//

Ass 3 notes

//

*****TA's never make mistakes with respects to parameters!!
Only some things like not registering before invoking server execute, etc

--------SUMMARY----------------

1) First CLIENT requests from BINDER the server_identifier (IP address or hostname) and port number of server capable of handling request. CLIENT marshals parameters into REQUEST MESSAGE and sends to SERVER and gets the result

2) SERVER makes a socket to listen for CLIENT requests. When requested, the correct procedure call is found, and STUB extracts parameters from CLIENT message and returns results to client. SERVER registers all server procedures with the BINDER and keeps a TCP connection to the binder open at all times!

3) BINDER takes registration requests from SERVER, and maintains a database of SERVERS and their respective SERVER PROCEDURES. Binder also takes location requests from client processes, returning either server_identifier (IP address or hostname) or port for suitable server, or saying the server does not exist. TERMINATE messages are sent to BINDER first then the server.

--------CLIENT SIDE----------------

CLIENT call for RPC is the **rpcCall** function

int rpcCall(char *name, int *argTypes, void ** args);

the return value is the result of the rpcCall and not the actual procedure
success: 0,
warning: > 0,
error: < 0

name: name of remote procedure, the name **MUST** be registered with binder

argTypes: array for type of argument for each argument in **args**

it is a **4** byte integer,
1st bit: arg_input for 1, 0 for no
2nd bit: arg_output for 1, 0 for no
next 6 bits are 0’s
2nd byte: the type of argument
3rd and 4th byte: length of array if array, if not then 0 for scalar

The last element of **argTypes** must be 0 so size is sizeof **args** + 1

```
#define ARG_CHAR 1
#define ARG_SHORT 2
#define ARG_INT 3
#define ARG_LONG 4
#define ARG_DOUBLE 5
#define ARG_FLOAT 6
#define ARG_INPUT 31
#define ARG_OUTPUT 30
```

```
// result = sum(vector);
#define PARAMETER_COUNT 2 // Number of RPC arguments
#define LENGTH 23 // Vector length
int argTypes[PARAMETER_COUNT+1];
void **args = (void **)malloc(PARAMETER_COUNT * sizeof(void *));
argTypes[0] = (1 << ARG_OUTPUT) | (ARG_INT << 16); // result
argTypes[1] = (1 << ARG_INPUT) | (ARG_INT << 16) | LENGTH; // vector
argTypes[2] = 0; // Terminator
args[0] = (void *)&result;
args[1] = (void *)vector;
rpcCall("sum", argTypes, args);
```

example code ^^^^^^^^^^^^^^^^^^^^^

*output args can be positioned anywhere in **args** array

rpcCall will first send **LOCATION REQUEST MESSAGE** to binder to locate the server for procedure. If cannot locate (failure), return negative integer. Otherwise return 0. If location is successful then, send **EXECUTE REQUEST MESSAGE** to the server.

--------SERVER SIDE------------

1) main server program
2) server functions

for each server function there is a skeleton that does **marshalling and unmarshalling** for actual server function.

3) function skeletons

SERVER TIMELINE:

i) Server calls **rpcInit**:

1) creates a connection socket for accepting connections from CLIENTS

2) opens a connection to binder, same connection is used by server for sending register requests to the binder (this connection is up as long as server is up, so binder knows server is

up)

int rpcInit(void);

RETURN: success is 0, negative if any part of initialization unsuccessful (different for different errors)

ii)Server makes calls to **rpcRegister**:

this registers each server procedure

int rpcRegister(char *name, int *argTypes, skeleton f);

This does 2 thing:

1) call binder saying a server procedure with name and list of argument types is available at this server.

RETURN: 0 for successful registration,
>0 for warning
<0 failure

2) makes an entry in a local **database**, associating each **server skeleton** with **name** and **argTypes**.

skeleton f is the address of the server skeleton which is the server procedure being registered

and

typedef int (*skeleton)(int *, void **);

this returns integer for whether the server function call executes correctly or not. If it bugs return negative, else return zero. If fail then RPC library at server side should return RPC failure message to client

iii)Server calls **rpcExecute(void)**

this gives the skeleton the control, which

1) unmarshalls the message
2) call the right procedure
3) marshal the return from procedure

then the function sends result back to client.

RETURNS:

0 for requested termination (binder has request termination of server,
negative else

This function should handle **<u>multiple</u>** requests from clients without blocking

*SINGLE machine can run multiple servers (same IP), use dynamic PORT!!!

--------BINDER---------------

Accepts REGISTRATION REQUESTS and LOCATION REQUESTS, and sends back replies.

Maintains a database of procedures that have been registered with it including arguments

procedure signature, location

*SERVER cannot know exact length of array for input or output when registering it with binder!!! SO just don’t include length and just find the right function.
if two functions have same name but diff array length, they considered the same. But if one scalar and other is array, they are different.
*HANDLE OVERLOADING!!! for function name with different arguments
*if same SERVER registers the same function name, replace with newest skeleton!
Use ROUND-ROBIN to send requests to SERVERS!!!
BINDER PRINTS FOLLOWING:
BINDER_ADDRESS <machine>
BINDER_PORT <port number>
*SERVER AND CLIENT MUST READ THESE FROM THE ENVIRONMENT VARIABLES

----------SYSTEM TERMINATION--------------

client calls
int rpcTerminate (void);
this request is passed to binder by client stub, binder in turn will inform the servers to terminate. Binder will terminate when all servers have terminated.

*CHECK termination request comes from binder’s IP

---------PROTOCOL---------
Messages that need to be sent: **sever/binder, client/binder, client/server**
in the form: **Length**, **Type**, **Message**

Length: integer for length of message
Type: integer indicating type
Message: the actual message
* Type is in "all caps" use variable size memory to allocate space for
function names

The following messages will have length excluded for simplicity
(SERVER/BINDER MESSAGE)
REGISTER, server_identifier, port, name, argTypes
{type} { Message }
assume fixed length for IP address or hostname, port and name
RETURN: **REGISTER_SUCCESS or REGISTER_FAILURE** and **integer** for warning or

error following

(CLIENT/BINDER MESSAGE)

from **CLIENTS** to **BINDER** used to locate appropriate server procedure:

LOC_REQUEST, name, argTypes

{type} { message }

name and argTypes are parameters from rpcCall

assume fixed length for name

RETURN:

On success: **LOC_SUCCESS, server_identifier, port**

assume fixed length for both

On request failed: **LOC_FAILURE, reasonCode**

where reasonCode is integer for failure condition

(CLIENT/SERVER MESSAGE)

from CLIENT to SERVERS to request execution of server procedure

EXECUTE, name, argTypes, args

RETURN:

On success: **EXECUTE_SUCCESS, name, argTypes, args**

on fail: **EXECUTE_FAILURE, reasonCode**

reasonCode is integer representing reason for failure

--------TERMINATE MESSAGES-------------

For terminating the binder and the servers, client sends the following

TERMINATE

to binder. Binder sends to all servers, servers verify msg is from binder.
Then terminate

-----------BONUS FUNCTIONALITY------------

implement cache in client side library. Cache server locations it received from the binder!!! Then client will not send location request for every rpc request.

Refer to 6.1.1 and 6.1.2

Store functions in library called **librpc.a**

other functions will be specified in **README and how to compile and run. Also documenting dependencies and other things. Include names and userids of both members.**

DO NOT modify rpc.h, creater another header for additional declarations

Makefile for generating RPC library and binder executable “binder”

g++ -L. client.o -lrpc -o client or **another command specified in README** must work for compiling.

1. The interface of rpcInit, rpcCall, rpcRegister, rpcExecute, rpcTerminate cannot change. We will be using these interfaces to run our server and clients. You may however develop your own protocol for internal communication e.g. the location request to the binder.
2. Do not modify rpc.h { instead create a different header file
3. Do not bind to static ports or assume that the server/binder is running on the local machine
4. Do not assume that the code will compile on linux.student.cs without actually compiling it on that environment