28 DE FEBRERO DE 2023

LABORATORY 1: C

OPERATING SYSTEMS

FRANCISCO SECCHI - ALEX HERNANDEZ
TECNOCAMPUS MARESME-MATARÓ

Index

Introduction

Repository: https://github.com/FranSecchi/Lab1-C

In this report we will be explaining the first C laboratory.

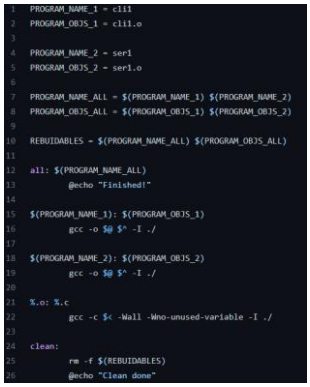
This laboratory consists in 3 activities, exemplifying a TCP server and client interaction with a simple guessing game. In the latest activity, we will also have an UDP server, where the TCP server will act as an UDP server.

Compiling c-files

Note that to compile our code each activity has a Makefile file, which will compile every c-program on the activity folder. We do it so by using the *make* command on our prompt, as:

"make <file-to-compile> -f act#.Makefile".

Every Makefile will follow a similar structure as:



```
PROGRAM_NAME_1 = cli1
PROGRAM_OBJS_1 = cli1.o

PROGRAM_NAME_2 = ser1
PROGRAM_OBJS_2 = ser1.o

PROGRAM_NAME_ALL = $(PROGRAM_NAME_1) $(PROGRAM_NAME_2)
PROGRAM_OBJS_ALL = $(PROGRAM_OBJS_1) $(PROGRAM_OBJS_2)

REBUIDABLES = $(PROGRAM_NAME_ALL) $(PROGRAM_OBJS_ALL)

all: $(PROGRAM_NAME_ALL)
	@echo "Finished!"

$(PROGRAM_NAME_1): $(PROGRAM_OBJS_1)
	gcc -o $@ $^ -I ./

$(PROGRAM_NAME_2): $(PROGRAM_OBJS_2)
	gcc -o $@ $^ -I ./

%.o: %.c
	gcc -c $< -Wall -Wno-unused-variable -I ./

clean:
	rm -f $(REBUIDABLES)
	@echo "Clean done"
```

We can also compile every file and get our executables by manually doing so, inputing in our prompt in the project directory:

"gcc -o <exe-name> <c-file>".

```
devasc@fsecchi:~/labs/devnet-src/c/lab1/activity_1$ gcc -o cli1 cli1.c
cli1.c: In function 'main':
cli1.c:72:12: warning: format not a string literal and no format arguments [-Wformat-security]
   72 |     printf(buffer);
      |            ^~~~~~
devasc@fsecchi:~/labs/devnet-src/c/lab1/activity_1$
```

Libraries

Every c-file will have the following libraries included, necessary for any socket implementation and basic coding:

```
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
```

For random numbers using seed (srand()), we will also need *<time.h>*.

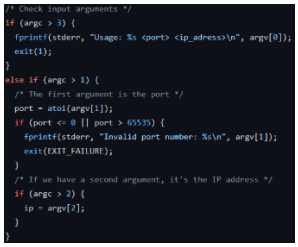
TCP Client

Since we won't change how the client behaves but instead the server internal functionality, n all 3 activities we will be working with the same client implementation.

- **Input check of the arguments:**

We need to check the input arguments to make sure we have 1 or more arguments, and the port number is a correct valid number.

We have both, port and IP variables, initialized to a default port and IP address (8888 and 127.0.0.1).



```
/* Check input arguments */
if (argc > 3) {
 fprintf(stderr, "Usage: %s <port> <ip_adress>\n", argv[0]);
 exit(1);
}
else if (argc > 1) {
 /* The first argument is the port */
 port = atoi(argv[1]);
 if (port <= 0 || port > 65535) {
  fprintf(stderr, "Invalid port number: %s\n", argv[1]);
  exit(EXIT_FAILURE);
 }
 /* If we have a second argument, it's the IP address */
 if (argc > 2) {
  ip = argv[2];
 }
}
```

- **TCP socket:**

To create a socket, we'll use a function of the utility we imported before, named <sys./socket.h>. The method has 3 parameters:

– DOMAIN: we are working on an Internet environment, "PF_INET".
– COMMUNICATION: connection or non-connection oriented. (Sock_Stream for connection oriented).
– TRANSPORT PROTOCOL: TCP or UDP, IPPROTO_TCP since we are working with a tcp-server.

```
/* Try to create TCP socket */
sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
if (sock < 0) {
  err_sys("Error socket");
}

/* Set information for sockaddr_in */
memset(&echoserver, 0, sizeof(echoserver));       /* Reset memory */
echoserver.sin_family = AF_INET;                  /* Internet/IP */
echoserver.sin_addr.s_addr = inet_addr(ip);  /* Server address */
echoserver.sin_port = htons(port);       /* Server port */

/* Try to have a connection with the server */
if (connect(sock, (struct sockaddr *) &echoserver, sizeof(echoserver)) < 0) {
  err_sys("Error connect");
}
```

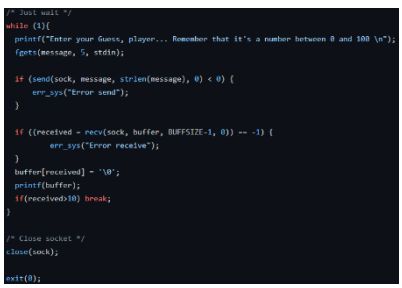
Memset resets all memory value to 0. Then we configure the family directions and the server IP address and port we want to connect to.

Finally, we establish a connection request with the server, using *connect* method, which has 3 parameters. The socket, the server IP address, and its size.

- **Client-server interaction**

We start on an infinite loop until the connection breaks. To interact with the TCP server we use both functions called *send()* and *recv()*, specifying the socket, the variable we want to store the message and its length (to send or receive) and the flag.

Once the connection is over or the game has ended, we close the socket.



```
/* Just wait */
while (1){
 printf("Enter your Guess, player... Remember that it's a number between 0 and 100 \n");
 fgets(message, 5, stdin);

 if (send(sock, message, strlen(message), 0) < 0) {
    err_sys("Error send");
 }

 if ((received = recv(sock, buffer, BUFFSIZE-1, 0)) == -1) {
        err_sys("Error receive");
 }
 buffer[received] = '\0';
 printf(buffer);
 if(received>10) break;
}

/* Close socket */
close(sock);

exit(0);
```

Activity 1

TCP Server

- **Input check of the arguments:**

This time we need to check the input arguments to make sure we have 1 or less arguments, and the port number is a correct valid number. We have only a default port: 8888.

```c
/* Check input arguments */
if (argc > 2) {
  fprintf(stderr, "Usage: %s <port>\n", argv[0]);
  exit(1);
}
if (argc == 2){
  if(!check_port(argv[1])){
      printf("Invalid port number: %s\n", argv[1]);
      exit(1);
  }
  port = atoi(argv[1]);
}
```

- **TCP socket:**

```c
/* Create TCP socket */
serversock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
if (serversock < 0) {
  err_sys("Error socket");
}

/* Set information for sockaddr_in structure */
memset(&echoserver, 0, sizeof(echoserver));       /* Reset memory */
echoserver.sin_family = AF_INET;                  /* Internet/IP */
echoserver.sin_addr.s_addr = htonl(INADDR_ANY);   /* Any address */
echoserver.sin_port = htons(port);       /* Server port */

/* Bind socket */
if (bind(serversock, (struct sockaddr *) &echoserver, sizeof(echoserver)) < 0) {
  err_sys("Error bind");
}

/* Listen socket */
if (listen(serversock, MAXPENDING) < 0) {
  err_sys("Error listen");
}
```

We create the socket the same way as for the client since it's only a TCP socket information structure. This time we set *INADDR_ANY* though, meaning we accept any available address.

Next, we *bind* the socket, indicating the server socket descriptor with the configurated socket structure and its size.

Finally, the *listen* method allows us to limit several pending connection requests.

