Data Networks Project 3

Austin Warnock

**SOURCE CODE:**

**Server.c**

1. // Server side implementation of UDP client-server model
2. #include <stdio.h>
3. #include <stdlib.h>
4. #include <unistd.h>
5. #include <string.h>
6. #include <sys/types.h>
7. #include <sys/socket.h>
8. #include <arpa/inet.h>
9. #include <netinet/in.h>
10. #include <pthread.h>
12. #define MAX 1024
13. typedef struct Group
14. {
15. char name[MAX];
16. int numOfClients;
17. int connectedClients;
18. int numOfMessages;
19. } group;
21. typedef struct threadArgs
22. {
23. struct Group group;
24. int sockfd;
25. struct sockaddr\_in cliaddr;
27. } args;
29. struct threadArgs args1;
31. void \*sendMessages()
32. {
33. int sockfd = args1.sockfd;
34. struct sockaddr\_in cliaddr = args1.cliaddr;
35. struct Group group = args1.group;
37. int len, numSent = 0;
38. len = sizeof(cliaddr); //len is value/resuslt
40. while (numSent < group.numOfMessages)
41. {
42. char buff[MAX];
43. int n;
44. bzero(buff, sizeof(buff));
45. printf("\nWhat do you want to broadcast? ");
46. n = 0;
47. while ((buff[n++] = getchar()) != '\n')
48. ;
49. n = 0;
50. n = sendto(sockfd, (const char \*)buff, strlen(buff),
51. MSG\_CONFIRM, (const struct sockaddr \*)&cliaddr,
52. len);
53. numSent++;
54. }
55. }
56. void \*handleClientMessage(void \*context)
57. {
58. struct threadArgs \*args = context;
59. pthread\_t readHandler;
60. int sockfd = args->sockfd;
61. struct sockaddr\_in cliaddr = args->cliaddr;
62. struct Group group = args->group;
63. for (;;)
64. {
65. printf("DATA: %s, %i, %i\n", group.name, group.numOfClients, group.connectedClients);
67. char buff[MAX];
68. int len, n;
69. len = sizeof(cliaddr); //len is value/resuslt
70. n = recvfrom(sockfd, (char \*)buff, MAX,
71. MSG\_WAITALL, (struct sockaddr \*)&cliaddr,
72. &len);
73. buff[n] = '\0';
74. if (strncmp(buff, "JOIN", sizeof("JOIN")) == 0)
75. {
76. printf("Attempting to join\n");
77. }
78. else if (strncmp(buff, "QUIT", sizeof("QUIT")) == 0)
79. {
80. printf("Attempting to Quit\n");
81. group.connectedClients--;
82. close(sockfd);
83. }
84. else if (strncmp(buff, group.name, strlen(group.name)) == 0)
85. {
86. printf("Attempting to join group\n");
87. if (group.connectedClients < group.numOfClients)
88. {
89. group.connectedClients++;
90. char \*buff = "Successfully connected";
91. sendto(sockfd, (const char \*)buff, strlen(buff),
92. MSG\_CONFIRM, (const struct sockaddr \*)&cliaddr,
93. len);
94. }
95. else
96. {
97. //SEND ERROR TO CLIENT TOO MANY CONNECTIONS
98. char \*buff = "Cannot connect too many clients";
99. sendto(sockfd, (const char \*)buff, strlen(buff),
100. MSG\_CONFIRM, (const struct sockaddr \*)&cliaddr,
101. len);
102. }
103. }
104. else
105. {
106. printf("GROUP DOES NOT EXIST\n");
107. }
109. //UPDATE ANY CHANGED VALUES
110. args1.sockfd = sockfd;
111. args1.cliaddr = cliaddr;
112. args1.group = group;
113. pthread\_create(&readHandler, NULL, sendMessages, NULL);
114. }
116. }
117. struct Group serverStartUp()
118. {
119. struct Group group;
120. char buff[MAX];
121. int n;
122. bzero(buff, sizeof(buff));
123. printf("\nWhat is name of your group? ");
124. n = 0;
125. while ((buff[n++] = getchar()) != '\n')
126. ;
127. memcpy(group.name, buff, sizeof(buff));
128. group.name[strlen(group.name) - 1] = '\0';
130. // ZERO OUT BUFF AND ASK FOR NUM CLIENTS
131. bzero(buff, sizeof(buff));
132. printf("\nWhat is the max number of clients? ");
133. n = 0;
134. while ((buff[n++] = getchar()) != '\n')
135. ;
136. group.numOfClients = atoi(buff);
138. // ZERO OUT BUFF AND ASK FOR NUM OF MESSAGES TO SEND
139. bzero(buff, sizeof(buff));
140. printf("\nWhat is the number of messages to send? ");
141. n = 0;
142. while ((buff[n++] = getchar()) != '\n')
143. ;
144. group.numOfMessages = atoi(buff);
145. group.connectedClients = 0;
146. return group;
147. }
149. // Driver code
150. int main(int argc, char \*\*argv)
151. {
152. int sockfd;
153. struct sockaddr\_in servaddr, cliaddr;
154. struct Group group1;
155. pthread\_t connectionHandler;
157. // Creating socket file descriptor
158. if ((sockfd = socket(AF\_INET, SOCK\_DGRAM, 0)) < 0)
159. {
160. perror("socket creation failed");
161. exit(EXIT\_FAILURE);
162. }
164. memset(&servaddr, 0, sizeof(servaddr));
165. memset(&cliaddr, 0, sizeof(cliaddr));
167. // Filling server information
168. servaddr.sin\_family = AF\_INET; // IPv4
169. servaddr.sin\_addr.s\_addr = INADDR\_ANY;
170. servaddr.sin\_port = htons(atoi(argv[1]));
172. // Bind the socket with the server address
173. if (bind(sockfd, (const struct sockaddr \*)&servaddr,
174. sizeof(servaddr)) < 0)
175. {
176. perror("bind failed");
177. exit(EXIT\_FAILURE);
178. }
180. //SETUP SERVER INFO HERE
182. group1 = serverStartUp();
184. //POPULATE ARGUMENT STRUCT
185. args1.sockfd = sockfd;
186. args1.cliaddr = cliaddr;
187. args1.group = group1;
189. //SPIN UP A THREAD TO HANDLE JOINS AND QUITS IN THE BACKGROUND
190. pthread\_create(&connectionHandler, NULL, handleClientMessage, (void \*)&args1);
191. printf("DATA IN (end): %i, %i\n", args1.sockfd, args1.cliaddr.sin\_family);
193. pthread\_join(connectionHandler, NULL);
195. return 0;
196. }

