Assignment 3: Server + Client Communication in C

Jake Gendreau

November 20, 2024

Contents

1 Documentation

1.1 Client

In this assignment, the client is responsible for prompting the user to enter an expression, then shipping the expression off to the server. The basic operation will be as follows:

1. Open socket and connect to server
2. Prompt user and get input
3. Check the input to ensure that it is the correct format (prefix notation)
4. Send it off to the server
5. Get a response from the server
6. Print the result
7. Close the socket

1.2 Server

In this assignment, the server is responsible for getting input from the client, parsing it to get a solution, then returning that solution to the client. The basic operation will be as follows:

1. Open socket and bind it to a port
2. Listen for connections
3. Once a connection has been established, receive an expression from the client
4. Parse the expression and store the result in a string
5. Return the result to the client
6. Close the socket

1.3 netInfo.h

This header file contains a shared function, as well as some declarations that will be useful in the client and server code. These are

- Defining MAX_STRING_SIZE
- Defining SOCK_PORT
- Establishing some networking types with typedef
- Create socket function

I did this so that these values will be consistent across client and server, and are defined in a single place.

1.4 makefile

Going into the assignment, I wasn't a huge fan of makefiles. After using one, I found that they are quite handy. Mine is simple but it does the trick. It does the following:

- Define the C Compiler as `gcc`.
- Define CLIENT and SERVER as `client` and `server`. This isn't particularly useful for this assignment, but with larger codebases and more complicated names, the abstraction could be useful.
- Default mode: all - Compiles changed scripts to .o files, then to ELF files.
- Clean mode: Deletes all ELF and .o files in the directory

Having all of these options available through just a few keystrokes was very helpful, and I will likely use makefiles again in the future.

2 Assignment Log

2.1 Hours

Prior to starting this assignment, I expect it will take me about 3 hours to complete.

- November 17, 3:30 PM - 4:00 PM: Completed basic client and server communication.
- November 17, 4:00 PM - 6:00 PM: Completed input validation on client side, and expression parsing on server side.
- November 18, 10:50 AM - 11:20 AM: Completed assignment
- October 18, 12:00 PM - 1:30 PM: Worked on program writeup

The actual networking part of the assignment was easier than I anticipated. There are so many resources readily available that finding one that I could understand wasn't a challenge. What took a lot of time was input validation and parsing. I decided that I wanted the program to support large numbers, not just single digit numbers. Implementing that took a little bit of time, but it wasn't too bad.

2.2 Notes

- I learned that makefiles are actually really helpful and speed up the development process by quite a lot.
- I also learned that networking in C isn't really that bad, at least with the assignment that we had. for future networking assignments, I'll probably stick to Python or JS, but knowing how to do it in C is quite handy.
- Geeks4Geeks resources about client and server communication, I found, were very helpful. Everything was clearly explained, and I was able to modify the example code that they gave to work with this assignment.

3 Output

Listing 1: combinedOut.txt

```
1  ====== SERVER ======
2  jake@pop-os:~/Documents/Schoolwork/CS270/Assignments/Assign3$ ./server
3  SERVER: Open and waiting for connection...
4  Connected to 4
5  CLIENT: + 9 8
6  SERVER: Result is 17
7  Closing server...
8
9  ====== CLIENT ======
10 jake@pop-os:~/Documents/Schoolwork/CS270/Assignments/Assign3$ ./client
11 CLIENT: Creating Socket...
12 CLIENT: Connecting to server...
13 CLIENT: Connected to server.
14 Enter a valid expression (<operator> <operand> <operand>)
15 + 9 8
16 SERVER: The result is 17
17 Closing socket...
```

4 Source Files

Listing 2: netInfo.h

```
/*
* netInfo.h
* A header file containing useful values across multiple files
* Jake Gendreau
* November 17, 2024
*/

#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>

const int SOCK_PORT = 4501;
const int MAX_STRING_SIZE = 255;

typedef struct sockaddr_in sockaddr_in;
typedef struct sockaddr sockaddr;

// Create the socket with IPv4, Stream type, and IP protocol
int CreateSocket()
{
    int retVal = socket(AF_INET, SOCK_STREAM, 0);

    if(retVal == -1)
    {
        printf("Failed to make socket!\n");
        exit(-1);
    }

    return retVal;
}
```

Listing 3: server.c

```
/*
* server.c
* A program that will act as a server to receive an
* RPN expression and return the result
*
* Jake Gendreau
* November 18, 2024
*/

#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>

#include "netInfo.h"

int ReceiveData(int sockD, char* strData, int strSize);
int GetOperands(int* op1, int* op2, char* expression);
int ParseExpression(char* expression);
int isNum(char curChar);

int main(int argc, char const* argv[])
{
    // Create socket
    int servSockD = CreateSocket();

    sockaddr_in servAddr;

    servAddr.sin_family = AF_INET;
    servAddr.sin_port = htons(SOCK_PORT);
    servAddr.sin_addr.s_addr = INADDR_ANY;

    // Bind the socket to the port
    bind(servSockD, (sockaddr*)&servAddr, sizeof(
       servAddr));

```

```
39      printf("SERVER: Open and waiting for connection...\n
           ");
40
41      // Listen for connections
42      listen(servSockD, 1);
43
44      // Hold client socket
45      int clientSocket = accept(servSockD, NULL, NULL);
46
47      printf("Connected to %i\n", clientSocket);
48
49      // Receive data from client
50      char strData[MAX_STRING_SIZE];
51      if(ReceiveData(clientSocket, strData,
           MAX_STRING_SIZE) == 0)
52      {
53          printf("Failed to receive data\n");
54          exit(-1);
55      }
56
57      int result = ParseExpression(strData);
58      char retStr[MAX_STRING_SIZE];
59
60      printf("CLIENT: %s", strData);
61      printf("SERVER: Result is %i\n", result);
62
63      snprintf(retStr, MAX_STRING_SIZE, "SERVER: The
           result is %i", result);
64
65      send(clientSocket, retStr, MAX_STRING_SIZE, 0);
66
67      printf("Closing socket...\n");
68      close(clientSocket);
69      close(servSockD);
70  }
71
72  int ParseExpression(char* expression)
73  {
74      // Get operator
75      char operator = expression[0];
```

```
76
77      int op1 = 0;
78      int op2 = 0;
79
80      // Get operands
81      GetOperands(&op1, &op2, expression);
82
83      switch(operator)
84      {
85          case '+':
86              return op1 + op2;
87              break;
88
89          case '-':
90              return op1 - op2;
91              break;
92
93          case '*':
94              return op1 * op2;
95              break;
96
97          case '/':
98              return op1 / op2;
99              break;
100
101         case '%':
102             return op1 % op2;
103             break;
104
105         default:
106             printf("Unexprected operator!\n");
107             return 0;
108     }
109 }
110
111 int GetOperands(int* op1, int* op2, char* expression)
112 {
113     int spaceIndex = 0;
114     char op1Str[MAX_STRING_SIZE];
115     char op2Str[MAX_STRING_SIZE];
```

```
116
117     // Read up to first space
118     for(int i = 2; i < MAX_STRING_SIZE; i++)
119     {
120         spaceIndex = i;
121
122         if(expression[i] == ' ')
123         {
124             break;
125         }
126
127         if(isNum(expression[i]))
128         {
129             op1Str[i - 2] = expression[i];
130         }
131
132         else
133         {
134             return 0;
135         }
136     }
137
138     // 3rd char = ' ' or first operand going to max
        strings size -> invalid input
139     if(spaceIndex == 2 || spaceIndex == MAX_STRING_SIZE)
140     {
141         return 0;
142     }
143
144     // Assign operand1
145     *op1 = atoi(op1Str);
146
147     // Read up to first space
148     for(int i = spaceIndex + 1; i < MAX_STRING_SIZE &&
        expression[i] != '\0'; i++)
149     {
150         if(expression[i] == ' ' || expression[i] == '\0'
            || expression[i] == '\n')
151         {
152             break;
```

```
153         }
154
155         if(isNum(expression[i]))
156         {
157             op2Str[i - (spaceIndex + 1)] = expression[i
                ];
158         }
159
160         else
161         {
162             return 0;
163         }
164     }
165
166     *op2 = atoi(op2Str);
167 }
168
169 int ReceiveData(int sockD, char* strData, int strSize)
170 {
171     int retVal = recv(sockD, strData, strSize - 1, 0);
172
173     if (retVal <= 0) {
174         return 0;
175     }
176
177     // Append null terminator
178     strData[retVal] = '\0';
179     return 1;
180 }
181
182 int isNum(char curChar)
183 {
184     char* validChars = "0123456789";
185
186     for(int i = 0; validChars[i] != '\0'; i++)
187     {
188         if(curChar == validChars[i])
189         {
190             return 1;
191         }
```

```
192         }
193
194         return 0;
195 }
```