- **Client-server interaction**

```c
/* As a server we are in an infinite loop, waiting forever */
while (1) {
  unsigned int clientlen = sizeof(echoclient);

  /* Wait for a connection from a client */
  clientsock = accept(serversock, (struct sockaddr *) &echoclient, &clientlen);
  if (clientsock < 0) {
    err_sys("Error accept");
  }
  fprintf(stdout, "Client: %s\n", inet_ntoa(echoclient.sin_addr));

  /* Call function to handle socket */
  handle_client(clientsock);
}
```

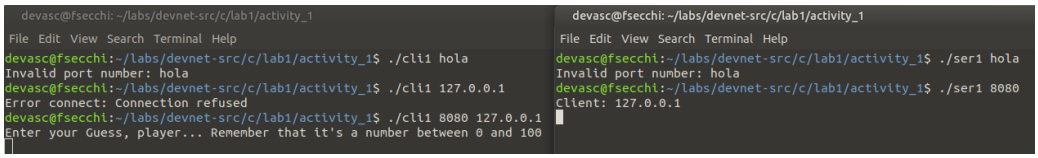
We begin an infinite loop since the server will be waiting forever to petitions. With the function *accept* we stablish a connection, then we are ready to handle the client:

```c
void handle_client(int sock) {
  char buffer[BUFFSIZE];
  int received = -1;
  int guess, num_guesses = 0;
  int answer = rand()%100;
  /* Just wait */
  while (1){
    // Receive guess from client
        int len = recv(sock, buffer, BUFFSIZE, 0);
        if (len < 0) {
            perror("recv");
            break;
        } else if (len == 0) {
            printf("Connection closed by client\n");
            break;
        }

        buffer[len] = '\0';
        guess = atoi(buffer);
        num_guesses++;

        // Compare guess to answer and send result to client
        if (guess < answer) {
            if (send(sock, "More bro\n", 9, 0) < 0) {
                perror("send");
                break;
            }
        } else if (guess > answer) {
            if (send(sock, "Less dude\n", 10, 0) < 0) {
                perror("send");
                break;
            }
        } else {
            if (send(sock, "There you go\n", 13, 0) < 0) {
                perror("send");
                break;
            }
            printf("Client guessed %d in %d tries\n", guess, num_guesses);
            break;
        }
    }
```

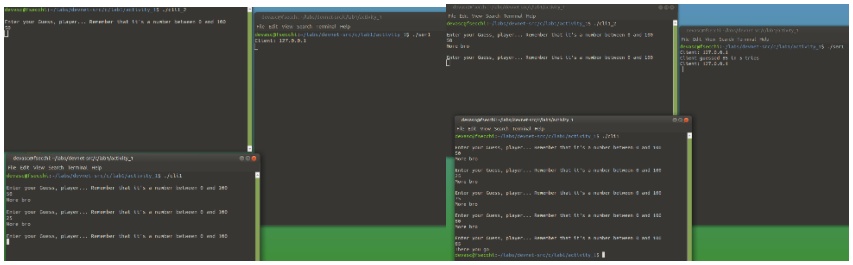
Using the functions *send()* and *recv()* we send and receive messages the same way as the client. Here the server generates a random number between 0-100 and responds to the client's guesses.

Testing

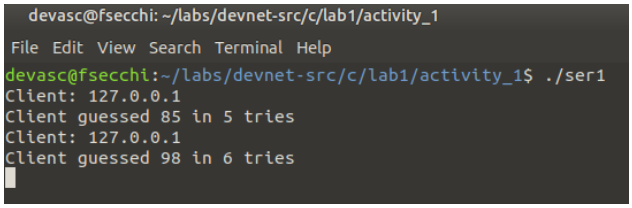
Input arguments check:



Testing with 2 clients, we can see that cli1_2 waits for the first to end:



Result for both clients from server side:



Activity 2

TCP Server

Socket creation, binding and listening works as the previous one.

- **Input check of the arguments:**

```c
/* Check input arguments */
if (argc > 3 || argc <= 1) {
    fprintf(stderr, "Usage: %s <file_name> <port>\n", argv[0]);
    exit(1);
}
else if(argc > 2){
    if(!check_port(argv[2])){
        printf("Invalid port number: %s\n", argv[2]);
        exit(1);
    }
    port = atoi(argv[2]);
    file_name = argv[1];
}
```

This time we need to check the input arguments to make sure we have 1 or more arguments and less than 3. We are taking 2 arguments, the file name from which we are going to get the random number and the port, last being optional since we have a default port 8888.

- **File access:**

```c
/*Save file info*/
FILE *fp = fopen(argv[1], "r");
if (fp == NULL) {
    err_sys("Error opening file: " + file_name);
}
int letters[MAX_LINES];
char line[MAX_LINE_LENGTH];

while(fgets(line, MAX_LINE_LENGTH, fp) != NULL && num_lines < MAX_LINES){
    letters[num_lines] = strlen(line) - 1;
    num_lines++;
}
fclose(fp);
```

Working with files in C must first open the file and save it on a FILE variable and once it is done close it (using the methods *fopen()* and *fcose()*). To access the file content we do it by using *fgets()* which parses it line per line.

Once we've saved the number of characters per line in an array, the game proceeds just the same as activity 1. We get the random number from this array instead:

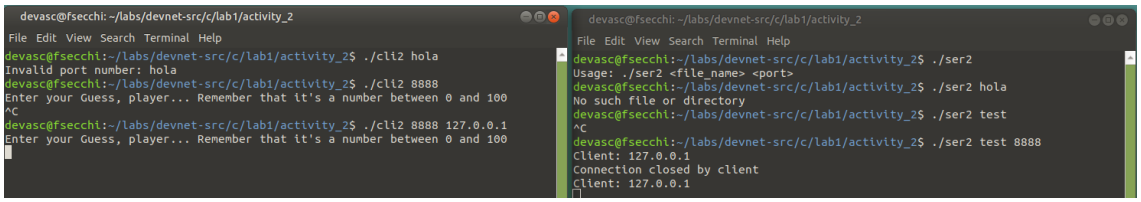
```c
void handle_client(int sock, int letters[], int num_lines) {
  char buffer[BUFFSIZE];
  int received = -1;
  int guess, num_guesses = 0;
  int answer = letters[rand()%num_lines]%100;
```

The method rand() can work with a seed, previously set:

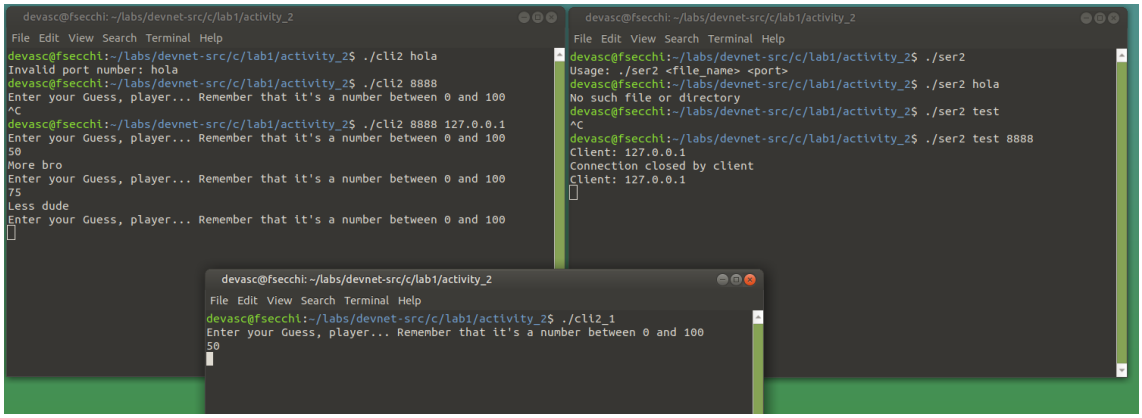
```c
/*Seed*/
srand(time(NULL));
```

Testing

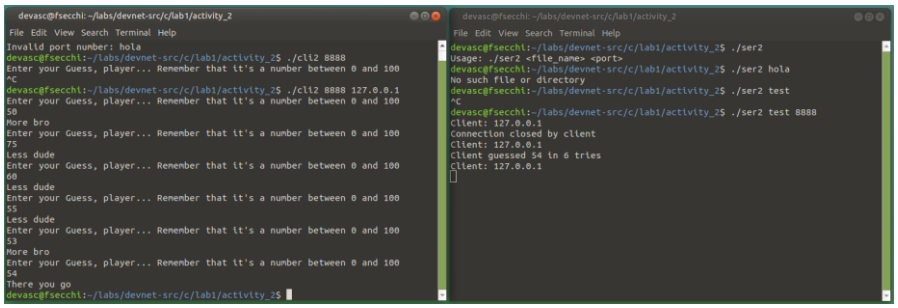
Input arguments check:



Testing with 2 clients, we can see that cli2_1 waits for the first to end:



Result for both clients from server side:



As we can see the game works just the same, but the internal functionality is different since the server gets the answer reading a file's content.

Activity 3

UDP Server

We have the same input arguments as the last TCP server, since this UDP server will be reading the file this time. Of course, the File access works the same as well.

```c
/* Check input arguments */
if (argc > 3 || argc <= 1) {
    fprintf(stderr, "Usage: %s <file_name> <port>\n", argv[0]);
    exit(1);
}
else if(argc > 2){
    if(!check_port(argv[2])){
        printf("Invalid port number: %s\n", argv[2]);
        exit(1);
    }
    port = atoi(argv[2]);
}
```

```c
/*Save file info*/
FILE *fp = fopen(argv[1], "r");
if (fp == NULL) {
    err_sys("Error opening file: " + *argv[1]);
}
int letters[MAX_LINES];
char line[MAX_LINE_LENGTH];

while(fgets(line, MAX_LINE_LENGTH, fp) != NULL && num_lines < MAX_LINES){
    letters[num_lines] = (strlen(line) - 1)%100;
    num_lines++;
}
fclose(fp);
```

- **UDP socket:**

```c
/* Create UDP socket */
sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);
if (sock < 0) {
    err_sys("Error socket");
}

/* Prepare sockaddr_in structure for server address */
memset(&echoserver, 0, sizeof(echoserver));       /* Erase the memory area */
echoserver.sin_family = AF_INET;                  /* Internet/IP */
echoserver.sin_addr.s_addr = htonl(INADDR_ANY); /* Receive requests from any IP address valid on server */
echoserver.sin_port = htons(port);       /* Server port */

/* Get size of echoserver structure */
serverlen = sizeof(echoserver);

/* Bind that socket with the OS, to be able to receive messages on that socket */
if (bind(sock, (struct sockaddr *) &echoserver, serverlen) < 0) {
    err_sys("Error bind");
}
```

As we can see, an UDP socket only differs from a TCP socket by the third parameter in its creation, where we indicate *IPPROTO_UDP* instead. We also don't have a listen function since UDP is non-connection oriented and we don't need to set a limit for clients.

- **Client-server interaction**

```c
/* As a server we are in an infinite loop, waiting forever */
while (1) {
    /* Set the maximum size for address */
    clientlen = sizeof(echoclient);
    echolen = sizeof(int);
    /* Receive a message from a particular client */
    if (first_request) {
        received = recvfrom(sock, buffer, BUFFSIZE, 0, (struct sockaddr *) &echoclient, &clientlen);
        if (received < 0) {
            err_sys("Error receiveing word from client");
        }
        buffer[received] = '\0';

        /* Print client address */
        fprintf(stderr, "Client: %s, Message: %s\n", inet_ntoa(echoclient.sin_addr), buffer);

        sendto(sock, &num_lines, echolen, 0, (struct sockaddr *) &echoclient, clientlen);
        first_request = 0;
    } else {
        received = recvfrom(sock, buffer, sizeof(int), 0, (struct sockaddr *) &echoclient, &clientlen);
        if (received != sizeof(int)) {
            err_sys("Error client didnt send an integer");
        }
        memcpy(&message, buffer, sizeof(int));
        /* Print client number */
        fprintf(stderr, "Client: %s, Message: %d\n", inet_ntoa(echoclient.sin_addr), message);
        sendto(sock, &letters[message], echolen, 0, (struct sockaddr *) &echoclient, clientlen);
        first_request = 1;
    }
}
```

We still have to remain on an infinite loop since this is a server, and we must wait for messages from clients.

Now we want to identify if a client's request is its first one, since the first message we will receive it's just his IP. We do so with *recvfrom()* method, which basically works the same as *recv()*, we save and print the IP and reply to our UDP client with the number of lines of the file. We use *sendto()*, which works also as *send()*, to send an integer.

Then if it's the second message, we will want to send the number of characters for the line number the client requests. As we are receiving an integer, we must use *memcpy()* to translate it and save it.

UDP Client – TCP Server

- **Input arguments check:**

```
/* Check input arguments */
if (argc != 4 ) {
    fprintf(stderr, "Usage: %s <port_udp> <ip> <port_tcp>\n", argv[0]);
    exit(1);
}
```

This time we are forcing to input all 3 arguments, the UDP port, the IP and the TCP port.

