



Universidad Nacional Autónoma de México

Facultad de Ingeniería

Proyecto Final

Arquitectura Cliente Servidor

“CLIENTE-SERVIDOR SSH”

Integrantes:

Bautista Flores Mauricio

Tenorio Veloz Alisson Dafne

Semestre: 2024-1

Fecha de Entrega: 02/12/2023

Índice

Introducción	2
Requisitos	3
Desarrollo	3
<i>Bibliotecas usadas</i>	4
<i>Comunicación Cliente-Servidor</i>	5
<i>Programa Cliente</i>	6
<i>Programa Servidor</i>	8
<i>Funciones dentro del programa</i>	9
<i>Salida final</i>	11
Conclusión	13
Referencias	12
Código	14

Introducción

En la era digital actual, los servidores cliente-servidor desempeñan un papel fundamental en la arquitectura de sistemas de información, proporcionando una base sólida para la interconexión y comunicación eficiente entre dispositivos en redes informáticas. Esta arquitectura, donde un servidor central gestiona y distribuye recursos o servicios a múltiples clientes, ha demostrado ser esencial para una amplia variedad de aplicaciones y servicios en línea.

La importancia de los servidores cliente-servidor radica en su capacidad para facilitar la colaboración, compartir información y ofrecer servicios de manera eficiente en entornos distribuidos. En este modelo, el servidor actúa como el núcleo central que almacena y administra los recursos, mientras que los clientes, que pueden ser dispositivos individuales o aplicaciones, solicitan y acceden a estos recursos según sea necesario. Este diseño fomenta una estructura escalable y modular, permitiendo la adaptación a diferentes niveles de demanda y facilitando la implementación de nuevas funcionalidades sin afectar la integridad del sistema.

En el ámbito de las aplicaciones web, los servidores cliente-servidor son esenciales para la entrega de contenido dinámico. Las páginas web modernas a menudo dependen de la comunicación constante entre el navegador del usuario (cliente) y el servidor que aloja la aplicación. Esta interacción permite la actualización en tiempo real de la información, la personalización del contenido y una experiencia de usuario más fluida.

En el ámbito empresarial, los servidores cliente-servidor son fundamentales para la gestión de bases de datos. La centralización de datos en un servidor facilita el mantenimiento, la seguridad y la realización de copias de seguridad de la información crítica de la organización. Los clientes, a través de aplicaciones específicas, pueden acceder y manipular estos datos de manera segura, garantizando la coherencia y la integridad de la información empresarial.

Además, los servidores cliente-servidor son la base de servicios esenciales en la nube. Plataformas de almacenamiento, servicios de correo electrónico, redes sociales y una variedad de aplicaciones en línea utilizan esta arquitectura para ofrecer servicios a millones de usuarios simultáneamente. La escalabilidad inherente al modelo cliente-servidor es crucial para satisfacer la creciente demanda de servicios digitales en un mundo interconectado.

En resumen, la importancia de los servidores cliente-servidor reside en su capacidad para facilitar la interconexión eficiente, la colaboración y la entrega de servicios en entornos digitales. Esta arquitectura robusta ha demostrado ser vital en la construcción y el mantenimiento de sistemas informáticos que impulsan la conectividad y la funcionalidad en diversos sectores de la sociedad moderna.

Requisitos

El presente proyecto debe cumplir con los siguientes requerimientos:

- El proyecto consiste en crear un Cliente-Servidor que ejecute comandos remotamente, como ocurre con un Cliente-Servidor SSH comercial o gratuito.
- Está desarrollado en una arquitectura cliente-servidor, con sockets TCP/IP y con conexión remota.
- Debe estar desarrollado en un ambiente Linux, en la distribución que gustes o en MacOS y programado en lenguaje C, no se aceptan proyectos programados en otros lenguajes.

Desarrollo

A continuación, se desarrollará y explicará el funcionamiento del proyecto, con el fin de mostrar su funcionamiento:

I. Bibliotecas usadas

Las librerías que se usaron dentro de los códigos, fueron para la programación de sockets y las operaciones de entrada/salida, algunas son:

1. ***stdio.h*** (*Standard Input/Output*): Contiene funciones para entrada y salida estándar, como printf y scanf.
2. ***stdlib.h*** (*Standard Library*): Proporciona funciones para la gestión de memoria dinámica, como malloc y free, además incluye funciones para la conversión de cadenas y números, como atoi.
3. ***string.h*** (*String Library*): Contiene funciones para manipulación de cadenas de caracteres, como strcpy, strlen y se utiliza para trabajar con operaciones de cadena.
4. ***unistd.h*** (*Unix Standard*): Contiene funciones estándar del sistema operativo Unix, como fork, pipe, dup, exec, entre otras. También se utiliza para realizar operaciones del sistema, como la creación de procesos y manipulación de descriptores de archivos.
5. ***sys/types.h*** (*System Types*): Define varios tipos de datos utilizados en llamadas al sistema, también contiene definiciones como pid_t (para identificadores de procesos) y size_t (para tamaños).

6. **sys/socket.h** (Socket Programming): Proporciona las definiciones de funciones y estructuras necesarias para trabajar con sockets; Incluye funciones como socket, bind, listen, accept, connect, las cuales son esenciales para la programación de red.
7. **netinet/in.h** (Internet Address Family): Define estructuras y constantes necesarias para trabajar con direcciones de red en el protocolo IP. Incluye estructuras como struct sockaddr_in para representar direcciones de socket.
8. **arpa/inet.h** (Internet Operations): Contiene funciones que realizan operaciones relacionadas con las direcciones IP e incluye funciones como inet_pton y inet_ntop para la conversión entre representaciones de direcciones IP.

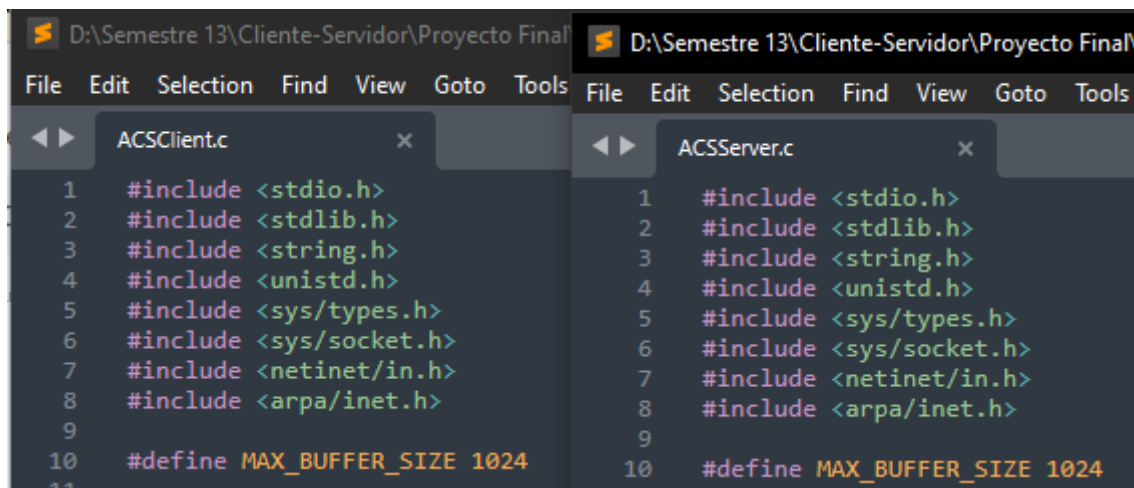


Imagen 1. Librerías que se usaron en los programas Cliente/Servidor.

II. Comunicación Cliente-Servidor

La comunicación entre el Cliente y el Servidor debe ser remota vía sockets TCP (Sockets Internet), por lo que se debe configurar en primera instancia los sockets

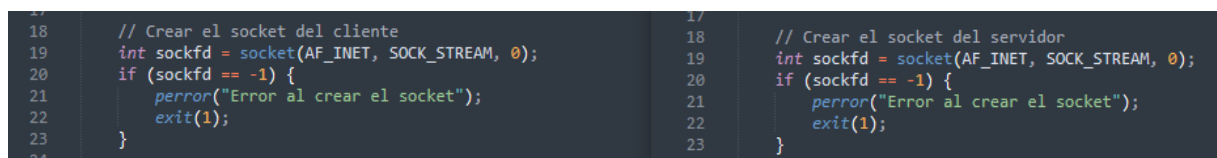


Imagen 2. Creación de los sockets en los programas Clientes/Servidor.

Y posteriormente configurarlo (Servidor) donde el socket se pone en modo escucha para escuchar las conexiones entrantes del cliente.

```
39 // Poner el socket en estado de escucha
40 if (listen(sockfd, 5) == -1) {
41     perror("Error al poner el socket en estado de escucha");
42     close(sockfd);
43     exit(1);
44 }
45
46 printf("Servidor esperando conexiones en el puerto %s...\n", argv[1]);
47
48 while (1) {
49     // Aceptar una conexión entrante
50     int clientfd = accept(sockfd, NULL, NULL);
51     if (clientfd == -1) {
52         perror("Error al aceptar la conexión");
53         close(sockfd);
54         exit(1);
55     }
56
57     struct sockaddr_in client_addr;
58     socklen_t client_addr_len = sizeof(client_addr);
59     if (getpeername(clientfd, (struct sockaddr*)&client_addr, &client_addr_len) == -1) {
60         perror("Error al obtener la dirección del cliente");
61         close(clientfd);
62         close(sockfd);
63         exit(1);
64     }
65 }
```

Imagen 3. Configuración de los sockets(Servidor).

III. Programa Cliente

El programa Servidor debe iniciarse en el host servidor (en el puerto que decidas), por lo que se implementó la verificación donde se revisa si se proporcionan los argumentos correctos en la línea de comandos (la dirección IP y el número de puerto). Si no se proporcionan, muestra un mensaje de uso y sale del programa.

```
12 int main(int argc, char *argv[]) {
13     if (argc != 3) {
14         fprintf(stderr, "Uso: %s <IP> <puerto>\n", argv[0]);
15         exit(1);
16     }
17 }
```

Imagen 4. Verificación de la dirección IP y el puerto.

El programa Cliente debe iniciarse en el host cliente (pasando el dominio o IP del servidor y el puerto desde línea de comandos).

Al crear el socket, se verifica si se proporcionan exactamente tres argumentos de línea de comandos (AF_INET, SOCK_STREAM, 0), los cuales son esenciales para

crear el servidor cliente. De no ser así, se imprime un mensaje de uso correcto y se termina el programa con un código de error.

```
17
18 // Crear el socket del cliente
19 int sockfd = socket(AF_INET, SOCK_STREAM, 0);
20 if (sockfd == -1) {
21     perror("Error al crear el socket");
22     exit(1);
23 }
24
25 // Configurar la estructura de dirección del servidor
26 struct sockaddr_in serv_addr;
27 memset(&serv_addr, 0, sizeof(serv_addr));
28 serv_addr.sin_family = AF_INET;
29 serv_addr.sin_port = htons(atoi(argv[2]));
30
31 // Convertir la dirección IP de texto a binario
32 if (inet_pton(AF_INET, argv[1], &serv_addr.sin_addr) <= 0) {
33     perror("Error al convertir la dirección IP");
34     close(sockfd);
35     exit(1);
36 }
37
38 // Conectar al servidor
39 if (connect(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) == -1) {
40     perror("Error al conectar al servidor");
41     close(sockfd);
42     exit(1);
43 }
44
45 printf("Conectado al servidor %s:%s\n", argv[1], argv[2]);
46
```

Imagen 5. Estructura del host cliente.

