procedure called *verify*(). Since multiple servers may register themselves under the directory service stack, after receiving a client request directory checks the program version along with its ID and name to return the compatible server address to the client stub. If the directory cannot successfully checks the client request against the current registered servers, it will return "1" as an error notifying the service description is wrong or the server is not existed yet. Since we don't have multiple instances of the same program, the process of finding the requested server for a client request includes a simple linear search algorithm. Therefore, the first hit will definitely be the requested server.

2.4.3 Service list

For ease of use, it is possible to initiate a *print*() command to list all registered servers under a directory service. This command will send a request to the main loop of directory service and leads to print the registered services.

2.5 Server Stub Component

Server stub is responsible for computation part of SRPC. It receives computing requests from client, computes them and sends them back to the client. The pseudocode of server stub is shown in Figure 8. In this figure, the *create*() procedure follows the service data structure (Section 2.2.1) for storing the directory and server endpoint addresses and specifications. Furthermore, calling a *procedure*() function, is analogous to calling the corresponding procedure of server application, e.g., if the client stub asks for maximum procedure, server stub will call this procedure through calling the server application.

```
Server stub pseudocode
create(server)
create(directory)
if(register(server))
    while(True)
        if(There is a connection)
            if(Request for computation) // Client request
                if(procedure = check(compute ID))
                    receive(compute parameters)
                    procedure(parameters) // Compute
                    send(results) // Server response
            else if (Request for withdrawal)
                withdraw(server) // Server request
            else
                continue // Invalid request
else
    exit(1) // Directory service is not available
```

Figure 8 Server stub pseudocode.

2.5.1 Service Registration

In order to advertise its service, the server needs to register its program to the directory service. The *register*() procedure handles this process. This function sends a message to the directory which contains the IP address, Port number and program description of the current service. This message follows the format of Figure 4. Please note that before sending this message, its length has already sent to the requested endpoint. Upon successful service registration, server waits for client computing requests in its main loop.

2.5.2 Service Withdrawal

There is a facility for server components to deliberately withdraw their own instances. The process is done by initiating the *withdraw*() command that sends a message to directory service regarding withdrawing the server instance. After receiving this message, the directory will update its service list for future client's requests.

2.5.3 Service Computation

Before sending the procedure's parameters, client should send the procedure ID, for the verification step. So, the server checks the ID of requested procedure against it's current ones and if the same procedure ID is existed in sever, it waits for receiving the computation buffer which contains the procedure's parameters. After receiving the procedure ID, depending on what procedure the client invokes, the corresponding parameters will marshal to the server stub from client stub. Figure 9 shows the initial compute request from the client to specify which procedure it wants. After that, depending on the function invocation from the client, the compute parameters will marshal to the client stub. Figure 10 shows the complete list of parameters that pass through SRPC. Different procedures in client or server implement the same interfaces for sending/receiving these data types.

Compute request
procedure ID
A number
int

A

Compute response
Compute procedure
A number
int

B

Figure 9 Client compute request format for computing specific function on server. (A) The client sends the procedure ID to the server (2) Server checks whether it has the invoked procedure or not and send the response ({0: successful, 1: Unsuccessful}) to the client.

Data type	Compute buffer size
char	1 byte
char array[ ]	|array| × 1 bytes
int	4 bytes
int array[ ]	|array| × 4 bytes
int matrix[ ][ ]	|rows| × |columns| × 4 bytes

Figure 10 Primitive data type supported from marshaling. Arrays and matrix are the same as arrays and matrix of pointers.

2.5.4 Common marshaling Interface

Currently, the SRPC supports for some of the primitive data types such as char, char pointer, int, int pointer. In order to have high flexibility, the current SRPC implementation does not stick to the toy procedures (wc (word count), min, max, sort and multiply) and it supports different combination of parameters for marshaling. The whole viewpoint is around that, each data type should have a unique pair of interfaces for marshaling/unmarshaling the computing parameter from/to client/server and vice versa. This common interface is implemented in both client and server for transporting the parameters and results back and forth. The buffer size and number of tries to send the parameters for aforementioned procedures are shown in Figure 11. A detailed discussion about marshaling/unmarshaling of these data types will be given in Section 3.