Listing 4: client.c

```
/*
* client.c
* A program that will get an RPN expression from the
* user, send it off to the server, and print the result
* from the server.
*
* Jake Gendreau
* November 18, 2024
*/
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <string.h>

#include "netInfo.h"

/*=====NETWORKING FUNCTIONS=====*/
int ConnectToServer(int sockD);
int ReceiveData(int sockD, char* strData, int strSize);

/*=====GET INPUT=====*/
char* GetUserExpression();
int GetOperands(int* op1, int* op2, char* expression);

/*=====STRING PROCESSING=====*/
int isOperator(char curChar);
int isNum(char curChar);


int main(int argc, char* argv)
{
    printf("CLIENT: Creating Socket...\n");

    int sockD = CreateSocket();

    printf("CLIENT: Connecting to server...\n");
```

```
40
41      ConnectToServer(sockD);
42
43      printf("CLIENT: Connected to server.\n");
44
45      char* expression = GetUserExpression();
46
47      // Attempt to send
48      if(send(sockD, expression, MAX_STRING_SIZE, 0) ==
           -1)
49      {
50          perror("Failed to send");
51      }
52
53      // Get data from server
54      char* strData = (char*)malloc(sizeof(char ) *
           MAX_STRING_SIZE);
55      int receiveCondition = ReceiveData(sockD, strData,
           MAX_STRING_SIZE);
56
57      if(receiveCondition == 0)
58      {
59          printf("Failed to get data from server\n");
60          exit(-1);
61      }
62
63      // Print data
64      printf("%s\n", strData);
65
66      printf("Closing socket...\n");
67      // Close socket
68      close(sockD);
69  }
70
71  int ConnectToServer(int sockD)
72  {
73      struct sockaddr_in servAddr;
74
75      // Initialize values
76      servAddr.sin_family = AF_INET;
```

```
77      servAddr.sin_port = htons(SOCK_PORT);
78      servAddr.sin_addr.s_addr = INADDR_ANY;
79
80      // Connect to server
81      int connectStatus = connect(sockD, (struct sockaddr
           *)&servAddr, sizeof(servAddr));
82
83      if (connectStatus == -1) {
84          perror("Error connecting to server");
85          close(sockD);
86          exit(-1);
87      }
88
89      return 0;  // Success
90  }
91
92  int ReceiveData(int sockD, char* strData, int strSize)
93  {
94      int retVal = recv(sockD, strData, strSize - 1, 0);
95
96      if (retVal <= 0) {
97          return 0;
98      }
99
100     // Append null terminator
101     strData[retVal] = '\0';
102     return 1;
103 }
104
105 char* GetUserExpression()
106 {
107     char* expression = (char*)malloc(sizeof(char) *
           MAX_STRING_SIZE);
108     int validInput = 0;
109
110     if(strncmp(expression, "quit", MAX_STRING_SIZE) ==
           0)
111     {
112         return expression;
113     }
```

```
114
115     // Loop to get valid input
116     while(!validInput)
117     {
118         printf("Enter a valid expression (<operator> <
                operand> <operand>)\n");
119
120         // Read in expression
121         fgets(expression, MAX_STRING_SIZE, stdin);
122
123         // Ensure first char is operator
124         if(!isOperator(expression[0]))
125         {
126             printf("Invalid expression! ");
127             validInput = 0;
128             continue;
129         }
130
131         else
132         {
133             validInput = 1;