Client.c

1. // Client side implementation of UDP client-server model
2. #include <stdio.h>
3. #include <stdlib.h>
4. #include <unistd.h>
5. #include <string.h>
6. #include <sys/types.h>
7. #include <sys/socket.h>
8. #include <arpa/inet.h>
9. #include <netinet/in.h>
10. #include <pthread.h>
12. #define MAX 1024
14. typedef struct threadArgs
15. {
16. int sockfd;
17. struct sockaddr\_in servaddr;
19. } args;
20. void readServer(int sockfd, struct sockaddr\_in servaddr)
21. {
22. for (;;)
23. {
24. int n, len;
25. char buffer[MAX];
26. n = recvfrom(sockfd, (char \*)buffer, MAX,
27. MSG\_WAITALL, (struct sockaddr \*)&servaddr,
28. &len);
29. buffer[n] = '\0';
31. int random = rand() % 10; //only accepts 90% of messages
32. if (random != 0)
33. printf("Server : %s\n", buffer);
34. }
35. }
36. void \*handleSubscription(void \*context)
37. {
38. struct threadArgs \*args = context;
40. int sockfd = args->sockfd;
41. struct sockaddr\_in servaddr = args->servaddr;
42. char buff[MAX];
43. printf("\nWhat do you want to do [JOIN/QUIT]? ");
44. do
45. {
46. int n;
47. bzero(buff, sizeof(buff));
48. n = 0;
49. while ((buff[n++] = getchar()) != '\n')
50. ;
52. sendto(sockfd, (const char \*)buff, strlen(buff) - 1,
53. MSG\_CONFIRM, (const struct sockaddr \*)&servaddr,
54. sizeof(servaddr));
55. printf("%s Request sent.\n", buff);
57. } while (strncmp(buff, "QUIT", strlen("QUIT")) != 0);
58. }
59. void setupClient(char \*groupName)
60. {
61. char buff[MAX];
62. int n;
63. bzero(buff, sizeof(buff));
64. printf("\nWhat group do you want to join? ");
65. n = 0;
66. while ((buff[n++] = getchar()) != '\n')
67. ;
68. strcpy(groupName, buff);
69. groupName[strlen(groupName) - 1] = '\0';
70. }
72. // Driver code
73. int main(int argc, char \*\*argv)
74. {
75. int sockfd;
76. char buffer[MAX];
77. struct sockaddr\_in servaddr;
78. struct threadArgs args1;
79. pthread\_t connectionHandler;
81. // Creating socket file descriptor
82. if ((sockfd = socket(AF\_INET, SOCK\_DGRAM, 0)) < 0)
83. {
84. perror("socket creation failed");
85. exit(EXIT\_FAILURE);
86. }
88. memset(&servaddr, 0, sizeof(servaddr));
90. // Filling server information
91. servaddr.sin\_family = AF\_INET;
92. servaddr.sin\_port = htons(atoi(argv[2]));
93. servaddr.sin\_addr.s\_addr = inet\_addr(argv[1]);
95. //Setup client server relation here
96. char groupName[MAX];
97. setupClient(groupName);
99. int n, len;
101. sendto(sockfd, (const char \*)groupName, strlen(groupName),
102. MSG\_CONFIRM, (const struct sockaddr \*)&servaddr,
103. sizeof(servaddr));
104. printf("Group name sent.\n");
106. n = recvfrom(sockfd, (char \*)buffer, MAX,
107. MSG\_WAITALL, (struct sockaddr \*)&servaddr,
108. &len);
109. buffer[n] = '\0';
110. printf("Server : %s\n", buffer);
112. //POPULATE ARGUMENT STRUCT
113. args1.sockfd = sockfd;
114. args1.servaddr = servaddr;
115. //SPIN UP A THREAD TO HANDLE JOINS AND QUITS IN THE BACKGROUND
116. pthread\_create(&connectionHandler, NULL, handleSubscription, (void \*)&args1);
118. readServer(sockfd, servaddr);
120. close(sockfd);
121. return 0;
122. }

