Remote Procedure Call

System Manual

CS454/654

Developed by: Shisong Tang, Haochen Ding

Date: July, 10, 2013

Important Notice:

This document is meant to accompany the Remote Procedure Call (Assignment 3) of CS454/654 written in C++ by students Haochen Ding and Shisong Tang. The document will detail the design choices chosen, such as the methods used in marshalling/unmarshalling of data, the internal structure of the binder database, handling of function overloading, round-robin scheduling, and the termination procedure among some other chosen design elements. Along with this, error codes will listed along with their respective meaning.

Ackaknowledgements:

The following approved sources were used as reference:

Static Libraries: http://www.linux.com/learn/docs/ldp/695-Program-Library-HOWTO

POSIX Threads: https://computing.llnl.gov/tutorials/pthreads/

Table of Contents

Design Choices

Message (Data) Format

The message protocols are of the form:

Length, Type, Message

Length is an integer detailing the length in bytes of the Message portion, and type is an integer representing the type of the message en route, and Message is the data being transferred.

Message.cpp/Message.h

(These two files are used mainly to marshal and unmarshal the data with a Message structure, not to be confused with the Message portion of the communication protocol)
This contains routines that operate on the Message structure, called **parseMessage** and **createMessage**. The Message structure will be generated after calling parseMessage which will contain the unmarshalled information of the received message buffer. This same structure can be used to marshall a message buffer to send by calling createMessage with some provided Message structure. This is done by calling memcpy and strcpy on the provided information within the Message structure, and packed into a buffer of characters (consecutive bytes). The message structure is as follows:

```
struct Message {
   char*        server_identifier;
   int*         argTypes;
   int          argTypesSize;
   char*        port;
   char*        name;
   int          reasonCode;
   void**       args;
   int          argsLength;
}
```

Marshalling Data

All data traveling outward whether from server, binder, or client will be in the form detailed above. The data will be marshalled into a sequence of bytes representing the specific message it contains.

Marshalling will be accomplished by dynamically allocating memory for a buffer by calling the **new** procedure with the set size, and information will be copied into this location in memory by using **memcpy**.

The following representation is used to describe the message type:

```
#define REGISTER            0
#define REGISTER_SUCCESS    1
#define REGISTER_FAILURE    2
#define LOC_REQUEST         3
#define LOC_SUCCESS         4
#define LOC_FAILURE     5
#define EXECUTE             6
#define EXECUTE_SUCCESS     7
#define EXECUTE_FAILURE     8
#define TERMINATE       9
```

Where REGISTER is the type of communication message sent from server to binder, and the response is type REGISTER_SUCCESS or REGISTER_FAILURE. Type LOC_REQUEST is sent from the client to the binder to request the location of the function requested, and the response is type LOC_SUCCESS or LOC_FAILURE. Type EXECUTE is sent to the server from the client requesting remote execution of the procedure, and the response is type EXECUTE_SUCCESS or EXECUTE_FAILURE. Type TERMINATE is the message sent from the client to the binder, and in response sent to all subsequent servers.

and the size of the specific fields in the marshalled message are:

```
#define LENGTH_SIZE         4
#define TYPE_SIZE           4
#define SERVER_ID_SIZE      128
#define NAME_SIZE           100
#define PORT_SIZE           6
#define ARGTYPE_SIZE    4
```

In the above all numbers are bytes. Where LENGTH_SIZE represents the size of the integer used to store the length of Message. TYPE_SIZE is the size of integer used in storing the type of the Message. SERVER_ID_SIZE is the length used for the host name of the server. NAME_SIZE is the length of the

function name. These sizes are hard coded in order to unmarshall and parse the message received. PORT_SIZE is length of the port, 6 is chosen because the port will be stored as 5 characters each a number making up the actual port and the 6th a null character (because port is represented by an unsigned short int, so max is some five digit number). ARGTYPE_SIZE is the size of integer for representing argument type.

Server Side

Register Message

The server marshals the register message by calculating the length of the message, including its own server identifier (128 bytes), the port it is using (6 bytes), the name of the function it wants to register (100 bytes), argTypes (4 bytes each for each argument). This length is first copied into the outgoing buffer (stored as a sequence of characters), then the type which is register. Then each subsequent field detailed above is copied into the buffer (using strcpy for strings, and memcpy otherwise), consecutively. Then the marshalling of the message buffer is complete and is sent out to the binder.

Execute Response Message

The server will receive the Execute Message from the client and will use parseMessage to parse the received message into a Message structure. Then createMessage will be called with its respective fields on the Message to create a new buffer for either execute success or execute failure.

Client Side

Location Request Message

The client marshals the location request message to the binder by calculating length of the message, by adding name of the function it is requesting (100 bytes), and each argType (4 bytes each) to a outgoing buffer. The length is first copied into the buffer, followed by the type, and then the above fields in order.

Execute Message

The client marshals the execute message to the server by calculating the length of the message by adding the name of the function (100 bytes), each argTypes (4 bytes in each), and the contents within the args void** pointer (which will be either scalar of size defined in the argType, or an array of length and size defined in the argType). This will be formulated from examining the specific bits and bytes of argTypes. These are copied into the buffer along with the type, and sent out.

rpc Cached Call

Write something here ----------

Binder

Response Messages (Register and Location)

The reply messages for both register and location requests will be packed into a buffer by createMessage which will consume the Message structure from the parsed received buffer. The binder will return to the client either a location success message containing the hostname and port of the server containing the requested function, or a location failure with the appropriate error code (see Error Codes). For register replies the binder will respond with a register success or register fail accompanied by the appropriate error or warning (see Error Codes).

Unmarshalling Data

All unmarshalling will be completed with the parseMessage function. The output Message structure will contain the unmarshalled/parsed data. This will allow further marshalling to be accomplished directly by accessing the Message structure's elements. Other operations on the data received can be worked directly on those elements (See above *Message.cpp/Message.h* for detailed fields within the structure).

Database

Binder Database

The binder's database, where the entries correspond to registered functions of servers, is a list of db_struct structures of the form:

```
struct db_struct {
        char*   name;
        int*    argTypes;
        int     argTypeSize;
        char*   address;
        char*   port;
}
```

and the actual database is comprised of the list:

list<db_struct*> binder_db

Database entries will be added in the form of db_struct, which will include the name of the function being registered, the argument types of the function and its size, the hostname of the server containing the function, and the port the server is listening on. Also it is to be noted that the binder contains a vector of server info structure containing the hostname, port number and socket number of all servers that have successfully registered with the binder.

Binder's Handling of Database Entries

Functions will be registered from many servers, and the binder must handle overloading of these functions. This means the binder will only not add entries to the database when the hostname, port number, name of the function, and all of the argtypes (the type of arguments and returns, whether input or output, and whether scalar or array) are the same as some entry already in the database. This means this function has already been registered, and with the new register a warning will be sent to the server (on the server side the server database will update with the newest registered skeleton). Port number is also checked due to that if the port number was different it means that even if hostnames are the same, they are two different servers. This means all other functions registered will be placed unto the database since even if names are the same, the functions can be overloaded.

Round Robin Scheduling

In order to implement fairness, the scheduling works as so: after functions from one server is executed following a location request to the binder, another server will be used for other calls to the same set of functions. This means the binder will use round robin scheduling on the access of server functions.

This is implemented in the following way: Then binder will traverse the database entries from beginning to end in order after it receives a location request. If the server having the requested function is found the database will be reordered; this means every entry belonging to the server with the requested function will be moved to the back of the binder database. The subsequent requests for the same function will search the moved server entries last, hence satisfying the round robin scheduling.

Local Database of Servers

The servers themselves, after successive rpcRegister calls, it must store entries into a local database of the form:

list<server_func*> server_db

where each entry is of the form:

```
struct server_func {
  char*         name;
  int*                  argTypes;
  skeleton              func;
}
```

When a server registers the exact same function in succession, the function skeleton will be updated each time. When an execute message is received on the server side, it will look up the function and call it with the correct argTypes and skeleton.

Shutdown

Unexpected Server Closure

Consider the case that a server finishes registering all functions located on that server and enters the execute phase. A client requesting some function that this server contains will be able to get the hostname and port from the binder and connect to the server. It could be possible that the server unexpectedly shuts down before receiving the client request and after it has registered with the binder. In this case the binder will recognize that the server has closed and will remove all functions from its database belonging to the closed server. Then subsequent client requests to the binder will not find the functions of the closed server.

Termination

The termination message is sent from the client to the binder which is then sent to all servers that are connected with the binder. Each server, upon receiving the terminate message, will wait for all executing threads to complete and then close. The binder refers to its stored vector of server infos to verify that each server has closed (by checking that recv gets 0 from servers). Every time a server closes, the binder complies with the above and removes that servers registered functions from its database. When the binder finalizes that all servers have shut down, it will then shut down itself.

Error Codes

The following error codes should be referred to in case of error:

EHOST	-1
ESOCK	-2
EBIND	-3
ELISTEN	-4
ESNAME	-5
EGADDR	-6
ESEND	-7

ERECV	-8
ESELECT	-9
ETHREAD	-10
ENOFUNC	-11

EHOST:	When the host could not be identified or reached
ESOCK:	Socket could not be used (bind, listen, send, recv, etc)
EBIND:	Failed on binding to the specified socket
ELISTEN:	Failed on trying to listen on the specified socket
ESNAME:	Failed on getting the address that the socket is bound to
EGADDR:	Could not call getaddrinfo correctly
ESEND:	Sending failed to the specified socket
ERECV:	Receiving failed from the specified socket
ESELECT:	Failed to select between specified sockets
ETHREAD:	Pthread creation failed in rpcExecute
ENOFUNC:	Function does not exist in binder database, returned with location request failed