Se crea un socket utilizando la función `socket()`, donde también se verifica y si hay un error en la creación del socket, se imprime un mensaje de error utilizando `perror()` y se termina el programa.

Finalmente, se configura la estructura `serv_addr` que contendrá la dirección del servidor y la dirección IP se convierte de texto a binario utilizando `inet_pton()`.

Se establece la conexión con el servidor utilizando la función `connect()`, si hay un error en la conexión, se imprime un mensaje de error y se termina el programa.

```

46
47 ▼ while (1) {
48     // Leer el comando desde la entrada estándar
49     char comando[MAX_BUFFER_SIZE];
50     printf("Ingrese un comando: ");
51     fgets(comando, sizeof(comando), stdin);
52
53     // Eliminar el carácter de nueva línea de la entrada
54     size_t len = strlen(comando);
55 ▼   if (len > 0 && comando[len - 1] == '\n') {
56       comando[len - 1] = '\0';
57   }
58
59     // Enviar el comando al servidor
60 ▼   if (send(sockfd, comando, strlen(comando), 0) == -1) {
61       perror("Error al enviar el comando al servidor");
62       close(sockfd);
63       exit(1);
64   }
65
66     // Recibir la salida del servidor
67     char buffer[MAX_BUFFER_SIZE];
68     memset(buffer, 0, sizeof(buffer));
69     ssize_t bytesRecibidos = recv(sockfd, buffer, sizeof(buffer) - 1, 0);
70
71 ▼   if (bytesRecibidos == -1) {
72       perror("Error al recibir la salida del servidor");
73       close(sockfd);
74       exit(1);
75 ▼   } else if (bytesRecibidos == 0) {
76       // Servidor cerró la conexión
77       printf("Conexión cerrada por el servidor.\n");
78       break;
79   }
80
81     // Imprimir la salida del servidor
82     printf("Salida del servidor:\n%s\n", buffer);
83 }
84
85 // Cerrar el socket del cliente
86 close(sockfd);
87

```

Imagen 6. Conexión al servidor.

En un bucle infinito, el programa solicita al usuario que ingrese un comando desde la entrada estándar, lo envía al servidor utilizando la función `send()` y maneja los errores si los hay. Después de enviar el comando al servidor, el cliente espera y recibe la respuesta del servidor utilizando `recv()` y se manejan los posibles errores y se imprime la salida del servidor.

Finalmente, el programa cierra el socket del cliente antes de terminar.

```

    // Cerrar el socket del cliente
    close(sockfd);

    return 0;
}

```

Imagen 7. Cierre del socket del Cliente.

IV. Programa Servidor

Se configura la estructura `sockaddr_in` llamada `serv_addr` con la información del servidor.

```
// Configurar la estructura de dirección del servidor
struct sockaddr_in serv_addr;
memset(&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(atoi(argv[2]));

// Convertir la dirección IP de texto a binario
if (inet_pton(AF_INET, argv[1], &serv_addr.sin_addr) <= 0) {
    perror("Error al convertir la dirección IP");
    close(sockfd);
    exit(1);
}
```

Imagen 8. Configuración de la estructura del servidor.

Utilizamos `connect()` para intentar conectarse al servidor utilizando el socket del cliente (`sockfd`). Si la conexión falla, muestra un mensaje de error, cierra el socket y sale del programa.

```
// Conectar al servidor
if (connect(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) == -1) {
    perror("Error al conectar al servidor");
    close(sockfd);
    exit(1);
}
```

Imagen 9. Conexión al servidor.

Utilizamos `send()` para enviar el comando al servidor a través del socket del cliente. Si hay algún error durante el envío, muestra un mensaje de error, cierra el socket y sale del programa.

```
// Enviar el comando al servidor
if (send(sockfd, comando, strlen(comando), 0) == -1) {
    perror("Error al enviar el comando al servidor");
    close(sockfd);
    exit(1);
}
```

Imagen 10. Envío del comando al servidor.

Utilizamos `recv()` para recibir la salida del servidor en un búfer llamado `buffer`. Si hay algún error durante la recepción, muestra un mensaje de error, cierra el socket y sale del programa. Si la cantidad de bytes recibidos es 0, significa que el servidor cerró la conexión y se sale del bucle.

```
// Recibir la salida del servidor
char buffer[MAX_BUFFER_SIZE];
memset(buffer, 0, sizeof(buffer));
ssize_t bytesRecibidos = recv(sockfd, buffer, sizeof(buffer) - 1, 0);

if (bytesRecibidos == -1) {
    perror("Error al recibir la salida del servidor");
    close(sockfd);
    exit(1);
} else if (bytesRecibidos == 0) {
    // Servidor cerró la conexión
    printf("Conexión cerrada por el servidor.\n");
    break;
}
```

Imagen 11. Recepción de salida del servidor.

Imprime la salida del servidor en la consola.

```
// Imprimir la salida del servidor
printf("Salida del servidor:\n%s\n", buffer);
}
```

Imagen 12. Impresión de la salida del servidor.

V. Funciones dentro del programa

A continuación, se describen las funciones clave que realiza este programa:

main(int argc, char *argv[]):

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

Imagen 13. Función main.

- La función principal del programa que se ejecuta al inicio.
- Verifica si se proporcionan los argumentos de línea de comandos esperados (la dirección IP del servidor y el puerto).
- Crea un socket del cliente y lo conecta al servidor utilizando la dirección IP y el puerto especificados.
- Entra en un bucle donde solicita al usuario que ingrese comandos, los envía al servidor, recibe y muestra la salida del servidor, repitiendo este proceso hasta que el servidor cierra la conexión.
- Finalmente, cierra el socket del cliente y termina el programa.

Crear el Socket del Cliente con socket()

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

Imagen 14. Función sockfd.

- Utiliza la función `socket()` para crear un socket del tipo `SOCK_STREAM` (TCP) en el dominio de direcciones `AF_INET` (IPv4).
- Maneja errores en la creación del socket, imprime un mensaje de error si es necesario y termina el programa.

Configurar la Estructura de Dirección del Servidor con struct sockaddr_in

```
struct sockaddr_in serv_addr;
```

Imagen 15. Estructura sockaddr.

- Configura una estructura `struct sockaddr_in` que contiene la dirección del servidor.
- Inicializa la estructura con ceros y establece el dominio de direcciones (`AF_INET`), el puerto del servidor y la dirección IP del servidor.

Convertir la Dirección IP de Texto a Binario

```
if (inet_pton(AF_INET, argv[1], &serv_addr.sin_addr) <= 0) {
```

Imagen 16. Función `inet_pton()`.

- Utiliza la función `inet_pton()` para convertir la dirección IP del formato de texto proporcionado por el usuario a su representación binaria.
- Maneja errores en la conversión, imprime un mensaje de error si es necesario y termina el programa.

Conectar al Servidor

```
// Conectar al servidor  
if (connect(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) == -1) {
```

Imagen 17. Función `connect()`.

- Utiliza la función `connect()` para establecer una conexión con el servidor.
- Maneja errores en la conexión, imprime un mensaje de error si es necesario y termina el programa.

Bucle Principal

- Dentro de un bucle infinito, el programa solicita al usuario que ingrese comandos desde la entrada estándar.
- Elimina el carácter de nueva línea de la entrada para obtener un comando limpio.
- Utiliza `send()` para enviar el comando al servidor.
- Utiliza `recv()` para recibir la salida del servidor.
- Imprime la salida del servidor en la pantalla.

Manejo de Errores y Cierre del Programa

- En caso de errores durante la comunicación con el servidor, el programa imprime mensajes de error específicos utilizando `perror()`.
- Cierra el socket del cliente y termina el programa si se encuentran errores.

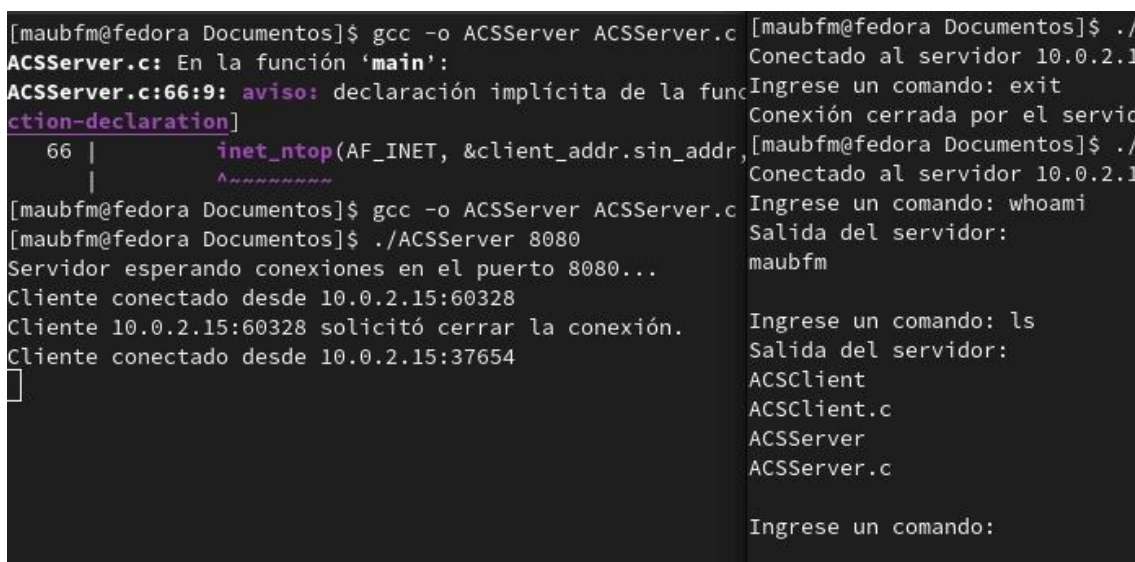
Cierre del Socket del Cliente

- Finalmente, después de salir del bucle principal, cierra el socket del cliente utilizando la función `close()`.

VI. Salida Final

Finalmente se muestra la salida de nuestro programa con el servidor de pruebas.

En la primera imagen, se puede apreciar la ejecución tanto del servidor, como del cliente.



```
[maubfm@fedora Documentos]$ gcc -o ACSServer ACSServer.c
ACSServer.c: En la función 'main':
ACSServer.c:66:9: aviso: declaración implícita de la función 'close' [-Wimplicit-function-declaration]
   66 |         inet_ntop(AF_INET, &client_addr.sin_addr,
      |         ~~~~~
[maubfm@fedora Documentos]$ gcc -o ACSServer ACSServer.c
[maubfm@fedora Documentos]$ ./ACSServer 8080
Servidor esperando conexiones en el puerto 8080...
Cliente conectado desde 10.0.2.15:60328
Cliente 10.0.2.15:60328 solicitó cerrar la conexión.
Cliente conectado desde 10.0.2.15:37654
[maubfm@fedora Documentos]$ ./ACSClient
Conectado al servidor 10.0.2.15
Ingrese un comando: exit
Conexión cerrada por el servidor
[maubfm@fedora Documentos]$ ./ACSClient
Conectado al servidor 10.0.2.15
Ingrese un comando: whoami
Salida del servidor:
maubfm
Ingrese un comando: ls
Salida del servidor:
ACSCClient
ACSCClient.c
ACSServer
ACSServer.c
Ingrese un comando:
```

Imagen 18. Ejecución y conexión del cliente con el servidor.

Una vez lograda la conexión, se observa que al ingresar cualquier comando, la salida se envía al cliente y se imprime en su pantalla.

```
[maubfm@fedora Documentos]$ gcc -o ACSServer ACSServ
[maubfm@fedora Documentos]$ ./ ACSServer
bash: ./: Es un directorio
[maubfm@fedora Documentos]$ ./ ACSServer
bash: ./: Es un directorio
[maubfm@fedora Documentos]$ ./ACSServer
Uso: ./ACSServer <puerto>
[maubfm@fedora Documentos]$ ./ACSServer 8080
Servidor esperando conexiones en el puerto 8080...
Cliente conectado desde 10.0.2.15:52752
Cliente desconectado desde 10.0.2.15:52752.
[maubfm@fedora Documentos]$
```

```
Salida del servidor:
maubfm
Ingrese un comando: ls
Salida del servidor:
ACSClient
ACSClient.c
ACSServer
ACSServer.c
Ingrese un comando: exit
cd
ls
^C
[maubfm@fedora Documentos]$
```

Imagen 19. Ejecución y conexión del cliente con el servidor.

Finalización de la conexión mediante el comando *exit*.

```
[maubfm@fedora Documentos]$ gcc -o ACSServer ACSServ
[maubfm@fedora Documentos]$ ./ACSServer 8080
Servidor esperando conexiones en el puerto 8080...
V Cliente conectado desde 10.0.2.15:60328
Cliente 10.0.2.15:60328 solicitó cerrar la conexión.
[maubfm@fedora Documentos]$
```

```
[maubfm@fedora Documentos]$ ./ACSClien
Conectado al servidor 10.0.2.15:8080
Ingrese un comando: exit
Conexión cerrada por el servidor.
[maubfm@fedora Documentos]$
```

Imagen 20. Finalización de la conexión.

Conclusiones

Bautista Flores Mauricio

El desarrollo de este proyecto de arquitectura cliente-servidor en C, con Fedora como sistema operativo, permitió adquirir una serie de conocimientos y habilidades en el desarrollo de aplicaciones de red.

En particular, la implementación de sockets TCP/IP permite comprender cómo se comunica un cliente con un servidor, lo que constituye un conocimiento fundamental en el desarrollo de este tipo de aplicaciones.

Además, la manipulación de comandos del sistema operativo mediante la ejecución remota permitió desarrollar habilidades prácticas en programación de red y sistemas operativos. Esto es importante, ya que la mayoría de las aplicaciones de red requieren la interacción con el sistema operativo.

Por último, la gestión de errores en la interacción cliente-servidor fortaleció la comprensión de la programación en C y la configuración de entornos Linux. Esto es esencial para el desarrollo de aplicaciones de red robustas y confiables.

Tenorio Veloz Alisson Dafne

La importancia de los servidores cliente-servidor radica en su capacidad para gestionar recursos centralizados, permitiendo a los clientes acceder y utilizar estos recursos de manera coordinada. En el ámbito empresarial, la gestión centralizada de bases de datos y servicios en la nube a través de esta arquitectura ha revolucionado la forma en que las organizaciones manejan la información y ofrecen servicios digitales a usuarios finales.

En el ámbito de las aplicaciones web, los servidores cliente-servidor posibilitan experiencias dinámicas y personalizadas para los usuarios. La interacción constante entre el navegador del cliente y el servidor permite la actualización en tiempo real de contenidos, la ejecución de operaciones complejas en el servidor y la adaptabilidad a las necesidades cambiantes de los usuarios.

El código proporcionado ejemplifica la implementación de un cliente que se conecta a un servidor utilizando sockets, un componente esencial de la comunicación en redes. A través de este ejemplo, se puede apreciar cómo la arquitectura cliente-servidor facilita la comunicación bidireccional, permitiendo la transmisión de datos desde el cliente al servidor y viceversa.

En conclusión, la arquitectura cliente-servidor se erige como un elemento esencial en la conectividad moderna. Desde la gestión empresarial hasta las aplicaciones cotidianas en línea, la capacidad de coordinar recursos de manera eficiente ha demostrado ser un componente clave en el tejido digital de la sociedad actual. La implementación de soluciones como el código analizado refleja la relevancia continua de esta arquitectura y su papel vital en la construcción y el mantenimiento de sistemas informáticos robustos y escalables.