 Makefile:

1. all: server client
3. server: Server.c
4. gcc -pthread -o server Server.c -Wall -pedantic
6. client: Client.c
7. gcc -pthread -o client Client.c -Wall -pedantic
9. clean:
10. rm -f \*.o client server

**Approach:**

I first started by taking the source code given to us in Project 0 and modified it to accept a UDP socket. I then shifted my focus to creating a suitable test environment to develop and test my code. This mainly involved making a Makefile to generate all the binary executables in one quick command. I also read through and took some of the sample code from G4G to help get me started: <https://www.geeksforgeeks.org/udp-server-client-implementation-c/> and <https://www.geeksforgeeks.org/multithreading-c-2/> . Once I got the main pieces of my code working, I abstracted them out into several functions to make the code a lot more readable.

Functions in Client.c:

* readServer() – takes in a file descriptor, and server socket in order to form a loop that reads 90% of the messages sent out by the server.
* handleSubscription – is a thread handler that takes in a threadArg struct in order to listen to user keyboard commands in the background
* setupClient – is a helper function in order to setup the desired group name

Functions in Server.c

* sendMessages is a thread handler that sends out the messages from the server
* handleClientMessages is also a thread handler that handles the user data sent from the various clients. It has the Boolean logic for updating the group struct.
* serverSetup is a helper function that helps define the behavior of the server. i.e name, number of connections, number of message, and number of connected clients

Data Structures:

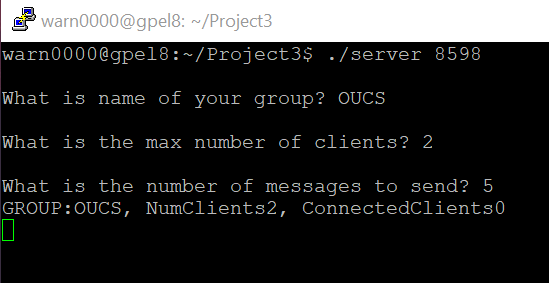
1. typedef struct threadArgs
2. {
3. struct Group group;
4. int sockfd;
5. struct sockaddr\_in cliaddr;
7. } args;
8. typedef struct Group
9. {
10. char name[MAX];
11. int numOfClients;
12. int connectedClients;
13. int numOfMessages;
14. } group;

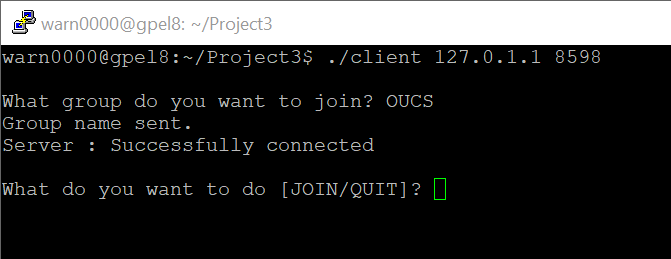
The threadArgs data structure is a way to for the driver code to send over multiple variables to the pthread. The group data structure is used to keep track of a given group and its users without the need to keep track of hundreds of variables all throughout the code base.

**Assumptions:**

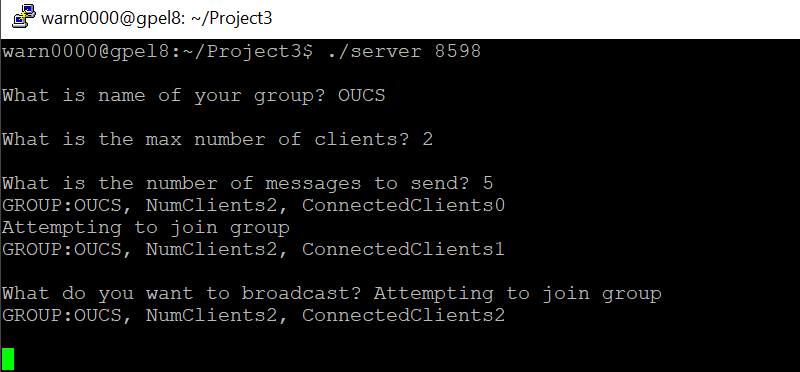
This program assumes that channel names are case and whitespace sensitive and treats them as such. i.e OUCS =/= oucs. This is assumed because case and whitespace sensitivity would make sense in a secure channel-based application.

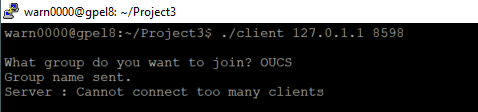
**Screenshots:**

Server startup: 

Client Startup: 

Server After 2 clients connect:



Client when there are too many connected

Sending Test:

