

1. OBJETIVO

Entender los conceptos básicos de los sockets tanto de manera local como a través de una red.

2. INTRODUCCIÓN

Un socket es una comunicación bidireccional que permite comunicación entre proceso alojados en la misma computadora o en otras. Para crear un socket se deben especificar tres parámetros: comunicación, espacio de nombres y protocolo.

La comunicación se encarga de controlar cómo es transmitida la información y especifica el número asociados para la comunicación. La información mandada es dividida en paquetes, la comunicación se encarga del manejo de estos paquetes y cómo debe ser enviada del emisor al receptor. La transmisión de los paquetes se puede mandar por:

- ✓ Conexión: Garantiza que se van a entregar todos los paquetes en el orden en el que fueron enviados. Si se perdieron paquetes o se re-ordenaron, el receptor solicita su retransmisión al emisor.
- ✓ Datagrama: No garantiza que los paquetes lleguen y mucho menos que lleguen en el mismo orden en el que fueron mandados.

El espacio de nombres (*namespace*) especifica la dirección del socket. La dirección del socket a dónde se va a establecer la conexión.

El protocolo sirve para especificar cómo deben de ser transmitidos los datos; por ejemplo el protocolo TCP/IP.

API de sockets de Berkeley

Debido a la variedad de protocolos, se popularizó el API de sockets de Berkeley como una forma de englobar la variedad de protocolos de red y proporcionar una interface de programación. Con los sockets de Berkeley se pueden crear sockets para trabajar en una sola máquina (POSIX local IPC) y comunicación con otras máquinas (TCP/IP).

La más fundamental estructura para la creación de sockets incluida en el API es `sockaddr`. Su sintaxis es la siguiente:

```
struct sockaddr {  
    unsigned short int sa_family;  
    char sa_data[14];  
};
```

`sa_family` describe el tipo de dirección almacenada y `sa_data` contiene la dirección actual. Los integrantes de `sa_family` se describe a continuación:

Address family	Protocol family	Descripción
AF_UNIX	PF_UNIX	Sockets de UNIX
AF_INET	PF_INET	TCP/IP (versión 4)
AF_AX25	PF_AX25	Protocolo amateur de radio
AF_IPX	PF_IPX	Protocolo Novell IPX
AF_APPLETALK	PF_APPLETALK	Protocolo Apple Talk DDS

Para el uso de sockets en C, se necesita incluir la cabecera `<sys/socket.h>`. Debido a que Linux ve todo como archivos, las funciones de lectura y escritura son usadas también en sockets.

Para crear un socket se usa la función `socket` cuya sintaxis se muestra a continuación.

```
int socket(int dominio, int tipo, int protocolo)
```

`dominio` es usado para especificar el protocolo de red a usar (PF_UNIX, PF_INET, etc.). `tipo` establece la categoría del protocolo ya sea streaming (SOCK_STREAM) o datagrama (SOCK_DGRAM). `protocolo` indica el protocolo por default a usar basado en el `dominio` y el `tipo` antes mencionados.

Para que un socket tenga uso, se necesita un servidor que se encarga de escuchar peticiones de un cliente. El cliente puede pedir información del servidor, enviar datos o acceder a alguno de los servicios proporcionados por él.

El servidor se encarga de crear el socket y crear una asociación entre el socket y una dirección que puede ser un archivo —localmente— o una dirección de internet. Después de la asociación, el servidor espera una conexión. Una vez que se establece la conexión, la acepta y abre la conexión para el intercambio de información.

Cada una de las operaciones antes mencionadas tiene su función representativa:

```
int bind(int sockfd, struct sockaddr *addr, int addrlen);
```

```
int listen(int sockfd, int backlog);
```

```
int accept(int sockfd, struct sockaddr *addr, int *addrlen);
```

```
int connect(int sockfd, struct sockaddr *addr, int addrlen);
```

sockfd es el descriptor de archivo regresado por la llamada previa de *socket()*.

sockaddr es un apuntador a la estructura de la dirección del socket.

addrlen indica el tamaño de *sockaddr*.

backlog define el máximo número de conexiones pendientes que son permitidas.

Las funciones *bind()*, *listen()* y *connect()* regresan el valor de cero si se ejecutan exitosamente y -1 si fallan.

3. DESARROLLO

>> Ejemplo 1 – Sockets locales

Los sockets que conectan procesos en la misma computadora pueden usar los espacios de nombres representados por *PF_LOCAL* y *PF_UNIX*. El nombre del socket es especificado en una estructura llamada *sckaddr_un* definida en `<sys/un.h>` cuya sintaxis se muestra a continuación:

```
struct sockaddr_un
{
    sa_family_t sun_family;    /* AF_UNIX */
    char        sun_path[108]; /* pathname */
};
```

Sólo los procesos que se están ejecutando en la misma computadora se pueden comunicar por sockets con espacios de nombres locales. Debido a que los sockets locales residen en el sistema de archivos, un socket local es listado como un archivo. Para remover a un socket se utiliza la llamada a *unlink()*.

El siguiente código muestra un servidor que está escuchando peticiones de los clientes. Cuando llega un mensaje por parte del cliente, el servidor lo imprime en pantalla y espera por más mensajes.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

int server (int client_socket)
{
    while (1) {
        int length;
        char* text;

        /* First, read the length of the text message from the socket. If
        read returns zero, the client closed the connection. */
        if (read (client_socket, &length, sizeof (length)) == 0)
            return 0;

        /* Allocate a buffer to hold the text. */
        text = (char*) malloc (length);

        /* Read the text itself, and print it. */
        read (client_socket, text, length);
        printf ("%s\n", text);
```

```

        /* If the client sent the message "quit," we're all done. */
        if (!strcmp (text, "quit"))
            return 1;
        else{
            /* Free the buffer. */
            free (text);
        }
    }
}

int main (int argc, char* const argv[])
{
    int socket_fd;          /*file descriptor for socket*/
    struct sockaddr_un name; /*server socket structure*/
    int client_sent_quit_message;
    const char* const socket_name = argv[1];

    /* Create the socket. */
    socket_fd = socket (PF_LOCAL, SOCK_STREAM, 0);

    /* Indicate that this is a server. */
    name.sun_family = AF_LOCAL;
    strcpy (name.sun_path, socket_name);
    bind (socket_fd, (struct sockaddr *)&name, SUN_LEN (&name));

    /* Listen for connections. */
    listen (socket_fd, 5);

    /* Repeatedly accept connections, spinning off one server() to deal
    with each client. Continue until a client sends a "quit" message. */
    do {
        struct sockaddr_un client_name;
        socklen_t client_name_len;
        int client_socket_fd;

        /* Accept a connection. */
        client_socket_fd = accept (socket_fd, (struct sockaddr *)&client_name, &client_name_len);

        /* Handle the connection. */
        client_sent_quit_message = server (client_socket_fd);

        /* Close our end of the connection. */
        close (client_socket_fd);
    } while (!client_sent_quit_message);

    /* Remove the socket file. */
    close (socket_fd);
    unlink (socket_name);
    return 0;
}

```

Código 1 Servidor, imprime los mensajes del cliente (socket-server.c)

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

/* Write TEXT to the socket given by file descriptor SOCKET_FD. */
void write_text (int socket_fd, const char* text)
{
    /* Write the number of bytes in the string, including
    NUL-termination. */
    int length = strlen (text) + 1;
    write (socket_fd, &length, sizeof (length));

    /* Write the string. */
    write (socket_fd, text, length);
}

int main (int argc, char* const argv[])
{
    const char* const socket_name = argv[1];
    const char* const message = argv[2];
    int socket_fd;
    struct sockaddr_un name;

    /* Create the socket. */
    socket_fd = socket (PF_LOCAL, SOCK_STREAM, 0);

    /* Store the server's name in the socket address. */
    name.sun_family = AF_LOCAL;
    strcpy (name.sun_path, socket_name);

    /* Connect the socket. */
    connect (socket_fd, (struct sockaddr *)&name, SUN_LEN (&name));

    /* Write the text on the command line to the socket. */
    write_text (socket_fd, message);
    close (socket_fd);
    return 0;
}
```

[Código 2 Cliente, manda un mensaje al servidor \(socket-cliente.c\)](#)

Antes de que el cliente mande el mensaje de texto, manda el tamaño del mismo para crear un buffer con el tamaño apropiado para poder almacenar el texto antes de leerlo del socket.

