```cpp
#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
```

```
69 struct addrinfo hints, *binder_info, *p;
70 char *binder_addr;
71 char *binder_port;
72 unsigned int binder_fd;
73 struct sockaddr_storage connector_addr_info; // connector can be binder or client
74 socklen_t addr_len;
75 int server_listen_fd; // The server listens on this file descriptor
76 int client_side_fd; // The file descriptor on a client's end
77
78
79 /**
80  *  struct argTypeStruct
81  */
82 struct argTypeStruct{
83     int arg_in_int_format;
84     bool input;
85     bool output;
86     int type;
87     int array_length;
88
89     argTypeStruct(int i) {
90         // the original int
91         arg_in_int_format = i;
92
93         // type
94         type = (((7 << 16) & i) >> 16);
95
96         // if array length is 0, set to 1
97         array_length = (((1 << 16) - 1) & i);
98         if (array_length ==0) array_length = 1;
99
100        // set input and output
101        if (i & 0x80000000) {
102            input = true;
103        } else {
104            input = false;
105        }
106
107        if (i & 0x40000000) {
108            output = true;
109        } else {
110            output = false;
111        }
112    }
113
114    bool operator == (const argTypeStruct & a) const {
115        return (input == a.input) && (output == a.output) && (type == a.type);
116    }
117 };
118
119 /**
120  *  struct function signature
121  */
122 struct signature {
123     string function_name;
124     vector <argTypeStruct> arg_types;
125
126     signature(string name, vector <argTypeStruct> v) {
127         function_name = name;
128         arg_types = v;
129     }
130
131     bool operator == (const signature & a) const {
132         string function_name_a = a.function_name;
133         vector <argTypeStruct> arg_types_a = a.arg_types;
134
135         if ((function_name_a != function_name) ||
136             (arg_types_a.size() != arg_types.size())) {
137             return false;
```

```
138         } else {
139             for (int i=0; i< arg_types_a.size(); i++){
140                 argTypeStruct arg_a = arg_types_a[i];
141                 argTypeStruct arg_b = arg_types[i];
142                 if ((arg_a.input != arg_b.input) ||
143                     (arg_a.output != arg_b.output) ||
144                     (arg_a.type != arg_b.type)) {
145                     return false;
146                 }
147             }
148         }
149         return true;
150     }
151 
152     bool operator < (const signature & a) const {
153         string function_name_a = a.function_name;
154         return (function_name < function_name_a);
155     }
156 };
157 
158 //map to store the function signature and skeleton, use custom operators
159 map< signature, skeleton> function_map;
160 
161 // helper for clients to connect to binder
162 int clientConnectBinder() {
163     binder_addr = getenv("BINDER_ADDRESS");
164     binder_port = getenv("BINDER_PORT");
165 
166     if ((binder_addr == NULL) || (binder_port == NULL)) return ERROR_INVALID_BINDER_INFO;
167 
168     int mysocket = socket(AF_INET, SOCK_STREAM, 0);
169     if (mysocket < 0) return ERROR_FAILED_TO_BIND;
170 
171     struct sockaddr_in hostaddr;
172     hostaddr.sin_family = AF_INET;
173     hostaddr.sin_port = htons(atoi(binder_port));
174 
175     struct hostent *hostname;
176     if ((hostname = gethostbyname(binder_addr)) == NULL) return ERROR_FAILED_TO_BIND;
177 
178     bcopy((char *) hostname->h_addr, (char *) &hostaddr.sin_addr.s_addr, hostname->h_length);
179 
180     if (connect(mysocket, (struct sockaddr *) &hostaddr, sizeof(hostaddr)) < 0) {
181         return ERROR_FAILED_TO_BIND;
182     }
183     return mysocket;
184 }
185 
186 
187 int rpcInit() {
188 
189     int listener;     // listening socket descriptor
190     int newfd;        // newly accept()ed socket descriptor
191     struct sockaddr_storage remoteaddr; // client address
192     socklen_t addrlen;
193 
194     int yes=1;        // for setsockopt() SO_REUSEADDR, below
195     int rv;
196 
197     struct addrinfo hints, *binderinfo, *ai, *p;
198 
199     /* 1. Get host name */
200     gethostname(server_hostname, sizeof(server_hostname));
201 
202     /* 2. Then try to connect binder */
203     // get addr and port from environment variables
204     binder_addr = getenv("BINDER_ADDRESS");
205     binder_port = getenv("BINDER_PORT");
206     memset(&hints, 0, sizeof (hints));
```

```
207     hints.ai_family = AF_UNSPEC;
208     hints.ai_socktype = SOCK_STREAM;
209     // then get binder addr info
210     if ((rv = getaddrinfo(binder_addr, binder_port, &hints, &binderinfo)) != 0) {
211         return ERROR_INVALID_BINDER_INFO;
212     }
213     // then loop to get one binder
214     for(p = binderinfo; p != NULL; p = p->ai_next) {
215         if ((binder_fd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) < 0) {
216             continue;
217         }
218 
219         if (connect(binder_fd, p->ai_addr, p->ai_addrlen) < 0) {
220             close(binder_fd);
221             continue;
222         }
223 
224         break;
225     }
226     // this mean we failed to bind
227     if (p == NULL) {
228         cout<<"ERROR_FAILED_TO_BIND, you should set up environment variables first."<<endl;
229         return ERROR_FAILED_TO_BIND;
230     }
231     // free the binderinfo
232     freeaddrinfo(binderinfo);  // all done with this
233 
234 
235     /* 3. Then create a socket to accept connection from clients */
236     // same codes as get binder addr info
237     FD_ZERO(&master);
238     FD_ZERO(&read_fds);
239     memset(&hints, 0, sizeof(hints));
240 
241     hints.ai_family = AF_UNSPEC;
242     hints.ai_socktype = SOCK_STREAM;
243     hints.ai_flags = AI_PASSIVE;
244 
245     if ((rv = getaddrinfo(NULL, "0", &hints, &ai)) != 0) {
246         return ERROR_INVALID_ADDR_INFO;
247     }
248 
249     for(p = ai; p != NULL; p = p->ai_next) {
250         if ((server_listen_fd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) < 0) {
251             continue;
252         }
253 
254         if (setsockopt(server_listen_fd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) < 0) {
255             continue;
256         }
257 
258         if (bind(server_listen_fd, p->ai_addr, p->ai_addrlen) < 0) {
259             close(server_listen_fd);
260             continue;
261         }
262 
263         break;
264     }
265 
266     if (p == NULL) {
267         return ERROR_FAILED_TO_BIND;
268     }
269 
270     freeaddrinfo(ai); // all done with this
271 
272     // get socket name
273     sockaddr* sa;
274     socklen_t* sl;
275     getsockname(server_listen_fd, sa, sl);
```

```
276
277     // listen
278     if (listen(server_listen_fd, MAX_CLIENTS) < 0) {
279         return ERROR_FAILED_TO_LISTEN;
280     }
281
282     // put listener to master set and set fdmax
283     FD_SET(server_listen_fd, &master);
284     fdmax = server_listen_fd;
285
286
287     /* 4. Retrieve server_port */
288     struct sockaddr_in sin;
289     socklen_t len = sizeof(sin);
290     if (getsockname(server_listen_fd, (struct sockaddr *)&sin, &len) < 0) {
291         return ERROR_FAILED_PORT_RETRIEVAL;
292     }
293
294     server_port = ntohs(sin.sin_port);
295
296     return 0;
297 }
298
299
300 int rpcCall(char* name, int* argTypes, void** args) {
301     int* argTypes_copy = argTypes;
302     void** args_copy = args;
303
304     vector <argTypeStruct> arg_types;
305
306     // 1. connect to binder and get server location
307     binder_fd = clientConnectBinder();
308     if (binder_fd < 0) return binder_fd;
309
310     int server_fd;
311     char msg;
312     do {
313         // send LOC_REQUEST to binder
314         msg = LOC_REQUEST;
315         send(binder_fd, &msg, sizeof(char), 0);
316         // then function name
317         // NOTE: add a trailing '\0' to the original name just to ensure no bug will occurs
318         // Assume that the given name is 127 characters or less.
319         int name_length = 128;
320         char *name_to_send = new char[name_length];
321         for (int c = 0; c < name_length - 1; c++) {
322             name_to_send[c] = name[c];
323         }
324         name_to_send[name_length - 1] = '\0';
325         send(binder_fd, name_to_send, name_length, 0);
326         delete [] name_to_send;
327         // argTypes
328         while(true) {
329             int temp_int = htonl( *argTypes_copy);
330             send(binder_fd, &temp_int, sizeof(temp_int), 0);
331             if (temp_int == 0) break;
332             argTypes_copy++;
333         }//while
334
335         // get server_id from binder
336         unsigned int serverPort;
337         char hostname[128];
338
339         int msg_int;
340         if (recv(binder_fd, &msg_int, sizeof(msg_int), 0) <= 0) {
341             return ERROR_NO_BINDER_AVALIABLE;
342         }
343         if (msg_int == LOC_SUCCESS) {
344             if( recv(binder_fd, &hostname, sizeof(char)*128, 0) <= 0) {
```

```
345                 return ERROR_NO_BINDER_AVALIABLE;
346             }
347 
348             if( recv(binder_fd, &serverPort, sizeof(serverPort), 0) <= 0) {
349                 return ERROR_NO_BINDER_AVALIABLE;
350             }
351 
352             serverPort = ntohl(serverPort);
353 
354         } else if (msg_int == LOC_FAILURE) { //LOC_FAILURE
355             recv(binder_fd, &serverPort, sizeof(serverPort), 0);
356             serverPort = ntohl(serverPort);
357             return ERROR_LOC_FAILURE;
358         }//end if (msg == LOC_SUCCESS)
359 
360         // try to connect server
361         if (hostname == NULL) return ERROR_CLIENT_FAILED_CONNECT_SERVER;
362 
363         server_fd = socket(AF_INET, SOCK_STREAM, 0);
364 
365         struct sockaddr_in hostaddr;
366         hostaddr.sin_family = AF_INET;
367         hostaddr.sin_port = htons(serverPort);
368 
369         struct hostent *host_ent;
370         if ((host_ent = gethostbyname(hostname)) == NULL) return ERROR_CLIENT_FAILED_CONNECT_SERVER;
371 
372         bcopy((char *) host_ent->h_addr, (char *) &hostaddr.sin_addr.s_addr, host_ent->h_length);
373 
374         if (connect(server_fd, (struct sockaddr *) &hostaddr, sizeof(hostaddr)) < 0) {
375             return ERROR_CLIENT_FAILED_CONNECT_SERVER;
376         }
377     } while (server_fd < 0);
378 
379     close(binder_fd);
380 
381     // 2. send execute msg to server
382     // send EXECUTE_MSG
383     msg = EXECUTE_MSG;
384     send(server_fd, &msg, sizeof(msg), 0);
385     // then function name
386     // NOTE: add a trailing '\0' to the original name just to ensure no bug will occurs
387     // Assume that the given name is 127 characters or less.
388     int name_length = 128;
389     char *name_to_send = new char[name_length];
390     for (int c = 0; c < name_length - 1; c++) {
391         name_to_send[c] = name[c];
392     }
393     name_to_send[name_length - 1] = '\0';
394     send(server_fd, name_to_send, name_length, 0);
395 
396     delete [] name_to_send;
397     // send argTypes, save into vector arg_types
398     argTypes_copy = argTypes;
399     while(true) {
400         int temp_int = *argTypes_copy;
401         int send_int = htonl(temp_int);
402         send(server_fd, &send_int, sizeof(send_int), 0);
403 
404         if (temp_int == 0) break;
405 
406         argTypeStruct *temp_argTypeStruct = new argTypeStruct(temp_int);
407         arg_types.push_back(*temp_argTypeStruct);
408         argTypes_copy++;
409     }
410     /* send args */
411     // first compute the send buffer length
412     unsigned int length_input = 0;
413     unsigned int length_output = 0;
```

```
414
415     for (vector<argTypeStruct>::iterator it = arg_types.begin(); it != arg_types.end(); it++) {
416         argTypeStruct arg = *it;
417         if (arg.input) {
418             switch (arg.type) {
419                 case ARG_CHAR:
420                     length_input += arg.array_length * sizeof(char);
421                     break;
422                 case ARG_SHORT:
423                     length_input += arg.array_length * sizeof(short);
424                     break;
425                 case ARG_INT:
426                     length_input += arg.array_length * sizeof(int);
427                     break;
428                 case ARG_LONG:
429                     length_input += arg.array_length * sizeof(long);
430                     break;
431                 case ARG_DOUBLE:
432                     length_input += arg.array_length * sizeof(double);
433                     break;
434                 case ARG_FLOAT:
435                     length_input += arg.array_length * sizeof(float);
436                     break;
437                 default: {
438                     std::cout<< "ERROR in a thread of rpcExecute: arg type not known." << std::endl;
439                     return ERROR_BAD_ARG_TYPE;
440                 }
441             }
442         }//end if (arg.input)
443
444         if (arg.output) {
445             switch (arg.type) {
446                 case ARG_CHAR:
447                     length_output += arg.array_length * sizeof(char);
448                     break;
449                 case ARG_SHORT:
450                     length_output += arg.array_length * sizeof(short);
451                     break;
452                 case ARG_INT:
453                     length_output += arg.array_length * sizeof(int);
454                     break;
455                 case ARG_LONG:
456                     length_output += arg.array_length * sizeof(long);
457                     break;
458                 case ARG_DOUBLE:
459                     length_output += arg.array_length * sizeof(double);
460                     break;
461                 case ARG_FLOAT:
462                     length_output += arg.array_length * sizeof(float);
463                     break;
464                 default: {
465                     std::cout<< "ERROR in a thread of rpcExecute: arg type not known." << std::endl;
466                     return ERROR_BAD_ARG_TYPE;
467                 }
468             }
469         }//end if (arg.output)
470
471     }//end for loop
472
473     // malloc the buffer
474     unsigned char * input_args = (unsigned char *) malloc(length_input * sizeof(char));
475     unsigned char * output_args = (unsigned char *) malloc(length_output * sizeof(char));
476     unsigned char * buffer_cursor = input_args;
477
478     // copy from args to send buffer(which is input_args)
479     for (vector<argTypeStruct>::iterator it = arg_types.begin(); it != arg_types.end(); it++) {
480         argTypeStruct arg = *it;
481         if (arg.input) {
482             switch (arg.type) {
```

```
483                 case (ARG_CHAR): {
484                     char* temp_buffer = (char* )(*args_copy);
485                     memcpy(buffer_cursor, temp_buffer, arg.array_length * sizeof(char));
486                     buffer_cursor += arg.array_length * sizeof(char);
487                     break;
488                 }
489                 case (ARG_SHORT): {
490                     short* temp_buffer = (short* )(*args_copy);
491                     memcpy(buffer_cursor, temp_buffer, arg.array_length * sizeof(short));
492                     buffer_cursor += arg.array_length * sizeof(short);
493                     break;
494                 }
495                 case (ARG_INT): {
496                     int* temp_buffer = (int* )(*args_copy);
497                     memcpy(buffer_cursor, temp_buffer, arg.array_length * sizeof(int));
498                     buffer_cursor += arg.array_length * sizeof(int);
499                     break;
500                 }
501                 case (ARG_LONG): {
502                     long* temp_buffer = (long* )(*args_copy);
503                     memcpy(buffer_cursor, temp_buffer, arg.array_length * sizeof(long));
504                     buffer_cursor += arg.array_length * sizeof(long);
505                     break;
506                 }
507                 case (ARG_DOUBLE): {
508                     double* temp_buffer = (double* )(*args_copy);
509                     memcpy(buffer_cursor, temp_buffer, arg.array_length * sizeof(double));
510                     buffer_cursor += arg.array_length * sizeof(double);
511                     break;
512                 }
513                 case (ARG_FLOAT): {
514                     float* temp_buffer = (float* )(*args_copy);
515                     memcpy(buffer_cursor, temp_buffer, arg.array_length * sizeof(float));
516                     buffer_cursor += arg.array_length * sizeof(float);
517                     break;
518                 }
519                 default: {
520                     std::cout<< "ERROR in rpcCall: arg type not known." << std::endl;
521                     return ERROR_BAD_ARG_TYPE;
522                 }
523             }
524         }//end if (arg.input)
525 
526         args_copy++;
527     }//end for loop
528 
529     // send the input_args buffer
530     send(server_fd, input_args, length_input, 0);
531 
532     // 3. send done. wait for response
533     if ( recv(server_fd, &msg, sizeof(msg), 0) <= 0) return ERROR_CLIENT_FAILED_CONNECT_SERVER;
534     if ( msg == EXECUTE_FAILURE) {
535         int error_code;
536         if ( recv(server_fd, &error_code, sizeof(error_code), 0) <= 0) return ERROR_CLIENT_FAILED_CONNECT_SERVER;
537         return ntohl(error_code);
538     }
539 
540     // 4. when EXECUTE_SUCCESS, receive args
541     if ( msg == EXECUTE_SUCCESS) {
542         if ( recv(server_fd, output_args, length_output, 0) <= 0) return ERROR_CLIENT_FAILED_CONNECT_SERVER;
543         buffer_cursor = output_args;
544 
545         for (int i = 0; i< arg_types.size(); i++) {
546             argTypeStruct arg = arg_types[i];
547             if (arg.output) {
548                 switch (arg.type) {
549                     case (ARG_CHAR): {
550                         char *temp_buffer = (char *)malloc(arg.array_length * sizeof(char));
551                         memcpy( temp_buffer, buffer_cursor, arg.array_length * sizeof(char));
```

```
552                         buffer_cursor += arg.array_length * sizeof(char);
553                         for (int j=0; j< arg.array_length; j++) {
554                             ((char *)args[i]) [j] = temp_buffer[j];
555                         }
556                         free(temp_buffer);
557                         break;
558                     }
559                     case (ARG_SHORT): {
560                         short *temp_buffer = (short *)malloc(arg.array_length * sizeof(short));
561                         memcpy( temp_buffer, buffer_cursor, arg.array_length * sizeof(short));
562                         buffer_cursor += arg.array_length * sizeof(short);
563                         for (int j=0; j< arg.array_length; j++) {
564                             ((short *)args[i]) [j] = temp_buffer[j];
565                         }
566                         free(temp_buffer);
567                         break;
568                     }
569                     case (ARG_INT): {
570                         int *temp_buffer = (int *)malloc(arg.array_length * sizeof(int));
571                         memcpy( temp_buffer, buffer_cursor, arg.array_length * sizeof(int));
572                         buffer_cursor += arg.array_length * sizeof(int);
573                         for (int j=0; j< arg.array_length; j++) {
574                             ((int *)args[i]) [j] = temp_buffer[j];
575                         }
576                         free(temp_buffer);
577                         break;
578                     }
579                     case (ARG_LONG): {
580                         long *temp_buffer = (long *)malloc(arg.array_length * sizeof(long));
581                         memcpy( temp_buffer, buffer_cursor, arg.array_length * sizeof(long));
582                         buffer_cursor += arg.array_length * sizeof(long);
583                         for (int j=0; j< arg.array_length; j++) {
584                             ((long *)args[i]) [j] = temp_buffer[j];
585                         }
586                         free(temp_buffer);
587                         break;
588                     }
589                     case (ARG_DOUBLE): {
590                         double *temp_buffer = (double *)malloc(arg.array_length * sizeof(double));
591                         memcpy( temp_buffer, buffer_cursor, arg.array_length * sizeof(double));
592                         buffer_cursor += arg.array_length * sizeof(double);
593                         for (int j=0; j< arg.array_length; j++) {
594                             ((double *)args[i]) [j] = temp_buffer[j];
595                         }
596                         free(temp_buffer);
597                         break;
598                     }
599                     case (ARG_FLOAT): {
600                         float *temp_buffer = (float *)malloc(arg.array_length * sizeof(float));
601                         memcpy( temp_buffer, buffer_cursor, arg.array_length * sizeof(float));
602                         buffer_cursor += arg.array_length * sizeof(float);
603                         for (int j=0; j< arg.array_length; j++) {
604                             ((float *)args[i]) [j] = temp_buffer[j];
605                         }
606                         free(temp_buffer);
607                         break;
608                     }
609                     default: {
610                         std::cout<< "ERROR in rpcCall: arg type not known." << std::endl;
611                         return ERROR_BAD_ARG_TYPE;
612                     }
613                 }
614             }//end if (arg.input)
615         }//end for loop
616     }// end if ( msg == EXECUTE_SUCCESS)
617 
618     close(server_fd);
619     free(input_args);
620     free(output_args);
```

```
621
622     return 0;
623 }
624
625
626 int rpcCacheCall(char* name, int* argTypes, void** args) {return 0;}
627
628
629 int rpcRegister(char* name, int* argTypes, skeleton f){
630
631     /* 1. Send register info to binder
632      * format: REGISTER, server_identifier, port, name, argTypes
633      */
634
635     // first send a character 'r' to notify the binder a register msg is coming
636     char register_msg = REGISTER_MSG;
637     send(binder_fd, &register_msg, sizeof(register_msg), 0);
638
639     // then send the server_identifier
640     send(binder_fd, server_hostname, 128 * sizeof(char), 0);
641
642     // then the server_port
643     int tmp_server_port = htonl((uint32_t)server_port);
644     send(binder_fd, &tmp_server_port, sizeof(tmp_server_port), 0);
645
646     // then function name
647     // NOTE: add a trailing '\0' to the original name just to ensure no bug will occurs
648     // Assume that the given name is 127 characters or less.
649     int name_length = 128;
650     char *name_to_send = new char[name_length];
651     for (int c = 0; c < name_length - 1; c++) {
652         name_to_send[c] = name[c];
653     }
654     name_to_send[name_length - 1] = '\0';
655     send(binder_fd, name_to_send, name_length, 0);
656     delete [] name_to_send;
657
658     // then the argTypes, send one by one
659     int *type = argTypes;
660     int tmp;
661     while (*type != 0) {
662         tmp = htonl((uint32_t)*type);
663         send(binder_fd, &tmp, sizeof(tmp), 0);
664         type++;
665     }
666     tmp = htonl((uint32_t)*type); // this is the trailing 0
667     send(binder_fd, &tmp, sizeof(tmp), 0);
668
669     /* 2. Recieve register response from binder
670      * format: first int {REGISTER_SUCCESS,REGISTER_FAILURE}
671      *         second int indicate warnings or errors type ( 0 when success)
672      */
673     unsigned int binder_response_result;
674     unsigned int rcv_status_code;
675
676     // receive response result
677     if (recv(binder_fd, &binder_response_result, sizeof(binder_response_result), 0) <= 0) {
678         return ERROR_REGISTER;
679     }
680     // receive status code
681     if (recv(binder_fd, &rcv_status_code, sizeof(rcv_status_code), 0) < 0) {
682         return ERROR_REGISTER;
683     }
684     binder_response_result = ntohl(binder_response_result);
685     rcv_status_code = ntohl(rcv_status_code);
686
687     if (rcv_status_code < 0) return rcv_status_code; // when it's an error
688
689     /* 3. Register locally
```

```
690      *
691      */
692     string name_string(name);
693     vector <argTypeStruct> arg_types;
694 
695     for (type = argTypes; *type != 0; type++) {
696         argTypeStruct *temp_argTypeStruct = new argTypeStruct( *type);
697         arg_types.push_back(*temp_argTypeStruct);
698     }
699 
700     signature *sig = new signature(name_string, arg_types);
701 
702     //if signature not exist, store it
703     auto it = function_map.find(*sig);
704     if(it == function_map.end()) {
705         function_map.insert( pair<signature, skeleton> (*sig,f) );
706         //cout<<"Local register done. "<< string(name) <<endl;
707     } else {
708         if (it->second != f) {
709             it->second = f;
710             //cout<<"Duplicate function signature. Override the skeleton."<<endl;
711         } else {
712             //cout<<"Duplicate function signature and skeleton. Do nothing."<<endl;
713         }
714     }
715 
716     return 0;
717 }
718 
719 
720 void * executeThread (void * parms) {
721     int fd = (intptr_t) parms;
722 
723     // 1. get name
724     char *name = new char [128];
725     if (recv(client_side_fd, name, 128, 0) <= 0) {
726         free(name);
727         std::cout<< "ERROR in a thread of rpcExecute: get client host name." << std::endl;
728         return NULL;
729     }
730     string name_string(name);
731     free(name);
732 
733     // 2. get argTypes
734     vector <argTypeStruct> arg_types;
735 
736     while (1) { //loop until get 0
737         int arg_type;
738         if (recv(client_side_fd, &arg_type, sizeof(int), 0) <= 0) {
739             std::cout<< "ERROR in a thread of rpcExecute: client hung up." << std::endl;
740             return NULL;
741         }
742         arg_type = ntohl(arg_type);
743 
744         if (arg_type == 0) {
745             break;
746         }
747 
748         argTypeStruct *temp_argTypeStruct = new argTypeStruct(arg_type);
749         arg_types.push_back(*temp_argTypeStruct);
750     }
751 
752     // 3. bulid signature
753     signature *sig = new signature(name_string, arg_types);
754 
755     // 4. check if signature exist, if exist get skeleton
756     auto it = function_map.find(*sig);
757     if (it == function_map.end()) {
758         std::cout<< "ERROR in a thread of rpcExecute: function not exist." << std::endl;
```

```
759         return NULL;
760     }
761     skeleton f = it->second;
762 
763     // 5. malloc memory for input args and output args
764     unsigned int length_input = 0;
765     unsigned int length_output = 0;
766 
767     for (vector<argTypeStruct>::iterator it = arg_types.begin(); it != arg_types.end(); it++) {
768         argTypeStruct arg = *it;
769         if (arg.input) {
770             switch (arg.type) {
771                 case ARG_CHAR:
772                     length_input += arg.array_length * sizeof(char);
773                     break;
774                 case ARG_SHORT:
775                     length_input += arg.array_length * sizeof(short);
776                     break;
777                 case ARG_INT:
778                     length_input += arg.array_length * sizeof(int);
779                     break;
780                 case ARG_LONG:
781                     length_input += arg.array_length * sizeof(long);
782                     break;
783                 case ARG_DOUBLE:
784                     length_input += arg.array_length * sizeof(double);
785                     break;
786                 case ARG_FLOAT:
787                     length_input += arg.array_length * sizeof(float);
788                     break;
789                 default: {
790                     std::cout<< "ERROR in a thread of rpcExecute: arg type not known." << std::endl;
791                     return NULL;
792                 }
793             }
794         }//end if (arg.input)
795 
796         if (arg.output) {
797             switch (arg.type) {
798                 case ARG_CHAR:
799                     length_output += arg.array_length * sizeof(char);
800                     break;
801                 case ARG_SHORT:
802                     length_output += arg.array_length * sizeof(short);
803                     break;
804                 case ARG_INT:
805                     length_output += arg.array_length * sizeof(int);
806                     break;
807                 case ARG_LONG:
808                     length_output += arg.array_length * sizeof(long);
809                     break;
810                 case ARG_DOUBLE:
811                     length_output += arg.array_length * sizeof(double);
812                     break;
813                 case ARG_FLOAT:
814                     length_output += arg.array_length * sizeof(float);
815                     break;
816                 default: {
817                     std::cout<< "ERROR in a thread of rpcExecute: arg type not known." << std::endl;
818                     return NULL;
819                 }
820             }
821         }//end if (arg.output)
822 
823     }//end for loop
824 
825     void **args = (void **)malloc(arg_types.size() * sizeof(void*));
826     unsigned char * input_args = (unsigned char *) malloc(length_input * sizeof(char));
827     unsigned char * output_args = (unsigned char *) malloc(length_output * sizeof(char));
```

```
828     unsigned char * buffer_cursor;
829
830     // 6. get args from clients
831     if ( recv(client_side_fd, input_args, length_input, 0) <=0 ) {
832         std::cout<< "ERROR in a thread of rpcExecute: client hung up." << std::endl;
833         return NULL;
834     }
835
836     // 7. copy memory from input_args into args
837     buffer_cursor = input_args;
838     for (int i = 0; i< arg_types.size(); i++) {
839         argTypeStruct arg = arg_types[i];
840
841         switch (arg.type) {
842             case (ARG_CHAR): {
843                 char *temp_buffer = (char *)malloc(arg.array_length * sizeof(char));
844                 if (arg.input) {
845                     memcpy( temp_buffer, buffer_cursor, arg.array_length * sizeof(char));
846                     buffer_cursor += arg.array_length * sizeof(char);
847                 }
848                 args[i] = temp_buffer;
849                 break;
850             }
851
852             case (ARG_SHORT): {
853                 short *temp_buffer = (short *)malloc(arg.array_length * sizeof(short));
854                 if (arg.input) {
855                     memcpy( temp_buffer, buffer_cursor, arg.array_length * sizeof(short));
856                     buffer_cursor += arg.array_length * sizeof(short);
857                 }
858                 args[i] = temp_buffer;
859                 break;
860             }
861
862             case (ARG_INT): {
863                 int *temp_buffer = (int *)malloc(arg.array_length * sizeof(int));
864                 if (arg.input) {
865                     memcpy( temp_buffer, buffer_cursor, arg.array_length * sizeof(int));
866                     buffer_cursor += arg.array_length * sizeof(int);
867                 }
868                 args[i] = temp_buffer;
869                 break;
870             }
871
872             case (ARG_LONG): {
873                 long *temp_buffer = (long *)malloc(arg.array_length * sizeof(long));
874                 if (arg.input) {
875                     memcpy( temp_buffer, buffer_cursor, arg.array_length * sizeof(long));
876                     buffer_cursor += arg.array_length * sizeof(long);
877                 }
878                 args[i] = temp_buffer;
879                 break;
880             }
881
882             case (ARG_DOUBLE): {
883                 double *temp_buffer = (double *)malloc(arg.array_length * sizeof(double));
884                 if (arg.input) {
885                     memcpy( temp_buffer, buffer_cursor, arg.array_length * sizeof(double));
886                     buffer_cursor += arg.array_length * sizeof(double);
887                 }
888                 args[i] = temp_buffer;
889                 break;
890             }
891
892             case (ARG_FLOAT): {
893                 float *temp_buffer = (float *)malloc(arg.array_length * sizeof(float));
894                 if (arg.input) {
895                     memcpy( temp_buffer, buffer_cursor, arg.array_length * sizeof(float));
896                     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));
```

```
966                         buffer_cursor += arg.array_length * sizeof(long);
967                     }
968                     break;
969                 }
970 
971                 case (ARG_DOUBLE): {
972                     if (arg.output) {
973                         double *temp_buffer = (double *) args[i];
974                         memcpy(buffer_cursor, temp_buffer, arg.array_length * sizeof(double));
975                         buffer_cursor += arg.array_length * sizeof(double);
976                     }
977                     break;
978                 }
979 