134         }
135
136         // Ensure operand is followed by a space
137         if(expression[1] != ' ')
138         {
139             printf("Invalid expression! ");
140             validInput = 0;
141             continue;
142         }
143
144         // Get the two operands
145         int op1;
146         int op2;
147
148         // Failure to get operands
149         if(GetOperands(&op1, &op2, expression) == 0)
150         {
151             printf("Invalid expression! ");
152             validInput = 0;
```

```
153             continue;
154         }
155 
156         // Valid string
157         validInput = 1;
158     }
159 
160     return expression;
161 }
162 
163 int GetOperands(int* op1, int* op2, char* expression)
164 {
165     int spaceIndex = 0;
166     char op1Str[MAX_STRING_SIZE];
167     char op2Str[MAX_STRING_SIZE];
168 
169     // Read up to first space
170     for(int i = 2; i < MAX_STRING_SIZE; i++)
171     {
172         spaceIndex = i;
173 
174         if(expression[i] == ' ')
175         {
176             break;
177         }
178 
179         if(isNum(expression[i]))
180         {
181             op1Str[i - 2] = expression[i];
182         }
183 
184         else
185         {
186             return 0;
187         }
188     }
189 
190     // 3rd char = ' ' or first operand going to max
        strings size -> invalid input
191     if(spaceIndex == 2 || spaceIndex == MAX_STRING_SIZE)
```

```
192     {
193         return 0;
194     }
195 
196     // Assign operand1
197     *op1 = atoi(op1Str);
198 
199     // Read up to first space
200     for(int i = spaceIndex + 1; i < MAX_STRING_SIZE &&
           expression[i] != '\0'; i++)
201     {
202         if(expression[i] == ' ' || expression[i] == '\0'
               || expression[i] == '\n')
203         {
204             break;
205         }
206 
207         if(isNum(expression[i]))
208         {
209             op2Str[i - (spaceIndex + 1)] = expression[i
                   ];
210         }
211 
212         else
213         {
214             return 0;
215         }
216     }
217 
218     *op2 = atoi(op2Str);
219 }
220 
221 int isNum(char curChar)
222 {
223     char* validChars = "0123456789";
224 
225     for(int i = 0; validChars[i] != '\0'; i++)
226     {
227         if(curChar == validChars[i])
228         {
```

```
229             return 1;
230         }
231     }
232 
233     return 0;
234 }
235 
236 int isOperator(char curChar)
237 {
238     char* validChars = "+-/%%*";
239 
240     for(int i = 0; validChars[i] != '\0'; i++)
241     {
242         if(curChar == validChars[i])
243         {
244             return 1;
245         }
246     }
247 
248     return 0;
249 }
```

5 makefile

Listing 5: makefile

```
# Makefile for client and server assignment, CS270.
# Jake Gendreau
# November 17, 2024

# Compiler and flags
CC = gcc

# Targets
CLIENT = client
SERVER = server

# Default mode
all: $(CLIENT) $(SERVER)

# Compile to .o files
$(CLIENT): client.o
  $(CC) $(CFLAGS) -o $(CLIENT) client.o

$(SERVER): server.o
  $(CC) $(CFLAGS) -o $(SERVER) server.o

# Compile to .c files
client.o: client.c
  $(CC) $(CFLAGS) -c client.c

server.o: server.c
  $(CC) $(CFLAGS) -c server.c

# Remove generated files
clean:
  rm -f *.o $(CLIENT) $(SERVER)
```