La siguiente imagen muestra la salida del servidor y los mensajes que recibió por parte del cliente a través del socket.

```
[root@vesta CLinux]# ./server /tmp/socket
hola mundo
probando conectividad
quit
```

La siguiente imagen muestra los mensajes que el cliente manda al servidor a través del socket.

```
[root@vesta CLinux]# ./client /tmp/socket "hola mundo"
[root@vesta CLinux]# ./client /tmp/socket "probando conectividad"
[root@vesta CLinux]# ./client /tmp/socket "quit"
[root@vesta CLinux]#
```

Los siguientes códigos muestran la interacción de un cliente y un servidor, el cliente le manda un mensaje al servidor, éste lo obtiene y se lo manda de nuevo al cliente.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

#define SOCK_PATH "echo_socket"

int main(void)
{
    int s, s2, t, len;
    struct sockaddr_un local, remote;
    char str[100];

    if ((s = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    local.sun_family = AF_UNIX;
    strcpy(local.sun_path, SOCK_PATH);
    unlink(local.sun_path);
    len = strlen(local.sun_path) + sizeof(local.sun_family);
    if (bind(s, (struct sockaddr *)&local, len) == -1) {
        perror("bind");
        exit(1);
    }

    if (listen(s, 5) == -1) {
        perror("listen");
```

```

        exit(1);
    }

    for(;;) {
        int done, n;
        printf("Waiting for a connection...\n");
        t = sizeof(remote);
        if ((s2 = accept(s, (struct sockaddr *)&remote, &t)) == -1) {
            perror("accept");
            exit(1);
        }

        printf("Connected.\n");

        done = 0;
        do {
            n = recv(s2, str, 100, 0);
            if (n <= 0) {
                if (n < 0) perror("recv");
                done = 1;
            }

            if (!done)
                if (send(s2, str, n, 0) < 0) {
                    perror("send");
                    done = 1;
                }
        } while (!done);

        close(s2);
    }

    return 0;
}

```

Código 3 Servidor echo (echos.c)

```

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

#define SOCK_PATH "echo_socket"

int main(void)
{
    int s, t, len;
    struct sockaddr_un remote;
    char str[100];

    if ((s = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }
}

```

```

printf("Trying to connect...\n");

remote.sun_family = AF_UNIX;
strcpy(remote.sun_path, SOCK_PATH);
len = strlen(remote.sun_path) + sizeof(remote.sun_family);
if (connect(s, (struct sockaddr *)&remote, len) == -1) {
    perror("connect");
    exit(1);
}

printf("Connected.\n");

while(printf("> "), fgets(str, 100, stdin), !feof(stdin)) {
    if (send(s, str, strlen(str), 0) == -1) {
        perror("send");
        exit(1);
    }

    if ((t=recv(s, str, 100, 0)) > 0) {
        str[t] = '\0';
        printf("echo> %s", str);
    } else {
        if (t < 0) perror("recv");
        else printf("Server closed connection\n");
        exit(1);
    }
}

close(s);

return 0;
}

```

Código 4 Cliente echo (echoc.c)

La siguiente imagen muestra que el servidor está esperando a que se conecte un cliente para devolver la cadena que éste le mande.



```

[root@vesta CLinux]# ./server
Waiting for a connection...
Connected.

```

La siguiente imagen muestra cómo el cliente manda mensaje al servidor y éste se los reenvía al cliente que los imprime en pantalla.



