```
#include <pthread.h>
#include <netdb.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdint.h>
#include <string>
#include <map>
#include <utility>
#include <vector>
#include <iostream>

#include "rpc.h"

using namespace std;

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

/**
 * Error codes
 */
const int ERROR_INVALID_BINDER_INFO = -10;
const int ERROR_INVALID_ADDR_INFO = -11;
const int ERROR_FAILED_TO_BIND = -12;
const int ERROR_FAILED_TO_LISTEN = -13;
const int ERROR_FAILED_PORT_RETRIEVAL = -14;
const int ERROR_REGISTER = -15;
const int ERROR_FAILED_TO_SELECT = -16;
const int ERROR_FAILED_TO_ACCEPT = -17;
const int ERROR_BINDER_CLOSED = -18;
const int ERROR_NO_BINDER_AVALIABLE = -19;
const int ERROR_CLIENT_FAILED_CONNECT_SERVER = -20;
const int ERROR_BAD_ARG_TYPE = -21;
const int ERROR_LOC_FAILURE = -22;

/**
 *  some const for communication
 */
const char REGISTER_MSG = 'r';
const char EXECUTE_MSG = 'e';
const char EXECUTE_SUCCESS = 'a';
const char EXECUTE_FAILURE = 'b';
const char TERMINATE_MSG = 't';
const char LOC_REQUEST = 'c';
const int LOC_SUCCESS = 1;
const int LOC_FAILURE = 0;


/**
 *  global variables for server side
 */
const int MAX_CLIENTS = 5;
pthread_t pthreadArray [MAX_CLIENTS];
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

char server_hostname[128];
unsigned server_port;

fd_set master;    // master file descriptor list
fd_set read_fds;  // temp file descriptor list for select()
int fdmax;        // maximum file descriptor number
```

```cpp
struct addrinfo hints, *binder_info, *p;
char *binder_addr;
char *binder_port;
unsigned int binder_fd;
struct sockaddr_storage connector_addr_info; // connector can be binder or client
socklen_t addr_len;
int server_listen_fd; // The server listens on this file descriptor
int client_side_fd; // The file descriptor on a client's end


/**
 *  struct argTypeStruct
 */
struct argTypeStruct{
    int arg_in_int_format;
    bool input;
    bool output;
    int type;
    int array_length;

    argTypeStruct(int i) {
        // the original int
        arg_in_int_format = i;

        // type
        type = (((7 << 16) & i) >> 16);

        // if array length is 0, set to 1
        array_length = (((1 << 16) - 1) & i);
        if (array_length ==0) array_length = 1;

        // set input and output
        if (i & 0x80000000) {
            input = true;
        } else {
            input = false;
        }

        if (i & 0x40000000) {
            output = true;
        } else {
            output = false;
        }
    }

    bool operator == (const argTypeStruct & a) const {
        return (input == a.input) && (output == a.output) && (type == a.type);
    }
};

/**
 *  struct function signature
 */
struct signature {
    string function_name;
    vector <argTypeStruct> arg_types;

    signature(string name, vector <argTypeStruct> v) {
        function_name = name;
        arg_types = v;
    }

    bool operator == (const signature & a) const {
        string function_name_a = a.function_name;
        vector <argTypeStruct> arg_types_a = a.arg_types;

        if ((function_name_a != function_name) ||
            (arg_types_a.size() != arg_types.size())) {
            return false;
        } else {
            for (int i=0; i< arg_types_a.size(); i++){
                argTypeStruct arg_a = arg_types_a[i];
                argTypeStruct arg_b = arg_types[i];
                if ((arg_a.input != arg_b.input) ||
                    (arg_a.output != arg_b.output) ||
                    (arg_a.type != arg_b.type)) {
                    return false;
                }
            }
        }
        return true;
    }

    bool operator < (const signature & a) const {
        string function_name_a = a.function_name;
        return (function_name < function_name_a);
    }
};

//map to store the function signature and skeleton, use custom operators
map< signature, skeleton> function_map;

// helper for clients to connect to binder
int clientConnectBinder() {
    binder_addr = getenv("BINDER_ADDRESS");
    binder_port = getenv("BINDER_PORT");

    if ((binder_addr == NULL) || (binder_port == NULL)) return ERROR_INVALID_BINDER_INFO;

    int mysocket = socket(AF_INET, SOCK_STREAM, 0);
    if (mysocket < 0) return ERROR_FAILED_TO_BIND;

    struct sockaddr_in hostaddr;
    hostaddr.sin_family = AF_INET;
    hostaddr.sin_port = htons(atoi(binder_port));

    struct hostent *hostname;
    if ((hostname = gethostbyname(binder_addr)) == NULL) return ERROR_FAILED_TO_BIND;

    bcopy((char *) hostname->h_addr, (char *) &hostaddr.sin_addr.s_addr, hostname->h_length);

    if (connect(mysocket, (struct sockaddr *) &hostaddr, sizeof(hostaddr)) < 0) {
        return ERROR_FAILED_TO_BIND;
    }
    return mysocket;
}


int rpcInit() {

    int listener;     // listening socket descriptor
    int newfd;        // newly accept()ed socket descriptor
    struct sockaddr_storage remoteaddr; // client address
    socklen_t addrlen;

    int yes=1;        // for setsockopt() SO_REUSEADDR, below
    int rv;

    struct addrinfo hints, *binderinfo, *ai, *p;

    /* 1. Get host name */
    gethostname(server_hostname, sizeof(server_hostname));

    /* 2. Then try to connect binder */
    // get addr and port from environment variables
    binder_addr = getenv("BINDER_ADDRESS");
    binder_port = getenv("BINDER_PORT");
    memset(&hints, 0, sizeof (hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    // then get binder addr info
    if ((rv = getaddrinfo(binder_addr, binder_port, &hints, &binderinfo)) != 0) {
        return ERROR_INVALID_BINDER_INFO;
    }
    // then loop to get one binder
    for(p = binderinfo; p != NULL; p = p->ai_next) {
        if ((binder_fd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) < 0) {
            continue;
        }

        if (connect(binder_fd, p->ai_addr, p->ai_addrlen) < 0) {
            close(binder_fd);
            continue;
        }

        break;
    }
    // this mean we failed to bind
    if (p == NULL) {
        cout<<"ERROR_FAILED_TO_BIND, you should set up environment variables first."<<endl;
        return ERROR_FAILED_TO_BIND;
    }
    // free the binderinfo
    freeaddrinfo(binderinfo);  // all done with this


    /* 3. Then create a socket to accept connection from clients */
    // same codes as get binder addr info
    FD_ZERO(&master);
    FD_ZERO(&read_fds);
    memset(&hints, 0, sizeof(hints));

    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;

    if ((rv = getaddrinfo(NULL, "0", &hints, &ai)) != 0) {
        return ERROR_INVALID_ADDR_INFO;
    }

    for(p = ai; p != NULL; p = p->ai_next) {
        if ((server_listen_fd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) < 0) {
            continue;
        }

        if (setsockopt(server_listen_fd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) < 0) {
            continue;
        }

        if (bind(server_listen_fd, p->ai_addr, p->ai_addrlen) < 0) {
            close(server_listen_fd);
            continue;
        }

        break;
    }

    if (p == NULL) {
        return ERROR_FAILED_TO_BIND;
    }

    freeaddrinfo(ai); // all done with this

    // get socket name
    sockaddr* sa;
    socklen_t* sl;
    getsockname(server_listen_fd, sa, sl);

    // listen
    if (listen(server_listen_fd, MAX_CLIENTS) < 0) {
        return ERROR_FAILED_TO_LISTEN;
    }

    // put listener to master set and set fdmax
    FD_SET(server_listen_fd, &master);
    fdmax = server_listen_fd;


    /* 4. Retrieve server_port */
    struct sockaddr_in sin;
    socklen_t len = sizeof(sin);
    if (getsockname(server_listen_fd, (struct sockaddr *)&sin, &len) < 0) {
        return ERROR_FAILED_PORT_RETRIEVAL;
    }

    server_port = ntohs(sin.sin_port);

    return 0;
}


int rpcCall(char* name, int* argTypes, void** args) {
    int* argTypes_copy = argTypes;
    void** args_copy = args;

    vector <argTypeStruct> arg_types;

    // 1. connect to binder and get server location
    binder_fd = clientConnectBinder();
    if (binder_fd < 0) return binder_fd;

    int server_fd;
    char msg;
    do {
        // send LOC_REQUEST to binder
        msg = LOC_REQUEST;
        send(binder_fd, &msg, sizeof(char), 0);
        // then function name
        // NOTE: add a trailing '\0' to the original name just to ensure no bug will occurs
        // Assume that the given name is 127 characters or less.