- **UDP socket:**

```
/* Create UDP socket */
udp_sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);
if (udp_sock < 0) {
    err_sys("Error on socket creation");
}
/* Configure/set socket address for the server */
memset(&udp_server, 0, sizeof(udp_server));       /* Erase the memory area */
udp_server.sin_family = AF_INET;                  /* Internet/IP */
udp_server.sin_addr.s_addr = inet_addr(argv[2]);  /* IP address */
udp_server.sin_port = htons(atoi(argv[1]));       /* Server port */
```

As a client, all we need to do is create an UDP socket and set the properties.

Then we can send our IP so we get our first request and save the number of lines that our UDP server will provide:

```
/* Get number of lines*/
echolen = strlen(argv[2]);
if (sendto(udp_sock, argv[2], echolen, 0, (struct sockaddr *) &udp_server, sizeof(udp_server)) != echolen) {
    err_sys("error writing word on socket");
}
clientlen = sizeof(udp_client);
if (recvfrom(udp_sock, &num_lines, sizeof(int), 0, (struct sockaddr *) &udp_client, &clientlen) != sizeof(int)) {
    err_sys("Error reading");
}
```

- **TCP socket:**

We follow by generating a TCP socket to act as a server, who will handle clients to play the guessing game:

```
/* Create TCP socket */
serversock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
if (serversock < 0) {
    err_sys("Error socket");
}

/* Set information for sockaddr_in structure */
memset(&tcp_server, 0, sizeof(tcp_server));       /* Reset memory */
tcp_server.sin_family = AF_INET;                  /* Internet/IP */
tcp_server.sin_addr.s_addr = htonl(INADDR_ANY);   /* Any address */
tcp_server.sin_port = htons(atoi(argv[3]));       /* Server port */

/* Bind socket */
if (bind(serversock, (struct sockaddr *) &tcp_server, sizeof(tcp_server)) < 0) {
    err_sys("Error bind");
}

/* Listen socket */
if (listen(serversock, MAXPENDING) < 0) {
    err_sys("Error listen");
}
```

- **Client-server interaction**

```c
/* As a server we are in an infinite loop, waiting forever */
while (1) {
    clientlen = sizeof(tcp_client);

    /* Wait for a connection from a client */
    clientsock = accept(serversock, (struct sockaddr *) &tcp_client, &clientlen);
    if (clientsock < 0) {
    err_sys("Error accept");
    }
    fprintf(stdout, "Client: %s\n", inet_ntoa(tcp_client.sin_addr));

    /* Get number of lines*/

    int num = rand()%num_lines;
    fprintf(stdout, "Random line: %d\n", num);
    echolen = sizeof(int);
    if (sendto(udp_sock, &num, sizeof(int), 0, (struct sockaddr *) &udp_server, sizeof(udp_server)) != echolen) {
        err_sys("error writing word on socket");
    }
    clientlen = sizeof(udp_client);
    if (recvfrom(udp_sock, &num, sizeof(int), 0, (struct sockaddr *) &udp_client, &clientlen) != sizeof(int)) {
        err_sys("Error reading");
    }
    fprintf(stdout, "Answer: %d\n", num);
    /* Call function to handle socket */
    handle_client(clientsock, num);
}
```

Here we act first as an UDP client, requesting the number of characters for a random line number, using both methods: *sendto()* and *recvfrom()*.

Once we have our final answer, we can start our guessing game, which work just the same as before:

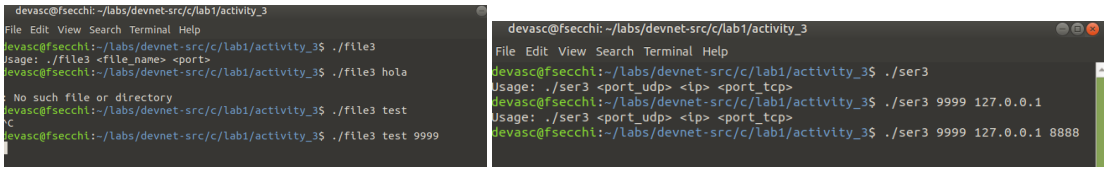
```c
void handle_client(int sock, int answer) {
  char buffer[BUFFSIZE];
  int received = -1;
  int guess, num_guesses = 0;

  /* Just wait */
  while (1){
    // Receive guess from client
    int len = recv(sock, buffer, BUFFSIZE, 0);
    if (len < 0) {
        perror("recv");
        break;
    } else if (len == 0) {
        printf("Connection closed by client\n");
        break;
    }
    buffer[len] = '\0';
    guess = atoi(buffer);
    num_guesses++;
    // Compare guess to answer and send result to client
    if (guess < answer) {
        if (send(sock, "More bro\n", 9, 0) < 0) {
            perror("send");
            break;
        }
    } else if (guess > answer) {
        if (send(sock, "Less dude\n", 10, 0) < 0) {
            perror("send");
            break;
        }
    } else {
        if (send(sock, "There you go\n", 13, 0) < 0) {
            perror("send");
            break;
        }
        printf("Client guessed %d in %d tries\n", guess, num_guesses);
        break;
    }
  }
```

We always want to close our socket once the client is done, using the correct method:

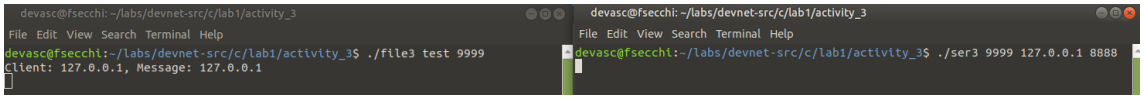
```c
/* Close socket */
close(sock);
```

Testing

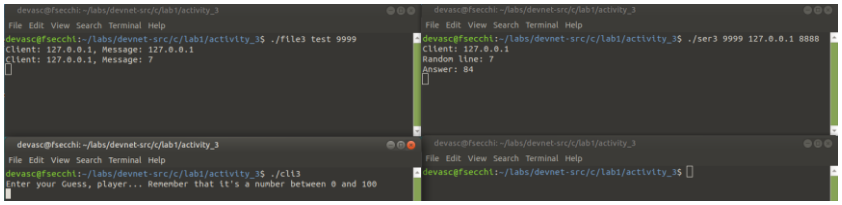
Input arguments check:



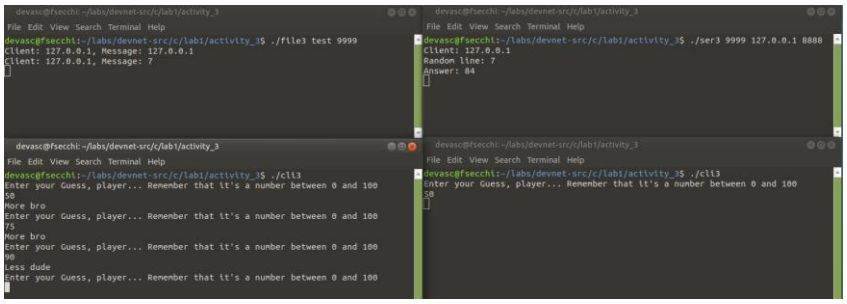
The UDP client makes the first request and the UDP server prints its IP, then the client waits for TCP connections:



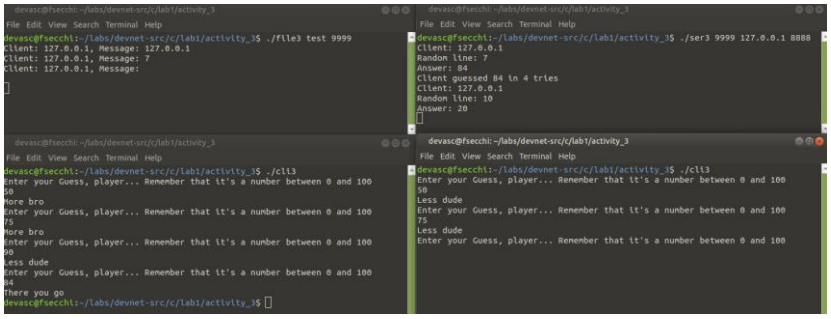
Here we can see an execution example when a client stablishes connection with the TCP server, which makes its final request to the UDP server, getting the game's final answer:



If a second client tries to connect with the TCP server, it'll have to wait for the first to finish.



Once the first clients finish the game, automatically starts the second:



Result with both clients: