```
/*
    This code used Beej's sample server code starting point and was extended to
    fit my needs for the assignment because the tutorial and sample code were
    provided to us from the course website.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <vector>
#include <string>
#include <iostream>

#define PORT "0"   // port we're listening on
#define REGISTER_FAILURE 0
#define REGISTER_SUCCESS 1
#define REGISTER_DUPLICATE 2
#define LOC_REQUEST 'c'
#define LOC_SUCCESS 1
#define LOC_FAILURE 0
#define TERMINATE 't'

struct registration{
    char server_identifier[128];
    int server_port;
    char procedure_name[128];
    int* arg_types;
    int robin_value;
} registration_struct;

std::vector<registration> registrations;
int addRegistration(struct registration newRegistration){
    // Add the registration to the registrations vector
    // Todo: Check for dupes, return a sensible error code.
    std::vector<registration>::size_type i;
    bool shouldBeRegistered = true;
    int duplicateIndex = -1;
    for (i = 0; i != registrations.size(); i++){
        bool same_identifier = (strcmp(newRegistration.server_identifier, registrations[i].server_identifier) == 0);
        bool same_port = (newRegistration.server_port == registrations[i].server_port);
        bool same_proc_name = (strcmp(newRegistration.procedure_name, registrations[i].procedure_name) == 0);
        // If any of the identifier, port, and proc name are the different, it's
        // not a duplicate. Move on.
        if (!same_identifier || !same_port || !same_proc_name){
            continue;
        }
        // Check if the argtypes are the same. Argtypes are integers with a 0 at
        // the end to indicate the end of the list.
        int j = 0;
        bool continueChecking = true;
        while (newRegistration.arg_types[j] != 0 && registrations[i].arg_types[j] != 0 && continueChecking == true){
            if (newRegistration.arg_types[j] != registrations[i].arg_types[j]){
                // If the args list is different at any point, leave the while loop
                // and move on.
                continueChecking = false;
            }
            j++;
        }
        // If a difference was found, move on.
        if (continueChecking == false){
            continue;
        }
        // We only reach this part of the loop if no differences have been found
        // yet and the end of one or both of the arg_types arrays has been reached.
        // If they're equal, they must necessarily both be 0 (otherwise we'd still
        // be in the while loop above). If this is the case, the new registration
        // is a duplicate. Otherwise, it's not a duplicate and we push the new
        // one onto the list.
```

```
75          if (newRegistration.arg_types[j] == registrations[i].arg_types[j]){
76              duplicateIndex = i;
77              shouldBeRegistered = false;
78              break;
79          }
80      }
81      if (shouldBeRegistered == false){
82          // Delete the existing entry, then find a suitable robin value and
83          // push_back as normal.
84          registrations.erase(registrations.begin() + duplicateIndex);
85          for (i = 0; i < registrations.size(); i++){
86              if (strcmp(newRegistration.server_identifier, registrations[i].server_identifier) == 0
87                  && newRegistration.server_port == registrations[i].server_port){
88                  // If the server_id and server_port are equal, it's the same server.
89                  // use this robin_value and break since all RVs should be the same
90                  // for all registrations of a single server.
91                  newRegistration.robin_value = registrations[i].robin_value;
92                  break;
93              }
94          }
95          registrations.push_back(newRegistration);
96          return REGISTER_DUPLICATE;
97      } else {
98          // Check if a registration already exists for this server. If it does,
99          // we need to set newRegistration.robin_value to the existing RV.
100         for (i = 0; i < registrations.size(); i++){
101             if (strcmp(newRegistration.server_identifier, registrations[i].server_identifier) == 0
102                 && newRegistration.server_port == registrations[i].server_port){
103                 // If the server_id and server_port are equal, it's the same server.
104                 // use this robin_value and break since all RVs should be the same
105                 // for all registrations of a single server.
106                 newRegistration.robin_value = registrations[i].robin_value;
107                 break;
108             }
109         }
110         registrations.push_back(newRegistration);
111         return REGISTER_SUCCESS;
112     }
113 }
114 /* Note that this returns structs not pointers to structs */
115 registration getRegistration(char* procedure_name, int* arg_types){
116     std::vector<registration>::size_type i;
117     int candidateRobinValue = 0;
118     int candidateIndex = -1;
119     for(i = 0; i != registrations.size(); i++){
120         // If the procedure names are different move on
121         if (strcmp(registrations[i].procedure_name, procedure_name) != 0){
122             continue;
123         }
124         // If any of the arg_types are different, move on.
125         int j = 0;
126         bool continueChecking = true;
127         while(registrations[i].arg_types[j] != 0 && arg_types[j] != 0 && continueChecking == true){
128             if (registrations[i].arg_types[j] != arg_types[j]){
129                 continueChecking = false;
130             }
131             j++;
132         }
133         // A difference was found in the argtypes, keep going.
134         if (continueChecking == false){
135             continue;
136         }
137         // We only reach this part of the loop if there's no difference in the
138         // proc name and no difference in the arg types. In this case, we've found
139         // a server that can handle the request. Update the candidate if its
140         // robin value is lower than the current candidate's robin value.
141         // We also want to update the candidate if one hasn't been found yet.
142         if (registrations[i].robin_value < candidateRobinValue || candidateIndex == -1){
143             // Replace the candidate with i.
144             candidateIndex = i;
145             candidateRobinValue = registrations[i].robin_value;
146         }
147     }
148     // The server info to return has been found. Build a new struct and return it.
149     if (candidateIndex != -1){
```

```
150         int newRobinValue = registrations[candidateIndex].robin_value + 1;
151         // Loop through the registrations and for each registration that's the
152         // same server (same id && port) increment the robin value.
153         // Note that this assumes that each server's robin value is kept in sync
154         // both here and when adding registrations.
155         for (i = 0; i != registrations.size(); i++){
156             bool sameName = (strcmp(registrations[i].server_identifier, registrations[candidateIndex].server_identifier) == 0);
157             bool samePort = (registrations[i].server_port == registrations[candidateIndex].server_port);
158             if (sameName && samePort){
159                 registrations[i].robin_value = newRobinValue;
160             }
161         }
162         return registrations[candidateIndex];
163     } else {
164         registration noneFound;
165         // use the robin_value as a flag for "not found"since -1 is never valid.
166         noneFound.robin_value = -1;
167         return noneFound;
168     }
169 }
170 
171 // get sockaddr, IPv4 or IPv6:
172 void *get_in_addr(struct sockaddr *sa)
173 {
174     if (sa->sa_family == AF_INET) {
175         return &(((struct sockaddr_in*)sa)->sin_addr);
176     }
177 
178     return &(((struct sockaddr_in6*)sa)->sin6_addr);
179 }
180 
181 void testRegistration(){
182     std::cout << "Beginning registration test" << std::endl;
183 
184     registration dummyReg;
185     char dummyIdentifier[] = "dummy_identifier";
186     strcpy(dummyReg.server_identifier, dummyIdentifier);
187     dummyReg.server_port = 1234;
188     char dummyProcName[] = "dummy_proc";
189     strcpy(dummyReg.procedure_name, dummyProcName);
190     dummyReg.robin_value = 0;
191     int dummyArgs[5] = {1, 2, 3, 4, 0};
192     dummyReg.arg_types = &dummyArgs[0];
193     addRegistration(dummyReg);
194 
195     // Create a 2nd dummy registration with the same sig but different ID
196     registration dummyReg2;
197     char dummyIdentifier2[] = "dummy_identifier2";
198     strcpy(dummyReg2.server_identifier, dummyIdentifier2);
199     dummyReg2.server_port = 1234;
200     char dummyProcName2[] = "dummy_proc";
201     strcpy(dummyReg2.procedure_name, dummyProcName2);
202     dummyReg2.robin_value = 0;
203     int dummyArgs2[5] = {1, 2, 3, 4, 0};
204     dummyReg2.arg_types = &dummyArgs2[0];
205     addRegistration(dummyReg2);
206 
207     registration testReg = getRegistration(&dummyProcName[0], &dummyArgs[0]);
208     std::cout << testReg.server_identifier << std::endl;
209     testReg = getRegistration(&dummyProcName[0], &dummyArgs[0]);
210     std::cout << testReg.server_identifier << std::endl;
211     std::cout << "The two values should be different" << std::endl;
212 }
213 int main(void)
214 {
215     //testRegistration();
216     fd_set master;    // master file descriptor list
217     fd_set read_fds;  // temp file descriptor list for select()
218     int fdmax;        // maximum file descriptor number
219 
220     int listener;     // listening socket descriptor
221     int newfd;        // newly accept()ed socket descriptor
222     struct sockaddr_storage remoteaddr; // client address
223     socklen_t addrlen;
224 
```

```
225     char buf[1];    // buffer for client data
226     int nbytes;
227
228     char remoteIP[INET6_ADDRSTRLEN];
229
230     int yes=1;        // for setsockopt() SO_REUSEADDR, below
231     int i, rv;
232
233     struct addrinfo hints, *ai, *p;
234
235     FD_ZERO(&master);    // clear the master and temp sets
236     FD_ZERO(&read_fds);
237
238     // Grab the local machine's hostname
239     char hostname[1024];
240     hostname[1023] = '\0';
241     gethostname(hostname, 1023);
242     printf("BINDER_ADDRESS %s \n", hostname);
243
244     // get us a socket and bind it
245     memset(&hints, 0, sizeof hints);
246     hints.ai_socktype = SOCK_STREAM;
247     hints.ai_family = AF_UNSPEC;
248     hints.ai_flags = AI_PASSIVE;
249
250     if ((rv = getaddrinfo(NULL, PORT, &hints, &ai)) != 0) {
251         //fprintf(stderr, "Binder: %s\n", gai_strerror(rv));
252         exit(1);
253     }
254
255     for(p = ai; p != NULL; p = p->ai_next) {
256         // p is a linked list. Whenever p successfully calls socket and bind,
257         // the break statement will be reached and the loop terminated. If the
258         // loop terminates on its own because p == NULL, then we didn't get bound.