Toy Functions	Parameter	Marshaling buffer	#Tries
wc	|array| + char array [ ]	(4 + |char array| × 1) bytes	2
min, max & sort	|array| + int array [ ]	(4 + |int array|× 4) bytes	2
multiply	|rows| + |columns| + int array [ ][ ]	(4 + 4 + |int array [ ][ ]|× 4) bytes	3

Figure 11 Common marshaling interface for parameter marshaling.

2.6 Client stub Component

The SRPC's client stub component follows the same paradigm of directory and server components. It uses the service data structure (Figure 3) along with *create*() procedure to store information of directory and server endpoints. Figure 12 is the pseudocode of this component. Since the function invocation always starts from the client application, the client does not have any main function. Also, while calling e.g. the maximum function, user think of a LPC call with some parameters, but actually user initiates a RPC call. Thus, the client completely hides the RPC complexities from the user.

```
Client stub pseudocode
create(directory)
procedure(parameters) // function call
    if(resolve(server))
        create(server)
        send(procedure ID) // Client request
        if(check(procedure ID))
            send(procedure parameters) // Client request
            receive(results) // Server response
            return(results)
        else
            exit(1)
```

Figure 12 Client stub pseudocode.

In Figure 12, after initiating a function call from client a.k.a user, the corresponding procedure will be invoked. At the first point, by calling the *resolve*() procedure, the client stub ask directory whether the corresponding program is available at the directory or not. Then, the directory checks the program description against its registered service list and returns the endpoint address (if it exists) to the client. After receiving the server endpoint's address, the client stub marshals the procedure ID, so that the server checks the ID toward its own procedure IDs. After successful procedure's ID checking, the client stub marshals the procedure parameters using the common marshaling interface (Figure 11) and waits for the results to return back. After receiving the results, the client stub unpacks them and returns them to the user program.

2.6.1 Service Resolution

In the simplest form, the directory act as a stateless server toward clients and does not store the client information. It just provides the server information to it. Moreover, since the client does not have an endpoint address and it should follow the service configuration buffer (Figure 4) to contact directory. The two first parts of service configuration buffer (Figure 4) is filled with the directory endpoint address. After that the requested program description is added to the rest of the buffer. Before sending the service configuration buffer, the *resolve*() function tries to contact directory by sending and initial request that specifies the incoming buffer belongs to the client. At the other side, the directory will be ready for receiving the configuration buffer and if it sees its own address in the configuration buffer, it knows that a client resolve request has just arrived and provides it with server address if available.

3 Stub Generation

The stub generator a.k.a stubgen (*stub_gen*.c) is a simple compiler for SRPC. It reads the program description and endpoints configuration files and generates the client and server stubs. This compiler uses a unique data structure for storing the program specifications and directory and server configurations. The input specification file of stubgen is XML formatted files (*spec.xml* and *conf.xml*) which have the program name, version and ID along with its procedures and procedure's parameters and directory and server addresses and ports. After parsing these files with *ezXML library* (an open source XML parser [1]), the output data is stored in the program, procedure and parameter nested data structures for spec.xml and node data structure for conf.xml.

3.1 Lower Level Data structures

Figure 13 shows the compiler specific data structures. The stubgen compiler utilizes the stored information of these data structures to generate server and client stubs. In addition to filling it with the proper information of spec.xml file, the stubgen also stores some necessary information about input parameters in this data structure in that it will use them while generating the header, client and server stub files.

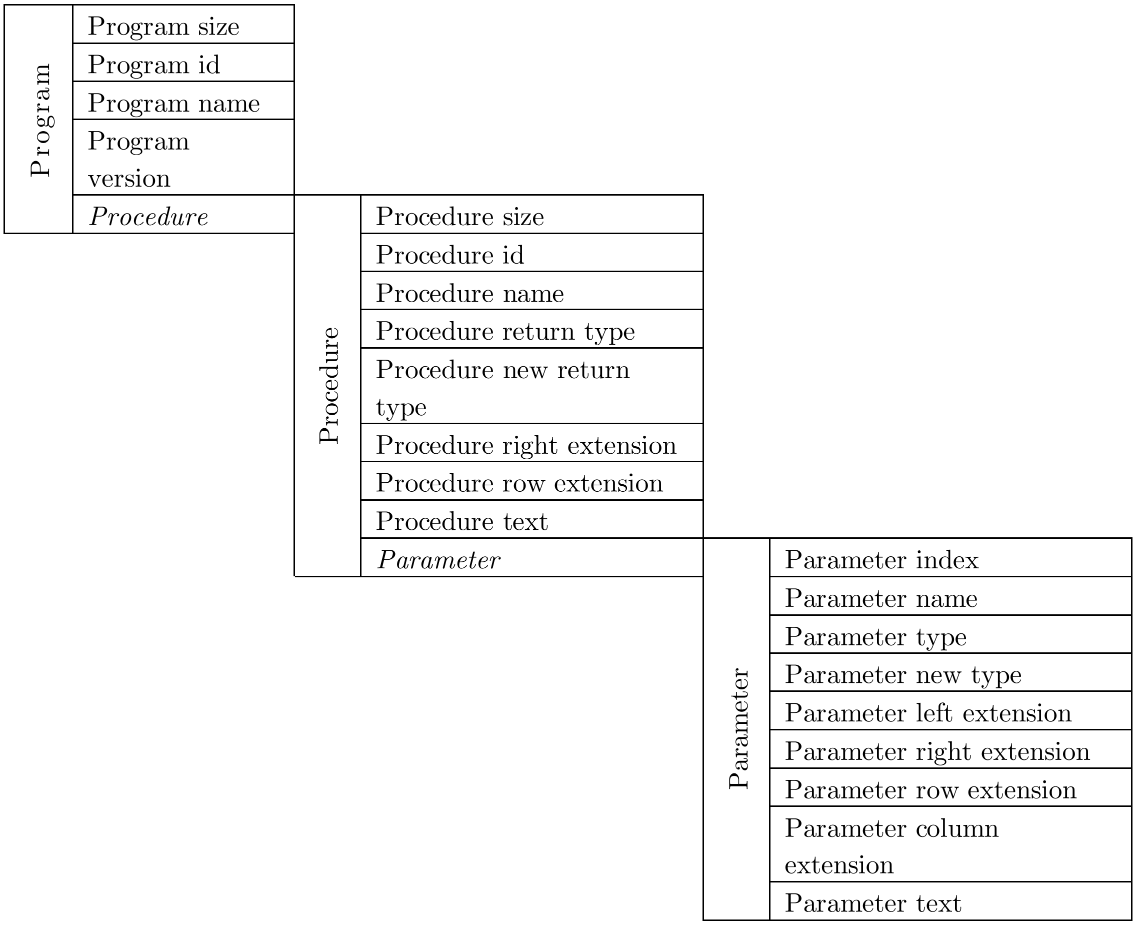

Figure 13 Program, procedure and parameter data structures.

3.2 Data Type Manipulation

As we noted before, the stubgen is able to generate different combinations for input parameters. So, the question is how it can reach to this level of flexibility for generating procedures. The answer to this question lays under the way of filling the program data structure and accurate usage of this data structure. The following is the discussion about this process.

3.2.1 Handling Pointers

In SRPC, although we stick to the integer and character primitive data types which are easy to handle, but we still have challenges ahead. Passing integer pointers (e.g., 1 dimensional array or 2 dimensional array (matrix)) is one of our aims in this project. Since, later we are going to marshal/unmarshal these arrays, we should have their sizes to comply with the introduced common marshaling interface (Figure 11). So, a binding step is proposed for writing the input XML file. By following these binding rules, the pointers will be easily pack/unpack using the common marshaling interface. Figure 14 is the common agreement for specifying the program's return type in the input XML file. The description is pretty straightforward except for int* which should be stated as int_pointer in the input XML file.

C style return type	XML specific return type	Stubgen new return type	Stubgen extensions
empty	empty	void	
void	void	void	
int *	int_pointer	int *	rext = *
int	int	int	

Figure 14 Program specific return type constraints.

Moreover, another convention is proposed for handling the parameter types which is depicted in Figure 15. Following this constraints, stubgen could fill out the missing parts of each variable type by extracting relevant information from the XML file. For example, if a function needs a matrix of integers, there should be a row and column parameter for the matrix in order to characterize its dimensions for marshaling. Furthermore, these two parameters should pass within two underscores "_" after the definition of the int_matrix_, whilst the associated parameters for dimensions should be among the input parameters of procedure.

C style type	XML specific type	Stubgen new type	Stubgen extensions
char	char	char	
char*	char_pointer	char*	lext = *
char[]	char_array	char[]	rext = [ ]
int	int	int	
int*	int_pointer_size	int*	lext = *, rowext = size
int[]	int_array_size	int[size]	rext = [ ], rowext = size
int[][]	int_matrix_size1_size2	int[size1][size2]	rext = [ ][ ], rowext = size1, colext = size2

Figure 15 Parameter specific type constraints.

The last column of Figure 15, states the extension tags of input parameters which are used for generating header and stub files.

3.2.2 Parameter Ordering

There might be a chance that input parameters does not entered in order. As long as the parameters orders are specified by their index values in spec.xml file, we already have the sufficient information about parameter orders. So, by applying a simple sort to input parameters, we ensure that we have the accurate parameter ordering for next steps of stubgen.

3.2.3 Importing Endpoints Configuration

In addition to spec.xml file, to import the specified configuration of directory and server, another file (*conf.xml*) is parsed by the XML parser. The resultant data is stored in the node data structure of Figure 16. This data structure contains the type, IP address and port of available endpoints. Please note that the order of appearance of directory and server nodes in conf.xml is important because stubgen assumes that directory is the first node and server is the second one.

Node	Node type
	Node IP
	Node port

Figure 16 Node data structure.

After storing the corresponding data of input configuration files, the main stream of stubgen for building the header and stubs will be started.

3.4 Templates

The stubgen aims to fully automate the process of building the stubs as much as possible. Applying the notion of creating and cloning templates while generating the stubs could be a superb idea. To support this notion, the directory, server and client are divided into two parts:

1. *The hardcoded part* which is independent of procedure implementation (marshaling) and responsible for handling internal tasks of each component such as sending program acknowledgment or verifications with respect to each component. Therefore, the hardcoded parts of directory, server and client files are stored in directory_t.c, server_t.c and client_t.c files, respectively.
2. *The marshaling part* which generates the dynamic portion of client and server stubs with respect to input spec.xml and conf.xml files. The code snippet of requested procedures are generated within this part for client and server stubs. Also, the directory and server is configured with respect to the information provided by conf.xml.

The idea of utilizing templates, tremendously reduces the code complexity of stubgen by focusing on generating the second part of client and server stubs. Also, Please note that the directory.c file does not subject to change during stubgen process except for updating its endpoint information.

3.3 Generating Output Files

To this point, the two input files of stubgen which are spec.xml and conf.xml are introduced. It's the time to put Legos together and talk about the main stream of stubgen for creating the output directory, server stub and client stub files (directory.c, server.c and client.c). Figure 17 sketches the whole process of sub generation from parsing input files to generating the output .c files. Since we almost covered the first half of Figure 17, the following discussion will be mostly about *create*() and *configure*() functions of this flowcharts.

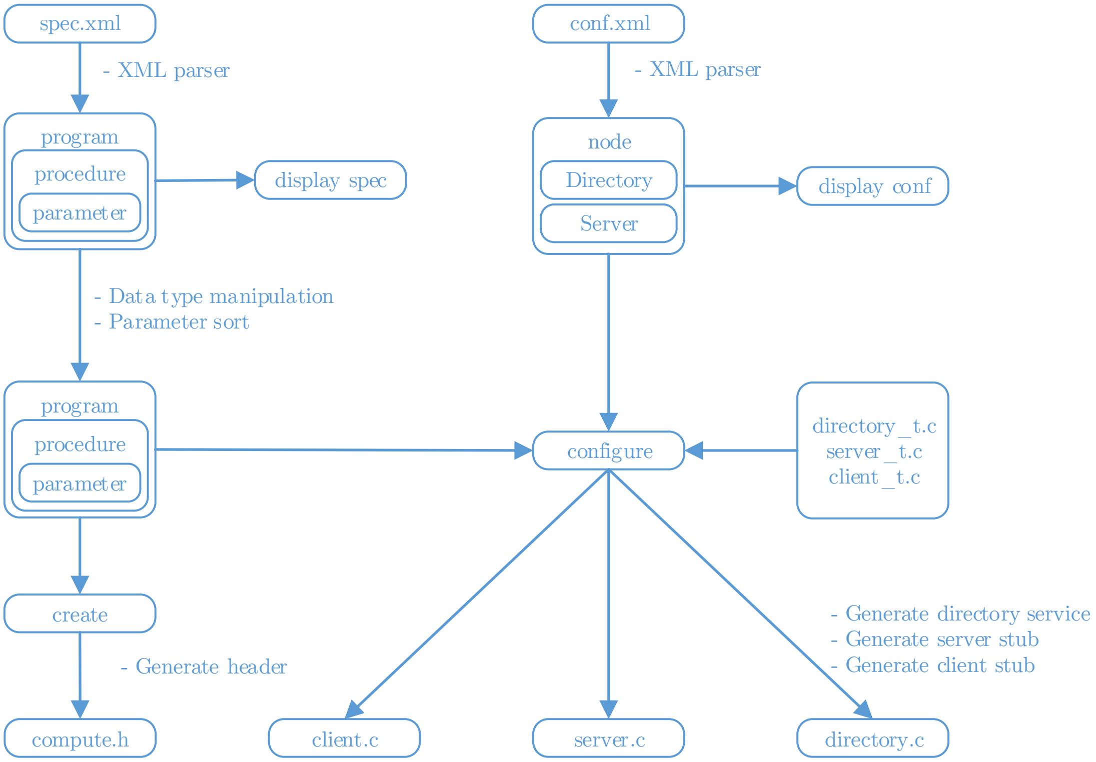

Figure 17 Stub generation flowchart.

3.3.1 Generating Header File

The *create*() function is responsible for generating the header file with respect to the information provided by input spec.xml file. This function used the corresponding information of input parameters which are stored at the program data structure for generating the appropriate header files. It uses the extension parts of each parameter as prefix or postfix in a regular expression fashion for generating procedure definitions and stored them in the compute.h.

3.3.2 Generating Directory Service

As shown in Figure 17 *configure*() is responsible for generating directory service and stubs. In the first step, the *configure* function receives directory template file along with node data structure to generate directory output file. This function applies proper changes to the directory output file with respect to its endpoint address and generates the directory.c file.

3.3.3 Generating Server Stub and Client Stub

The key mission of stubgen is generating server and client stubs. Therefore, aside from generating directory output file, the *configure*() generates the server and client stubs (Figure 17). The process of creating the stubs follows a symmetric fashion for each procedure. It means that while creating client stub the corresponding parts of server stub is also going to be generated. This process is depicted in Figure 18.

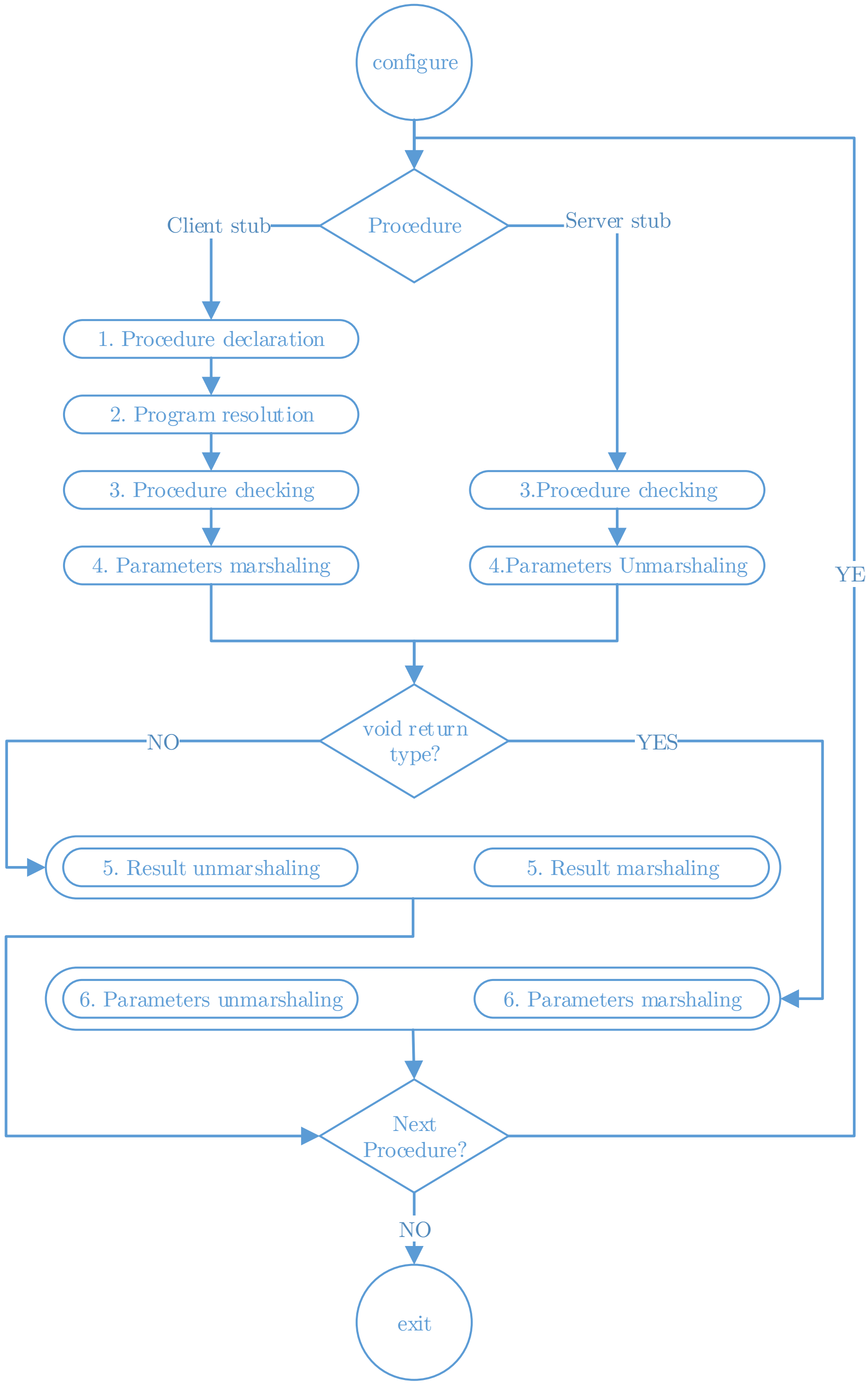

Figure 18 Server and client stub generation.

The whole process of Figure 18 contains the following 6 primary steps.

1. *Procedure declaration*: In this step, the proper procedure declaration is created for client stub. The declaration follows the following totalistic rule which uses the corresponding information of Figure 15 for procedures and parameters:

Procedure$_{\text{returntype}}$ Procedure$_{\text{name}}$ (Parameter $_{\text{type }1}$, Parameter $_{\text{name }1}$, ..., Parameter$_{\text{type }n}$, Parameter$_{\text{name }n}$)

2. *Program resolution*: In this step, the service resolution part of client stub along with directory service endpoint address is inserted to the client stub.
3. *Procedure checking*: In this step, the ID of corresponding procedure is inserted to server stub and the corresponding procedure of client stub for further checking step.
4. *Parameters marshaling/unmarshaling*: In this step, the snippet code for parameters packing in client stub and parameter unpacking in server stub is generated. This step is strictly followed the common marshaling interface of Figure 11 for handling different kinds of parameters.
5. *Result marshaling*: Depending on the return type of procedure, the client and server stubs output step may change. If procedure has an explicit output type, it will be generated in this step. If not, it's going to the next step.
6. *Parameters marshaling/unmarshaling (as outputs)*: A procedure may declare as void, in this case there is no specific result to return for client. Hence, we assume the result may be the associated values of input pointer parameters which are changed during server procedure call. Thus, after executing the corresponding procedure, all the input pointer parameters will be marshaled from server to client. Therefore, the client have access to pointers' new values so that it makes them available in user address space.

4 Compilation Details

There are still some implementation and compilation details for SRPC that are going to be presented in this section. For the sake of simplicity a makefile is written for directory service configuration and stubs generation. Figure 19 shows how this makefile compiles different .c files against each other and generates the directory, server and client binaries. We currently introduce the directory.c, server.c, client.c and compute.h file, but Figure 19 contains two new files compute.c and user.c. The compute.c contains the server application for computing client requests. Moreover, the user.c is the client application which connects to the clients stub for computation. Therefore, in case of adding new a procedure, server side application should be added to the compute.c and client side application should be added to user.c. The primary steps of Figure 19 are as follows:

1. Compiling the stubgen and executing it.
2. Creating the user object file which is the client application.
3. Creating the compute object file which is the server application
4. Creating the client object file which is the server stub.
5. Creating the server object file which is the client stub.
6. Linking the client stub object file against compute and user object files to create client binary file.
7. Linking the client stub object file against compute object file to create server binary file.
8. Creating directory binary file which is the directory service.

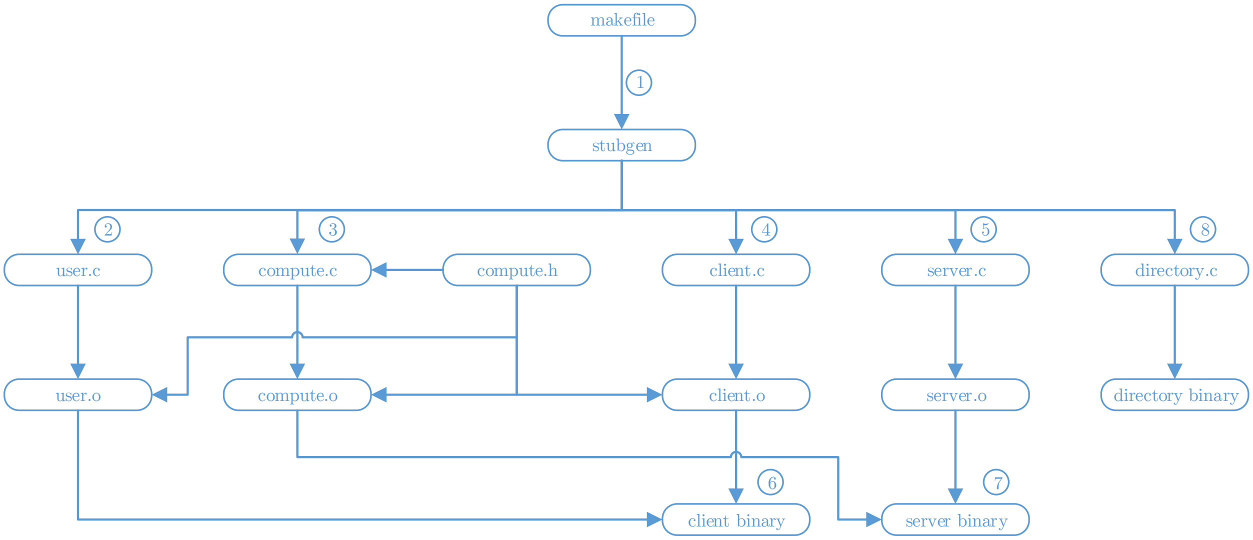

Figure 19 Compilation steps for generating directory and client and server stubs binaries.

6 Conclusion

This report discusses the systematic and logical views of SRPC in terms of design and implementation details. The SRPC contains 3 logical components: directory service, server stub and client stub. The directory service handles server registration requests from server stub and server resolution requests from client stub. The client stub gets the LPC call from user and converts it to RPC for further execution in the server application. Moreover, the server stub receives the RPC call from the client stub and provides the client stub with the corresponding results that are produced by server application.

All of the message passing interfaces of SRPC follows a common message passing convention which consists of a two steps process for sending the request and receiving the ack/nack response. This Simple 2 step process is utilized in register/resolve/check functions for synchronous server registration, server resolution and client checking. Furthermore, SRPC complies a common marshaling interface for transporting its components' configuration and parameter payloads. This interface introduces a common agreement for buffer type and size based on data types. It also tries to limit the number of send/receive attempts with respect to buffer type.

The stubgen (SRPC compiler) utilizes the concept of templates and cloning to reduce the code complexities. Employing some comprehensive regular expressions, it supports unlimited parameter combinations for procedure declaration without putting any constraints on the order of appearance of parameters. Moreover, stubgen has a pre compiled binding step for pointer type to guarantee an optimized send/receive buffer payloads while having an asynchronous parameter marshaling between client and server.

SRPC supports some basic features including multiple service registration for different server instances and service withdrawal for servers. It also provides a simple command line for printing the services list.

References

[1] ezXML library, "http://ezxml.sourceforge.net/"