```

[root@vesta CLinux]# ./client
Trying to connect...
Connected.
> hola
echo> hola
> probando echo
echo> probando echo
> adios
echo> adios
>

```

>> Ejemplo 1 – Sockets TCP/IP

Los sockets de Unix pueden ser usados para comunicación entre dos procesos que se encuentran en la misma computadora. Los sockets basados Internet pueden ser usados para tener comunicación entre diferentes computadoras a través de una red y se componen de dos partes: una computadora y un puerto. Esta información es almacenada en una estructura cuya sintaxis se muestra a continuación:

```
struct sockaddr_in {
    short int sin_family;    /* AF_INET */
    uint16_t sin_port;      /* Port number */
    struct in_addr sin_addr; /* IP address */
};
```

sin_port es el puerto al que se va a conectar
sin_addr es la dirección IP.

Los parámetros antes descritos deben de estar en representación binaria. Dado que los humanos pueden aprender más fácilmente una palabra que una serie de números, se hace uso del Domain Name Service (DNS), pero las máquinas entienden números en representación binaria, por lo que tenemos que transformar el nombre en su representación numérica. La función *gethostbyname()* transforma el nombre en una dirección IP. Una vez que se tiene la IP, ésta se tiene que transformar a su representación binaria debido a que *sin_addr* lo necesita de esa manera; además de la IP, también el puerto se debe de cambiar a su representación binaria. Para facilitar este proceso se tienen las siguientes funciones:

```
char *inet_ntoa(struct in_addr addr);
char *inet_aton(const char *ddaddr, struct in_addr *ipaddr);
```

inet_ntoa para convertir de binario a decimal.

inet_aton para convertir de decimal a binario.

El siguiente código crea un socket a un sitio de Internet para extraer su información por medio del comando GET del protocolo HTTP.

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

/* Print the contents of the home page for the server's socket.
```

Return an indication of success. */

```
void get_home_page (int socket_fd)
{
    char buffer[10000];
    ssize_t number_characters_read;

    /* Send the HTTP GET command for the home page. */
    sprintf (buffer, "GET /\n");
    write (socket_fd, buffer, strlen (buffer));

    /* Read from the socket. The call to read may not
    return all the data at one time, so keep
    trying until we run out. */
    while (1) {
        number_characters_read = read (socket_fd, buffer, 10000);
        if (number_characters_read == 0)
            return;

        /* Write the data to standard output. */
        fwrite (buffer, sizeof (char), number_characters_read, stdout);
    }
}

int main (int argc, char* const argv[]){

    int socket_fd;
    struct sockaddr_in name;
    struct hostent* hostinfo;

    /* Create the socket. */
    socket_fd = socket (PF_INET, SOCK_STREAM, 0);

    /* Store the server's name in the socket address. */
    name.sin_family = AF_INET;

    /* Convert from strings to numbers. */
    hostinfo = gethostbyname (argv[1]);

    if (hostinfo == NULL)
        return 1;
    else
        name.sin_addr = *((struct in_addr *) hostinfo->h_addr);

    /* Web servers use port 80. */
    name.sin_port = htons (80);

    /* Connect to the Web server */
    if (connect (socket_fd, (struct sockaddr *)&name, sizeof (struct sockaddr_in)) == -1) {
        perror ("connect");
        return 1;
    }

    /* Retrieve the server's home page. */
    get_home_page (socket_fd);
```

```
return 0;
```

```
}
```

Código 5 Extrayendo información de una página por el método GET (mkget.c)

La función `gethostbyname()` regresa un apuntador *hostinfo* apunta a una estructura llamada *hostent* que se describe a continuación.

```
struct hostent
{
    char *h_name;      /* Nombre del host*/
    char **h_aliases;  /* Arreglo de nombres alternos del host*/
    int  h_addrtype;   /*AF_INET*/
    int  h_length;     /*Tamaño de la dirección en bytes*/
    char **h_addr_list; /* Arreglo de direcciones de internet*/
};
#define h_addr h_addr_list[0]
```



```
[root@vesta CLinux]# ./exe www.google.com
HTTP/1.0 302 Found
Location: http://www.google.com.mx/
Cache-Control: private
Content-Type: text/html; charset=UTF-8
Set-Cookie: PREF=ID=3c95dad4dbbe4a15:FF=0:TM=1330395119:LM=1330395119:S=QYFjq79e7msNKYuM; expires=Thu, 27-Feb-2014 02:11:59 GMT; path=/; domain=.google.com
Date: Tue, 28 Feb 2012 02:11:59 GMT
Server: gws
Content-Length: 222
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN

<HTML><HEAD><meta http-equiv="content-type" content="text/html; charset=utf-8">
<TITLE>302 Moved</TITLE></HEAD><BODY>
<H1>302 Moved</H1>
The document has moved
<A HREF="http://www.google.com.mx/">here</A>.
</BODY></HTML>
```

El siguiente código se encarga de obtener las direcciones IP asociadas a un dominio.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```

#include <arpa/inet.h>

int main(int argc, char *argv[ ])
{
    struct hostent *h;

    /* error check the command line */
    if(argc != 2)
    {
        fprintf(stderr, "Usage: %s <domain_name>\n", argv[0]);
        exit(1);
    }

    /* get the host info */
    if((h=gethostbyname(argv[1])) == NULL)
    {
        perror("gethostbyname(): ");
        exit(1);
    }
    else
        printf("gethostbyname() is OK.\n");

    printf("The host name is: %s\n", h->h_name);
    printf("The IP Address is: %s\n", inet_ntoa(*(struct in_addr *)h->h_addr));
    printf("The address length is: %d\n", h->h_length);

    printf("Sniffing other names...sniff...sniff...sniff...\n");
    int j = 0;
    do
    {
        printf("An alias #%d is: %s\n", j, h->h_aliases[j]);
        j++;
    }
    while(h->h_aliases[j] != NULL);

    printf("Sniffing other IPs...sniff....sniff...sniff...\n");
    int i = 0;
    do
    {
        printf("Address #%i is: %s\n", i, inet_ntoa(*(struct in_addr *)h->h_addr_list[i]));
        i++;
    }
    while(h->h_addr_list[i] != NULL);
    return 0;
}

```

Código 6 Extrae las direcciones Ip de un dominio (ipaddr.c)

En el código `h->h_addr` es un `char*` e `inet_ntoa()` requiere una estructura de tipo `in_addr`. Es por esta razón que se hace un cast a una estructura `in_addr` y después se hace la desreferencia para obtener los datos.

```
[root@vesta CLinux]# ./exe www.google.com
gethostbyname() is OK.
The host name is: www.l.google.com
The IP Address is: 74.125.227.144
The address length is: 4
Sniffing other names...sniff...sniff...sniff...
An alias #0 is: www.google.com
Sniffing other IPs...sniff...sniff...sniff...
Address #0 is: 74.125.227.144
Address #1 is: 74.125.227.147
Address #2 is: 74.125.227.148
Address #3 is: 74.125.227.145
Address #4 is: 74.125.227.146
```

El siguiente código muestra la interacción de un servidor con uno o varios clientes. El servidor va a estar escuchando por el puerto 3490, esperando a que algún cliente se conecte. La función *sigaction()* es responsable de eliminar a los procesos zombis que se crean por el uso de *fork()* para la creación de procesos hijos.

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

/* the port users will be connecting to */
#define MYPORT 3490
/* how many pending connections queue will hold */
#define BACKLOG 10

void sigchld_handler(int s)
{
    while(wait(NULL) > 0);
}

int main(int argc, char *argv[ ])
{
    /* listen on sock_fd, new connection on new_fd */
    int sockfd, new_fd;
    /* my address information */
    struct sockaddr_in my_addr;
    /* connector's address information */
    struct sockaddr_in their_addr;
    int sin_size;
    struct sigaction sa;
    int yes = 1;

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
```

```

{
    perror("Server-socket() error lol!");
    exit(1);
}
else
    printf("Server-socket() sockfd is OK...\n");

if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1)
{
    perror("Server-setsockopt() error lol!");
    exit(1);
}
else
    printf("Server-setsockopt is OK...\n");

/* host byte order */
my_addr.sin_family = AF_INET;
/* short, network byte order */
my_addr.sin_port = htons(MYPORT);
/* automatically fill with my IP */
my_addr.sin_addr.s_addr = INADDR_ANY;

printf("Server-Using %s and port %d...\n", inet_ntoa(my_addr.sin_addr), MYPORT);

/* zero the rest of the struct */
memset(&(my_addr.sin_zero), '\0', 8);

if(bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) == -1)
{
    perror("Server-bind() error");
    exit(1);
}
else
    printf("Server-bind() is OK...\n");

if(listen(sockfd, BACKLOG) == -1)
{
    perror("Server-listen() error");
    exit(1);
}
printf("Server-listen() is OK...Listening...\n");

/* clean all the dead processes */
sa.sa_handler = sigchld_handler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART;

if(sigaction(SIGCHLD, &sa, NULL) == -1)
{
    perror("Server-sigaction() error");
    exit(1);
}
else
    printf("Server-sigaction() is OK...\n");

/* accept() loop */

```

```

while(1)
{
    sin_size = sizeof(struct sockaddr_in);
    if((new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size)) == -1)
    {
        perror("Server-accept() error");
        continue;
    }
    else
        printf("Server-accept() is OK...\n");
    printf("Server-new socket, new_fd is OK...\n");
    printf("Server: Got connection from %s\n", inet_ntoa(their_addr.sin_addr));

    /* this is the child process */
    if(!fork())
    {
        /* child doesn't need the listener */
        close(sockfd);

        if(send(new_fd, "This is a test string from server!\n", 37, 0) == -1)
            perror("Server-send() error lol!");
        close(new_fd);
        exit(0);
    }
    else
        printf("Server-send is OK...!\n");

    /* parent doesn't need this */
    close(new_fd);
    printf("Server-new socket, new_fd closed successfully...\n");
}
return 0;
}

```

Código 7 Servidor, utilizando sockets TCP/IP (serverstream.c)

```

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

// the port client will be connecting to
#define PORT 3490
// max number of bytes we can get at once
#define MAXDATASIZE 300

int main(int argc, char *argv[])
{
    int sockfd, numbytes;
    char buf[MAXDATASIZE];
    struct hostent *he;

```

```

// connector's address information
struct sockaddr_in their_addr;

// if no command line argument supplied
if(argc != 2)
{
    fprintf(stderr, "Client-Usage: %s the_client_hostname\n", argv[0]);
    // just exit
    exit(1);
}

// get the host info
if((he=gethostbyname(argv[1])) == NULL)
{
    perror("gethostbyname()");
    exit(1);
}
else
    printf("Client-The remote host is: %s\n", argv[1]);

if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{
    perror("socket()");
    exit(1);
}
Código 8 Cliente, utilizando sockets TCP/IP (clientstream.c)
else
    printf("Client-The socket() sockfd is OK...\n");

// host byte order
their_addr.sin_family = AF_INET;
// short, network byte order
printf("Server-Using %s and port %d...\n", argv[1], PORT);
their_addr.sin_port = htons(PORT);
their_addr.sin_addr = *((struct in_addr *)he->h_addr);
// zero the rest of the struct
memset(&(their_addr.sin_zero), '\0', 8);

if(connect(sockfd, (struct sockaddr *)&their_addr, sizeof(struct sockaddr)) == -1)
{
    perror("connect()");
    exit(1);
}
else
    printf("Client-The connect() is OK...\n");


if((numbytes = recv(sockfd, buf, MAXDATASIZE-1, 0)) == -1)
{
    perror("recv()");
    exit(1);
}
else
    printf("Client-The recv() is OK...\n");

buf[numbytes] = '\0';
printf("Client-Received: %s", buf);

```

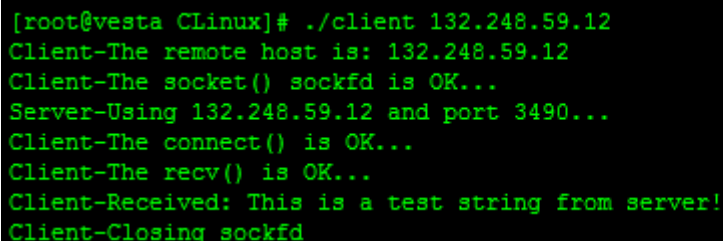
```
printf("Client-Closing sockfd\n");
close(sockfd);
return 0;
}
```

La siguiente imagen muestra la salida del servidor, cuando recibe una petición por parte del cliente manda una cadena al cliente diciendo: "This is a test string from server". Esta cadena es impresa en la salida del cliente que solicitó la conexión. Mientras no se termine el proceso del servidor, éste seguirá esperando clientes que se conecten a él.

A terminal window with a black background and green text. The text shows the execution of a server program. It starts with a prompt [root@vesta CLinux]# ./server. The output includes status messages for socket creation, binding to 0.0.0.0 on port 3490, and listening. It then shows two successful connections from 132.248.59.12 and 132.248.59.43, each followed by a successful send operation. The last line shows a successful close of the socket for the second connection.

```
[root@vesta CLinux]# ./server
Server-socket() sockfd is OK...
Server-setsockopt is OK...
Server-Using 0.0.0.0 and port 3490...
Server-bind() is OK...
Server-listen() is OK...Listening...
Server-sigaction() is OK...
Server-accept() is OK...
Server-new socket, new_fd is OK...
Server: Got connection from 132.248.59.12
Server-send is OK...!
Server-new socket, new_fd closed successfully...
Server-accept() is OK...
Server-new socket, new_fd is OK...
Server: Got connection from 132.248.59.43
Server-send is OK...!
Server-new socket, new_fd closed successfully...
```

Las siguientes imágenes muestran las salidas de dos diferentes clientes que se conectaron al servidor y que recibieron la cadena por parte de éste.

A terminal window with a black background and green text. The text shows the execution of a client program. It starts with a prompt [root@vesta CLinux]# ./client 132.248.59.12. The output includes status messages for connecting to 132.248.59.12 on port 3490, receiving the test string "This is a test string from server!", and closing the socket.

```
[root@vesta CLinux]# ./client 132.248.59.12
Client-The remote host is: 132.248.59.12
Client-The socket() sockfd is OK...
Server-Using 132.248.59.12 and port 3490...
Client-The connect() is OK...
Client-The recv() is OK...
Client-Received: This is a test string from server!
Client-Closing sockfd
```

```
[root@lestat tmp]# ./client 132.248.59.12
Client-The remote host is: 132.248.59.12
Client-The socket() sockfd is OK...
Server-Using 132.248.59.12 and port 3490...
Client-The connect() is OK...
Client-The recv() is OK...
Client-Received: This is a test string from server!
Client-Closing sockfd
```

4. CUESTIONARIO

- [1] Hacer que el servidor imprima lo que recibe del cliente.
- [2] Hacer que el servidor soporte más de un cliente.

5. BIBLIOGRAFÍA

- [1] Francisco Manuel Márquez García. “Unix: programación avanzada”. Editorial RA-MA
- [2] Richard Stevens. “Advanced Unix programming”. Editorial Addison Wesley
- [3] Kurt Wall. “Linux Programming by Example”. Editorial QUE
- [4] <http://mermaja.act.uji.es/docencia/ii22/teoria/TraspasTema2.pdf>
- [5] <http://blog.txipinet.com/2006/11/05/48-curso-de-programacion-en-c-para-gnu-linux-vii/>
- [6] <http://www.dlsi.ua.es/asignaturas/sid/JSockets.pdf>
- [7] <http://www.tenouk.com/cnlinuxsockettutorials.html>