        int name_length = 128;
        char *name_to_send = new char[name_length];
        for (int c = 0; c < name_length - 1; c++) {
            name_to_send[c] = name[c];
        }
        name_to_send[name_length - 1] = '\0';
        send(binder_fd, name_to_send, name_length, 0);
        delete [] name_to_send;
        // argTypes
        while(true) {
            int temp_int = htonl( *argTypes_copy);
            send(binder_fd, &temp_int, sizeof(temp_int), 0);
            if (temp_int == 0) break;
            argTypes_copy++;
        }//while

        // get server_id from binder
        unsigned int serverPort;
        char hostname[128];

        int msg_int;
        if (recv(binder_fd, &msg_int, sizeof(msg_int), 0) <= 0) {
            return ERROR_NO_BINDER_AVALIABLE;
        }
        if (msg_int == LOC_SUCCESS) {
            if( recv(binder_fd, &hostname, sizeof(char)*128, 0) <= 0) {
                return ERROR_NO_BINDER_AVALIABLE;
            }

            if( recv(binder_fd, &serverPort, sizeof(serverPort), 0) <= 0) {
                return ERROR_NO_BINDER_AVALIABLE;
            }

            serverPort = ntohl(serverPort);

        } else if (msg_int == LOC_FAILURE) { //LOC_FAILURE
            recv(binder_fd, &serverPort, sizeof(serverPort), 0);
            serverPort = ntohl(serverPort);
            return ERROR_LOC_FAILURE;
        }//end if (msg == LOC_SUCCESS)

        // try to connect server
        if (hostname == NULL) return ERROR_CLIENT_FAILED_CONNECT_SERVER;

        server_fd = socket(AF_INET, SOCK_STREAM, 0);

        struct sockaddr_in hostaddr;
        hostaddr.sin_family = AF_INET;
        hostaddr.sin_port = htons(serverPort);

        struct hostent *host_ent;
        if ((host_ent = gethostbyname(hostname)) == NULL) return ERROR_CLIENT_FAILED_CONNECT_SERVER;

        bcopy((char *) host_ent->h_addr, (char *) &hostaddr.sin_addr.s_addr, host_ent->h_length);

        if (connect(server_fd, (struct sockaddr *) &hostaddr, sizeof(hostaddr)) < 0) {
            return ERROR_CLIENT_FAILED_CONNECT_SERVER;
        }
    } while (server_fd < 0);

    close(binder_fd);

    // 2. send execute msg to server
    // send EXECUTE_MSG
    msg = EXECUTE_MSG;
    send(server_fd, &msg, sizeof(msg), 0);
    // then function name
    // NOTE: add a trailing ’\0’ to the original name just to ensure no bug will occurs
    // Assume that the given name is 127 characters or less.
    int name_length = 128;
    char *name_to_send = new char[name_length];
    for (int c = 0; c < name_length - 1; c++) {
        name_to_send[c] = name[c];
    }
    name_to_send[name_length - 1] = ’\0’;
    send(server_fd, name_to_send, name_length, 0);

    delete [] name_to_send;
    // send argTypes, save into vector arg_types
    argTypes_copy = argTypes;
    while(true) {
        int temp_int = *argTypes_copy;
        int send_int = htonl(temp_int);
        send(server_fd, &send_int, sizeof(send_int), 0);

        if (temp_int == 0) break;

        argTypeStruct *temp_argTypeStruct = new argTypeStruct(temp_int);
        arg_types.push_back(*temp_argTypeStruct);
        argTypes_copy++;
    }
    /* send args */
    // first compute the send buffer length
    unsigned int length_input = 0;
    unsigned int length_output = 0;
```

```cpp

    for (vector<argTypeStruct>::iterator it = arg_types.begin(); it != arg_types.end(); it++) {
        argTypeStruct arg = *it;
        if (arg.input) {
            switch (arg.type) {
                case ARG_CHAR:
                    length_input += arg.array_length * sizeof(char);
                    break;
                case ARG_SHORT:
                    length_input += arg.array_length * sizeof(short);
                    break;
                case ARG_INT:
                    length_input += arg.array_length * sizeof(int);
                    break;
                case ARG_LONG:
                    length_input += arg.array_length * sizeof(long);
                    break;
                case ARG_DOUBLE:
                    length_input += arg.array_length * sizeof(double);
                    break;
                case ARG_FLOAT:
                    length_input += arg.array_length * sizeof(float);
                    break;
                default: {
                    std::cout<< "ERROR in a thread of rpcExecute: arg type not known." << std::endl;
                    return ERROR_BAD_ARG_TYPE;
                }
            }
        }//end if (arg.input)

        if (arg.output) {
            switch (arg.type) {
                case ARG_CHAR:
                    length_output += arg.array_length * sizeof(char);
                    break;
                case ARG_SHORT:
                    length_output += arg.array_length * sizeof(short);
                    break;
                case ARG_INT:
                    length_output += arg.array_length * sizeof(int);
                    break;
                case ARG_LONG:
                    length_output += arg.array_length * sizeof(long);
                    break;
                case ARG_DOUBLE:
                    length_output += arg.array_length * sizeof(double);
                    break;
                case ARG_FLOAT:
                    length_output += arg.array_length * sizeof(float);
                    break;
                default: {
                    std::cout<< "ERROR in a thread of rpcExecute: arg type not known." << std::endl;
                    return ERROR_BAD_ARG_TYPE;
                }
            }
        }//end if (arg.output)

    }//end for loop

    // malloc the buffer
    unsigned char * input_args = (unsigned char *) malloc(length_input * sizeof(char));
    unsigned char * output_args = (unsigned char *) malloc(length_output * sizeof(char));
    unsigned char * buffer_cursor = input_args;

    // copy from args to send buffer(which is input_args)
    for (vector<argTypeStruct>::iterator it = arg_types.begin(); it != arg_types.end(); it++) {
        argTypeStruct arg = *it;
        if (arg.input) {
            switch (arg.type) {
                case (ARG_CHAR): {
                    char* temp_buffer = (char* )(*args_copy);
                    memcpy(buffer_cursor, temp_buffer, arg.array_length * sizeof(char));
                    buffer_cursor += arg.array_length * sizeof(char);
                    break;
                }
                case (ARG_SHORT): {
                    short* temp_buffer = (short* )(*args_copy);
                    memcpy(buffer_cursor, temp_buffer, arg.array_length * sizeof(short));
                    buffer_cursor += arg.array_length * sizeof(short);
                    break;
                }
                case (ARG_INT): {
                    int* temp_buffer = (int* )(*args_copy);
                    memcpy(buffer_cursor, temp_buffer, arg.array_length * sizeof(int));
                    buffer_cursor += arg.array_length * sizeof(int);
                    break;
                }
                case (ARG_LONG): {
                    long* temp_buffer = (long* )(*args_copy);
                    memcpy(buffer_cursor, temp_buffer, arg.array_length * sizeof(long));
                    buffer_cursor += arg.array_length * sizeof(long);
                    break;
                }
                case (ARG_DOUBLE): {
                    double* temp_buffer = (double* )(*args_copy);
                    memcpy(buffer_cursor, temp_buffer, arg.array_length * sizeof(double));
                    buffer_cursor += arg.array_length * sizeof(double);
                    break;
                }
                case (ARG_FLOAT): {
                    float* temp_buffer = (float* )(*args_copy);
                    memcpy(buffer_cursor, temp_buffer, arg.array_length * sizeof(float));
                    buffer_cursor += arg.array_length * sizeof(float);
                    break;
                }
                default: {
                    std::cout<< "ERROR in rpcCall: arg type not known." << std::endl;
                    return ERROR_BAD_ARG_TYPE;
                }
            }
        }//end if (arg.input)

        args_copy++;
    }//end for loop

    // send the input_args buffer
    send(server_fd, input_args, length_input, 0);

    // 3. send done. wait for response
    if ( recv(server_fd, &msg, sizeof(msg), 0) <= 0) return ERROR_CLIENT_FAILED_CONNECT_SERVER;
    if ( msg == EXECUTE_FAILURE) {
        int error_code;
        if ( recv(server_fd, &error_code, sizeof(error_code), 0) <= 0) return ERROR_CLIENT_FAILED_CONNECT_SERVER;
        return ntohl(error_code);
    }

    // 4. when EXECUTE_SUCCESS, receive args
    if ( msg == EXECUTE_SUCCESS) {
        if ( recv(server_fd, output_args, length_output, 0) <= 0) return ERROR_CLIENT_FAILED_CONNECT_SERVER;
        buffer_cursor = output_args;

        for (int i = 0; i< arg_types.size(); i++) {
            argTypeStruct arg = arg_types[i];
            if (arg.output) {
                switch (arg.type) {
                    case (ARG_CHAR): {
                        char *temp_buffer = (char *)malloc(arg.array_length * sizeof(char));
                        memcpy( temp_buffer, buffer_cursor, arg.array_length * sizeof(char));
                        buffer_cursor += arg.array_length * sizeof(char);
                        for (int j=0; j< arg.array_length; j++) {
                            ((char *)args[i]) [j] = temp_buffer[j];
                        }
                        free(temp_buffer);
                        break;
                    }
                    case (ARG_SHORT): {
                        short *temp_buffer = (short *)malloc(arg.array_length * sizeof(short));
                        memcpy( temp_buffer, buffer_cursor, arg.array_length * sizeof(short));
                        buffer_cursor += arg.array_length * sizeof(short);
                        for (int j=0; j< arg.array_length; j++) {
                            ((short *)args[i]) [j] = temp_buffer[j];
                        }
                        free(temp_buffer);
                        break;
                    }
                    case (ARG_INT): {
                        int *temp_buffer = (int *)malloc(arg.array_length * sizeof(int));
                        memcpy( temp_buffer, buffer_cursor, arg.array_length * sizeof(int));
                        buffer_cursor += arg.array_length * sizeof(int);
                        for (int j=0; j< arg.array_length; j++) {
                            ((int *)args[i]) [j] = temp_buffer[j];
                        }
                        free(temp_buffer);
                        break;
                    }
                    case (ARG_LONG): {
                        long *temp_buffer = (long *)malloc(arg.array_length * sizeof(long));
                        memcpy( temp_buffer, buffer_cursor, arg.array_length * sizeof(long));
                        buffer_cursor += arg.array_length * sizeof(long);
                        for (int j=0; j< arg.array_length; j++) {
                            ((long *)args[i]) [j] = temp_buffer[j];
                        }
                        free(temp_buffer);
                        break;
                    }
                    case (ARG_DOUBLE): {
                        double *temp_buffer = (double *)malloc(arg.array_length * sizeof(double));
                        memcpy( temp_buffer, buffer_cursor, arg.array_length * sizeof(double));
                        buffer_cursor += arg.array_length * sizeof(double);
                        for (int j=0; j< arg.array_length; j++) {
                            ((double *)args[i]) [j] = temp_buffer[j];
                        }
                        free(temp_buffer);
                        break;
                    }
                    case (ARG_FLOAT): {
                        float *temp_buffer = (float *)malloc(arg.array_length * sizeof(float));
                        memcpy( temp_buffer, buffer_cursor, arg.array_length * sizeof(float));
                        buffer_cursor += arg.array_length * sizeof(float);
                        for (int j=0; j< arg.array_length; j++) {
                            ((float *)args[i]) [j] = temp_buffer[j];
                        }
                        free(temp_buffer);
                        break;
                    }
                    default: {
                        std::cout<< "ERROR in rpcCall: arg type not known." << std::endl;
                        return ERROR_BAD_ARG_TYPE;
                    }
                }
            }//end if (arg.input)
        }//end for loop
    }// end if ( msg == EXECUTE_SUCCESS)

    close(server_fd);
    free(input_args);
    free(output_args);

    return 0;
}


int rpcCacheCall(char* name, int* argTypes, void** args) {return 0;}


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

    /* 1. Send register info to binder
     * format: REGISTER, server_identifier, port, name, argTypes
     */

    // first send a character 'r' to notify the binder a register msg is coming
    char register_msg = REGISTER_MSG;
    send(binder_fd, &register_msg, sizeof(register_msg), 0);

    // then send the server_identifier
    send(binder_fd, server_hostname, 128 * sizeof(char), 0);

    // then the server_port
    int tmp_server_port = htonl((uint32_t)server_port);
    send(binder_fd, &tmp_server_port, sizeof(tmp_server_port), 0);

    // then function name
    // NOTE: add a trailing '\0' to the original name just to ensure no bug will occurs
    // Assume that the given name is 127 characters or less.
    int name_length = 128;
    char *name_to_send = new char[name_length];
    for (int c = 0; c < name_length - 1; c++) {
        name_to_send[c] = name[c];
    }
    name_to_send[name_length - 1] = '\0';
    send(binder_fd, name_to_send, name_length, 0);
    delete [] name_to_send;

    // then the argTypes, send one by one
    int *type = argTypes;
    int tmp;
    while (*type != 0) {
        tmp = htonl((uint32_t)*type);
        send(binder_fd, &tmp, sizeof(tmp), 0);
        type++;
    }
    tmp = htonl((uint32_t)*type); // this is the trailing 0
    send(binder_fd, &tmp, sizeof(tmp), 0);

    /* 2. Recieve register response from binder
     * format: first int {REGISTER_SUCCESS,REGISTER_FAILURE}
     *         second int indicate warnings or errors type ( 0 when success)
     */
    unsigned int binder_response_result;
    unsigned int rcv_status_code;

    // receive response result
    if (recv(binder_fd, &binder_response_result, sizeof(binder_response_result), 0) <= 0) {
        return ERROR_REGISTER;
    }
    // receive status code
    if (recv(binder_fd, &rcv_status_code, sizeof(rcv_status_code), 0) < 0) {
        return ERROR_REGISTER;
    }
    binder_response_result = ntohl(binder_response_result);
    rcv_status_code = ntohl(rcv_status_code);

    if (rcv_status_code < 0) return rcv_status_code; // when it's an error

    /* 3. Register locally
```

```
     *
     */
    string name_string(name);
    vector <argTypeStruct> arg_types;

    for (type = argTypes; *type != 0; type++) {
        argTypeStruct *temp_argTypeStruct = new argTypeStruct( *type);
        arg_types.push_back(*temp_argTypeStruct);
    }

    signature *sig = new signature(name_string, arg_types);

    //if signature not exist, store it
    auto it = function_map.find(*sig);
    if(it == function_map.end()) {
        function_map.insert( pair<signature, skeleton> (*sig,f) );
        //cout<<"Local register done. "<< string(name) <<endl;
    } else {
        if (it->second != f) {
            it->second = f;
            //cout<<"Duplicate function signature. Override the skeleton."<<endl;
        } else {
            //cout<<"Duplicate function signature and skeleton. Do nothing."<<endl;
        }
    }

    return 0;
}


void * executeThread (void * parms) {
    int fd = (intptr_t) parms;

    // 1. get name
    char *name = new char [128];
    if (recv(client_side_fd, name, 128, 0) <= 0) {
        free(name);
        std::cout<< "ERROR in a thread of rpcExecute: get client host name." << std::endl;
        return NULL;
    }
    string name_string(name);
    free(name);

    // 2. get argTypes
    vector <argTypeStruct> arg_types;

    while (1) { //loop until get 0
        int arg_type;
        if (recv(client_side_fd, &arg_type, sizeof(int), 0) <= 0) {
            std::cout<< "ERROR in a thread of rpcExecute: client hung up." << std::endl;
            return NULL;
        }
        arg_type = ntohl(arg_type);

        if (arg_type == 0) {
            break;
        }

        argTypeStruct *temp_argTypeStruct = new argTypeStruct(arg_type);
        arg_types.push_back(*temp_argTypeStruct);
    }

    // 3. bulid signature
    signature *sig = new signature(name_string, arg_types);

    // 4. check if signature exist, if exist get skeleton
    auto it = function_map.find(*sig);
    if (it == function_map.end()) {
        std::cout<< "ERROR in a thread of rpcExecute: function not exist." << std::endl;
        return NULL;
    }
    skeleton f = it->second;

    // 5. malloc memory for input args and output args
    unsigned int length_input = 0;
    unsigned int length_output = 0;

    for (vector<argTypeStruct>::iterator it = arg_types.begin(); it != arg_types.end(); it++) {
        argTypeStruct arg = *it;
        if (arg.input) {
            switch (arg.type) {
                case ARG_CHAR:
                    length_input += arg.array_length * sizeof(char);
                    break;
                case ARG_SHORT:
                    length_input += arg.array_length * sizeof(short);
                    break;
                case ARG_INT:
                    length_input += arg.array_length * sizeof(int);
                    break;
                case ARG_LONG:
                    length_input += arg.array_length * sizeof(long);
                    break;
                case ARG_DOUBLE:
                    length_input += arg.array_length * sizeof(double);
                    break;
                case ARG_FLOAT:
                    length_input += arg.array_length * sizeof(float);
                    break;
                default: {
                    std::cout<< "ERROR in a thread of rpcExecute: arg type not known." << std::endl;
                    return NULL;
                }
            }
        }//end if (arg.input)

        if (arg.output) {
            switch (arg.type) {
                case ARG_CHAR:
                    length_output += arg.array_length * sizeof(char);
                    break;
                case ARG_SHORT:
                    length_output += arg.array_length * sizeof(short);
                    break;
                case ARG_INT:
                    length_output += arg.array_length * sizeof(int);
                    break;
                case ARG_LONG:
                    length_output += arg.array_length * sizeof(long);
                    break;
                case ARG_DOUBLE:
                    length_output += arg.array_length * sizeof(double);
                    break;
                case ARG_FLOAT:
                    length_output += arg.array_length * sizeof(float);
                    break;
                default: {
                    std::cout<< "ERROR in a thread of rpcExecute: arg type not known." << std::endl;
                    return NULL;
                }
            }
        }//end if (arg.output)

    }//end for loop

    void **args = (void **)malloc(arg_types.size() * sizeof(void*));
    unsigned char * input_args = (unsigned char *) malloc(length_input * sizeof(char));
    unsigned char * output_args = (unsigned char *) malloc(length_output * sizeof(char));
    unsigned char * buffer_cursor;

    // 6. get args from clients
    if ( recv(client_side_fd, input_args, length_input, 0) <=0 ) {
        std::cout<< "ERROR in a thread of rpcExecute: client hung up." << std::endl;
        return NULL;
    }

    // 7. copy memory from input_args into args
    buffer_cursor = input_args;
    for (int i = 0; i< arg_types.size(); i++) {
        argTypeStruct arg = arg_types[i];

        switch (arg.type) {
            case (ARG_CHAR): {
                char *temp_buffer = (char *)malloc(arg.array_length * sizeof(char));
                if (arg.input) {
                    memcpy( temp_buffer, buffer_cursor, arg.array_length * sizeof(char));
                    buffer_cursor += arg.array_length * sizeof(char);
                }
                args[i] = temp_buffer;
                break;
            }

            case (ARG_SHORT): {
                short *temp_buffer = (short *)malloc(arg.array_length * sizeof(short));
                if (arg.input) {
                    memcpy( temp_buffer, buffer_cursor, arg.array_length * sizeof(short));
                    buffer_cursor += arg.array_length * sizeof(short);
                }
                args[i] = temp_buffer;
                break;
            }

            case (ARG_INT): {
                int *temp_buffer = (int *)malloc(arg.array_length * sizeof(int));
                if (arg.input) {
                    memcpy( temp_buffer, buffer_cursor, arg.array_length * sizeof(int));
                    buffer_cursor += arg.array_length * sizeof(int);
                }
                args[i] = temp_buffer;
                break;
            }

            case (ARG_LONG): {
                long *temp_buffer = (long *)malloc(arg.array_length * sizeof(long));
                if (arg.input) {
                    memcpy( temp_buffer, buffer_cursor, arg.array_length * sizeof(long));
                    buffer_cursor += arg.array_length * sizeof(long);
                }
                args[i] = temp_buffer;
                break;
            }

            case (ARG_DOUBLE): {
                double *temp_buffer = (double *)malloc(arg.array_length * sizeof(double));
                if (arg.input) {
                    memcpy( temp_buffer, buffer_cursor, arg.array_length * sizeof(double));
                    buffer_cursor += arg.array_length * sizeof(double);
                }
                args[i] = temp_buffer;
                break;
            }

            case (ARG_FLOAT): {
                float *temp_buffer = (float *)malloc(arg.array_length * sizeof(float));
                if (arg.input) {
                    memcpy( temp_buffer, buffer_cursor, arg.array_length * sizeof(float));
                    buffer_cursor += arg.array_length * sizeof(float);
                }
                args[i] = temp_buffer;
                break;
            }

            default: {
                std::cout<< "ERROR in a thread of rpcExecute: arg type not known." << std::endl;
                return NULL;
            }
        }//switch
    }//end for loop

    // 8. execute on server

    int * temp_args = new int [ arg_types.size() + 1];
    for (int i = 0; i < arg_types.size(); i++) {
        temp_args[i] = arg_types[i].arg_in_int_format;
    }
    temp_args[ arg_types.size() ] = 0;

    int result = (*f)(temp_args, args);

    // 9. send back result
    if (result < 0) {
        char temp_char = EXECUTE_FAILURE;
        send(client_side_fd, &temp_char, sizeof(temp_char), 0);
        result = htonl((uint32_t)result);
        send(client_side_fd, &result, sizeof(result), 0);
    } else {
        char temp_char = EXECUTE_SUCCESS;
        send(client_side_fd, &temp_char, sizeof(temp_char), 0);

        buffer_cursor = output_args;

        for (int i = 0; i< arg_types.size(); i++) {
            argTypeStruct arg = arg_types[i];

            switch (arg.type) {
                case (ARG_CHAR): {
                    if (arg.output) {
                        char *temp_buffer = (char *) args[i];
                        memcpy(buffer_cursor, temp_buffer, arg.array_length * sizeof(char));
                        buffer_cursor += arg.array_length * sizeof(char);
                    }
                    break;
                }

                case (ARG_SHORT): {
                    if (arg.output) {
                        short *temp_buffer = (short *) args[i];
                        memcpy(buffer_cursor, temp_buffer, arg.array_length * sizeof(short));
                        buffer_cursor += arg.array_length * sizeof(short);
                    }
                    break;
                }

                case (ARG_INT): {
                    if (arg.output) {
                        int *temp_buffer = (int *) args[i];
                        memcpy(buffer_cursor, temp_buffer, arg.array_length * sizeof(int));
                        buffer_cursor += arg.array_length * sizeof(int);
                    }
                    break;
                }

                case (ARG_LONG): {
                    if (arg.output) {
                        long *temp_buffer = (long *) args[i];
                        memcpy(buffer_cursor, temp_buffer, arg.array_length * sizeof(long));
                        buffer_cursor += arg.array_length * sizeof(long);
                    }
                    break;
                }

                case (ARG_DOUBLE): {
                    if (arg.output) {
                        double *temp_buffer = (double *) args[i];
                        memcpy(buffer_cursor, temp_buffer, arg.array_length * sizeof(double));
                        buffer_cursor += arg.array_length * sizeof(double);
                    }
                    break;
                }

                case (ARG_FLOAT): {
                    if (arg.output) {
                        float *temp_buffer = (float *) args[i];
                        memcpy(buffer_cursor, temp_buffer, arg.array_length * sizeof(float));
                        buffer_cursor += arg.array_length * sizeof(float);
                    }
                    break;
                }

                default: {
                    std::cout<< "ERROR in a thread of rpcExecute: arg type not known." << std::endl;
                    return NULL;
                }
            }//switch
        }//end for loop

        send(client_side_fd,output_args,length_output,0);

    }//end if else

    // 10. free memory
    delete [] temp_args;
    free (input_args);
    free (output_args);

    for (int i = 0; i< arg_types.size(); i++) {
        free( args[i] );
    }
    free(args);

    // 11. remove from master
    pthread_mutex_lock(&mutex);
        close(fd);
        FD_CLR(fd, &master);
    pthread_mutex_unlock(&mutex);
}


int rpcExecute(){

    int listener;     // listening socket descriptor
    int newfd;        // newly accept()ed socket descriptor
    struct sockaddr_storage remoteaddr; // client address
    socklen_t addrlen;

    char buf[256];    // buffer for client data
    int nbytes;

    char remoteIP[INET6_ADDRSTRLEN];

    int yes=1;        // for setsockopt() SO_REUSEADDR, below
    int i, rv;

    struct addrinfo hints, *ai, *p;

    // add the listener to the master set
    FD_SET(binder_fd, &master);

    // main loop
    while(1) {
        read_fds = master; // copy it

        if (select(fdmax+1, &read_fds, NULL, NULL, NULL) == -1) {
            return ERROR_FAILED_TO_SELECT;
        }
        // run through the existing connections looking for data to read
        for(i = 0; i <= fdmax; i++) {
            if (FD_ISSET(i, &read_fds)) { // we got one!!
                if (i == server_listen_fd) {
                    // handle new connections
                    addrlen = sizeof remoteaddr;
                    client_side_fd = accept(server_listen_fd,
                                   (struct sockaddr *)&remoteaddr,
                                   &addrlen);
                    if (client_side_fd == -1) {
                        return ERROR_FAILED_TO_ACCEPT;
                    } else {
                        pthread_mutex_lock(&mutex);
                            FD_SET(client_side_fd, &master); // add to master set
                            if (client_side_fd > fdmax) {    // keep track of the max
                                fdmax = client_side_fd;
                            }
                        pthread_mutex_unlock(&mutex);

                    }
                } else {
                    // handle data from a client
                    char msg_type;
                    if (recv(i, &msg_type, sizeof(msg_type), 0) <= 0) {
                        pthread_mutex_lock(&mutex);
                            close(i); // bye!
                            FD_CLR(i, &master); // remove from master set
                        pthread_mutex_unlock(&mutex);
                        return ERROR_BINDER_CLOSED;
                    } else {
                        switch (msg_type) {
                            case EXECUTE_MSG: {
                                // only client can call execute
                                if (i != binder_fd){
                                    pthread_create(&(pthreadArray[i]), NULL, &executeThread, (void *) (intptr_t) i);
                                    FD_CLR(i, &master);
                                }
                                break;
                            }
                            case TERMINATE_MSG: {
                                // only binder can call terminate
                                if (i == binder_fd){
                                    printf("Terminate message from %d .\n", i);
                                    for (int ii = 0; ii < MAX_CLIENTS; ii++) {
                                        pthread_join( pthreadArray[ii] , NULL );
                                    }
                                    //free(pthreadArray);
                                    return 0; // successfully terminated
                                }
                                break;
                            }
                            default:{
                                printf("Invalid message. Just ignored. \n");
                                break;
                            }
                        }//switch
                    }
                } // END handle data from client
            } // END got new incoming connection
        } // END looping through file descriptors
    } // END for(;;)--and you thought it would never end!
    return 0;
}



int rpcTerminate() {
    int temp_socket= clientConnectBinder();
    if (temp_socket < 0) {
      return temp_socket;
    }

    char temp_char = TERMINATE_MSG;
    send(temp_socket, &temp_char, sizeof(temp_char), 0);

    close(temp_socket);

    return 0;
}
```