259         listener = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
260         if (listener < 0) {
261             continue;
262         }
263
264         // lose the pesky "address already in use" error message
265         //setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));
266         if (bind(listener, p->ai_addr, p->ai_addrlen) < 0) {
267             close(listener);
268             continue;
269         }
270
271     if (p == NULL) {
272         // if we got here, it means we didn't get bound
273         //fprintf(stderr, "Binder: failed to bind\n");
274         exit(2);
275     }
276     // listen
277     if (listen(listener, 10) == -1) {
278         //perror("listen");
279         exit(3);
280     }
281     connect(listener, p->ai_addr, p->ai_addrlen);
282
283     struct sockaddr_in sin = *((struct sockaddr_in*) p->ai_addr);
284     socklen_t len = sizeof(sin);
285     if (getsockname(listener, (struct sockaddr *)&sin, &len) == -1){
286         //perror("Binder: getsockname failed");
287     } else {
288         printf("BINDER_PORT %i \n", (int) ntohs(sin.sin_port));
289     }
290         break;
291     }
292
293     freeaddrinfo(ai); // all done with this
294
295     // add the listener to the master set
296     FD_SET(listener, &master);
297
298     // keep track of the biggest file descriptor
299     fdmax = listener; // so far, it's this one
```

```
300
301     // main loop
302     for(;;) {
303         read_fds = master; // copy it
304         if (select(fdmax+1, &read_fds, NULL, NULL, NULL) == -1) {
305             //perror("select");
306             exit(4);
307         }
308
309         // run through the existing connections looking for data to read
310         for(i = 0; i <= fdmax; i++) {
311             if (FD_ISSET(i, &read_fds)) { // we got one!!
312                 if (i == listener) {
313                     // handle new connections
314                     addrlen = sizeof remoteaddr;
315                     newfd = accept(listener,
316                         (struct sockaddr *)&remoteaddr,
317                         &addrlen);
318
319                     if (newfd == -1) {
320                         //perror("accept");
321                     } else {
322                         FD_SET(newfd, &master); // add to master set
323                         if (newfd > fdmax) {    // keep track of the max
324                             fdmax = newfd;
325                         }
326                         /*
327                         printf("selectserver: new connection from %s on "
328                             "socket %d\n",
329                             inet_ntop(remoteaddr.ss_family,
330                                 get_in_addr((struct sockaddr*)&remoteaddr),
331                                 remoteIP, INET6_ADDRSTRLEN),
332                             newfd);
333                         */
334                     }
335                 } else {
336                     // handle data from a client
337                     if ((nbytes = recv(i, buf, sizeof buf, 0)) <= 0) {
338                         // got error or connection closed by client
339                         if (nbytes == 0) {
340                             // connection closed
341                             //printf("selectserver: socket %d hung up\n", i);
342                         } else {
343                             //perror("recv");
344                         }
345                         close(i); // bye!
346                         FD_CLR(i, &master); // remove from master set
347                     } else {
348                         //printf("%s\n", buf);
349                         // buf contains 'r' if it's from a server
350                         if (buf[0] == 'r'){
351                             // Grab the various pieces of information from the
352                             // buffer.
353                             char server_identifier[128];
354                             int server_port[1];
355                             char procedure_name[128];
356                             std::vector<uint32_t> argtypes_vector;
357                             recv(i, server_identifier, sizeof server_identifier, 0);
358                             recv(i, server_port, sizeof server_port, 0);
359                             recv(i, procedure_name, sizeof procedure_name, 0);
360
361                             // Since we don't know how many args there will be,
362                             // just keep grabbing args until an arg is 0 (this is
363                             // not valid input so we can safely use it as a "no
364                             // more args" signal)
365                             int tmp[1];
366                             do{
367                                 recv(i, tmp, sizeof tmp, 0);
368                                 argtypes_vector.push_back(tmp[0]);
369                             }while(tmp[0] != 0);
370                             // Turn the argtypes vector into an array.
371                             int argtypes[argtypes_vector.size()];
372                             std::copy(argtypes_vector.begin(), argtypes_vector.end(), argtypes);
373
374                             // Pack all this data into a registration struct
```

```
375                             registration new_registration;
376                             strcpy(new_registration.server_identifier, server_identifier);
377                             new_registration.server_port = server_port[0];
378                             strcpy(new_registration.procedure_name, procedure_name);
379                             new_registration.arg_types = &argtypes[0];
380                             new_registration.robin_value = 0;
381                             // Register the struct!
382                             int result = addRegistration(new_registration);
383                             if (result == REGISTER_SUCCESS){
384                                 int tmp = 0;
385                                 if(send(i, &tmp, sizeof(tmp), 0) < 0){
386                                     //perror("send error");
387                                 }
388                                 // No errors, so a 0 is sent.
389                                 if(send(i, &tmp, sizeof(tmp), 0) < 0){
390                                     //perror("send error");
391                                 }
392                             } else if (result == REGISTER_DUPLICATE){
393                                 int tmp = 1;
394                                 if(send(i, &tmp, sizeof(tmp), 0) < 0){
395                                     //perror("send error");
396                                 }
397                                 if(send(i, &tmp, sizeof(tmp), 0) < 0){
398                                     //perror("send error");
399                                 }
400                             }
401                         } else if (buf[0] == 'c'){
402                             char procedure_name[128];
403                             std::vector<uint32_t> argtypes_vector;
404                             // receive the procedure name and argtypes the same
405                             // way as before.
406                             recv(i, procedure_name, sizeof procedure_name, 0);
407                             int tmp[1];
408                             do{
409                                 recv(i, tmp, sizeof tmp, 0);
410                                 argtypes_vector.push_back(tmp[0]);
411                             }while(tmp[0] != 0);
412                             // Turn the argtypes vector into an array.
413                             int argtypes[argtypes_vector.size()];
414                             std::copy(argtypes_vector.begin(), argtypes_vector.end(), argtypes);
415                             registration returnedRegistration = getRegistration(procedure_name, argtypes);
416                             if (returnedRegistration.robin_value == -1){
417                                 // The -1 flag on robin_value indicates no suitable
418                                 // server was found for the given signature.
419                                 int tmp = LOC_FAILURE;
420                                 if (send(i, &tmp, sizeof(int), 0) == -1){
421                                     //perror("send error");
422                                 }
423                                 // Send a reasonCode of 0.
424                                 tmp = 1;
425                                 if (send(i, &tmp, sizeof(int), 0) == -1){
426                                     //perror("send error");
427                                 }
428                             } else {
429                                 // a server was found successfully.
430                                 int tmp = LOC_SUCCESS;
431                                 if (send(i, &tmp, sizeof(int), 0) == -1){
432                                     //perror("send error");
433                                 }
434                                 char * tmp_char = returnedRegistration.server_identifier;
435                                 if (send(i, tmp_char, sizeof(char) * 128, 0) == -1){
436                                     //perror("send error");
437                                 }
438                                 tmp = returnedRegistration.server_port;
439                                 if (send(i, &tmp, sizeof(int), 0) == -1){
440                                     //perror("send error");
441                                 }
442                             }
443                         } else if (buf[0] == TERMINATE){
444                             // Send TERMINATE to each of our servers and exit
445                             for(i = 0; i <= fdmax; i++) {
446                                 char terminate = 't';
447                                 if (send(i, &terminate, sizeof(terminate), 0) == -1){
448                                     //perror("send error");
449                                 }
```

```
                        }

                    }

                    /*
                    // Loop until the null terminator. If the character before
                    // the current character is a space, capitalize this
                    // character if it's between a and z.
                    for (stringIndex = 0; buf[stringIndex] != '\0'; stringIndex++){
                        if (stringIndex == 0 || buf[stringIndex-1] == ' '){
                            if (buf[stringIndex] >= 'a' && buf[stringIndex] <= 'z'){
                                buf[stringIndex] = buf[stringIndex] - 32;
                            }
                        }
                    }

                    // The string is now in title case, send it back!
                    if (FD_ISSET(i, &master)) {
                        if(send(i, buf, nbytes, 0) == -1){
                            //perror("send");
                        }
                    }
                    */
                }
            } // 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;
}
```