980                 case (ARG_FLOAT): {
981                     if (arg.output) {
982                         float *temp_buffer = (float *) args[i];
983                         memcpy(buffer_cursor, temp_buffer, arg.array_length * sizeof(float));
984                         buffer_cursor += arg.array_length * sizeof(float);
985                     }
986                     break;
987                 }
988 
989                 default: {
990                     std::cout<< "ERROR in a thread of rpcExecute: arg type not known." << std::endl;
991                     return NULL;
992                 }
993             }//switch
994         }//end for loop
995 
996         send(client_side_fd,output_args,length_output,0);
997 
998     }//end if else
999 
1000     // 10. free memory
1001     delete [] temp_args;
1002     free (input_args);
1003     free (output_args);
1004 
1005     for (int i = 0; i< arg_types.size(); i++) {
1006         free( args[i] );
1007     }
1008     free(args);
1009 
1010     // 11. remove from master
1011     pthread_mutex_lock(&mutex);
1012         close(fd);
1013         FD_CLR(fd, &master);
1014     pthread_mutex_unlock(&mutex);
1015 }
1016 
1017 
1018 int rpcExecute(){
1019 
1020     int listener;     // listening socket descriptor
1021     int newfd;        // newly accept()ed socket descriptor
1022     struct sockaddr_storage remoteaddr; // client address
1023     socklen_t addrlen;
1024 
1025     char buf[256];    // buffer for client data
1026     int nbytes;
1027 
1028     char remoteIP[INET6_ADDRSTRLEN];
1029 
1030     int yes=1;        // for setsockopt() SO_REUSEADDR, below
1031     int i, rv;
1032 
1033     struct addrinfo hints, *ai, *p;
1034 
```

```
1035    // add the listener to the master set
1036    FD_SET(binder_fd, &master);
1037
1038    // main loop
1039    while(1) {
1040        read_fds = master; // copy it
1041
1042        if (select(fdmax+1, &read_fds, NULL, NULL, NULL) == -1) {
1043            return ERROR_FAILED_TO_SELECT;
1044        }
1045        // run through the existing connections looking for data to read
1046        for(i = 0; i <= fdmax; i++) {
1047            if (FD_ISSET(i, &read_fds)) { // we got one!!
1048                if (i == server_listen_fd) {
1049                    // handle new connections
1050                    addrlen = sizeof remoteaddr;
1051                    client_side_fd = accept(server_listen_fd,
1052                                    (struct sockaddr *)&remoteaddr,
1053                                    &addrlen);
1054                    if (client_side_fd == -1) {
1055                        return ERROR_FAILED_TO_ACCEPT;
1056                    } else {
1057                        pthread_mutex_lock(&mutex);
1058                            FD_SET(client_side_fd, &master); // add to master set
1059                            if (client_side_fd > fdmax) {    // keep track of the max
1060                                fdmax = client_side_fd;
1061                            }
1062                        pthread_mutex_unlock(&mutex);
1063
1064                    }
1065                } else {
1066                    // handle data from a client
1067                    char msg_type;
1068                    if (recv(i, &msg_type, sizeof(msg_type), 0) <= 0) {
1069                        pthread_mutex_lock(&mutex);
1070                            close(i); // bye!
1071                            FD_CLR(i, &master); // remove from master set
1072                        pthread_mutex_unlock(&mutex);
1073                        return ERROR_BINDER_CLOSED;
1074                    } else {
1075                        switch (msg_type) {
1076                            case EXECUTE_MSG: {
1077                                // only client can call execute
1078                                if (i != binder_fd){
1079                                    pthread_create(&(pthreadArray[i]), NULL, &executeThread, (void *) (intptr_t) i);
1080                                    FD_CLR(i, &master);
1081                                }
1082                                break;
1083                            }
1084                            case TERMINATE_MSG: {
1085                                // only binder can call terminate
1086                                if (i == binder_fd){
1087                                    printf("Terminate message from %d .\n", i);
1088                                    for (int ii = 0; ii < MAX_CLIENTS; ii++) {
1089                                        pthread_join( pthreadArray[ii] , NULL );
1090                                    }
1091                                    //free(pthreadArray);
1092                                    return 0; // successfully terminated
1093                                }
1094                                break;
1095                            }
1096                            default:{
1097                                printf("Invalid message. Just ignored. \n");
1098                                break;
1099                            }
1100                        }//switch
1101                    }
1102                } // END handle data from client
1103            } // 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;
}
```