Referencias

PROTECO. (s/f). *Sockets TCP/IP*. Unam.mx. Recuperado el 4 de diciembre de 2023, de http://profesores.fi-b.unam.mx/carlos/acs/Tema-04-Sockets-INTERNET-C/01_Sockets_AF_INTERNET.pdf

Código

Cliente

```
#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>

#define MAX_BUFFER_SIZE 1024

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Uso: %s <IP> <puerto>\n", argv[0]);
        exit(1);
    }

    // Crear el socket del cliente
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        perror("Error al crear el socket");
        exit(1);
    }

    // Configurar la estructura de dirección del servidor
    struct sockaddr_in serv_addr;
    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(atoi(argv[2]));

    // Convertir la dirección IP de texto a binario
    if (inet_pton(AF_INET, argv[1], &serv_addr.sin_addr) <= 0) {
        perror("Error al convertir la dirección IP");
        close(sockfd);
        exit(1);
    }

    // Conectar al servidor
    if (connect(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) ==
-1) {
        perror("Error al conectar al servidor");
        close(sockfd);
        exit(1);
    }

    printf("Conectado al servidor %s:%s\n", argv[1], argv[2]);

    while (1) {
        // Leer el comando desde la entrada estándar
        char comando[MAX_BUFFER_SIZE];
```



```

printf("Ingrese un comando: ");
fgets(comando, sizeof(comando), stdin);

// Eliminar el carácter de nueva línea de la entrada
size_t len = strlen(comando);
if (len > 0 && comando[len - 1] == '\n') {
    comando[len - 1] = '\0';
}

// Enviar el comando al servidor
if (send(sockfd, comando, strlen(comando), 0) == -1) {
    perror("Error al enviar el comando al servidor");
    close(sockfd);
    exit(1);
}

// Recibir la salida del servidor
char buffer[MAX_BUFFER_SIZE];
memset(buffer, 0, sizeof(buffer));
ssize_t bytesRecibidos = recv(sockfd, buffer, sizeof(buffer) - 1,
0);

if (bytesRecibidos == -1) {
    perror("Error al recibir la salida del servidor");
    close(sockfd);
    exit(1);
} else if (bytesRecibidos == 0) {
    // Servidor cerró la conexión
    printf("Conexión cerrada por el servidor.\n");
    break;
}

// Imprimir la salida del servidor
printf("Salida del servidor:\n%s\n", buffer);
}

// Cerrar el socket del cliente
close(sockfd);

return 0;
}

```

Servidor

```

#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>

#define MAX_BUFFER_SIZE 1024

```

```

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Uso: %s <puerto>\n", argv[0]);
        exit(1);
    }

    // Crear el socket del servidor
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        perror("Error al crear el socket");
        exit(1);
    }

    // Configurar la estructura de dirección del servidor
    struct sockaddr_in serv_addr;
    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(atoi(argv[1]));
    serv_addr.sin_addr.s_addr = INADDR_ANY;

    // Vincular el socket a la dirección y puerto
    if (bind(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) ==
-1) {
        perror("Error al vincular el socket");
        close(sockfd);
        exit(1);
    }

    // Poner el socket en estado de escucha
    if (listen(sockfd, 5) == -1) {
        perror("Error al poner el socket en estado de escucha");
        close(sockfd);
        exit(1);
    }

    printf("Servidor esperando conexiones en el puerto %s...\n", argv[1]);

    while (1) {
        // Aceptar una conexión entrante
        int clientfd = accept(sockfd, NULL, NULL);
        if (clientfd == -1) {
            perror("Error al aceptar la conexión");
            close(sockfd);
            exit(1);
        }

        struct sockaddr_in client_addr;
        socklen_t client_addr_len = sizeof(client_addr);
        if (getpeername(clientfd, (struct sockaddr*)&client_addr,
&client_addr_len) == -1) {
            perror("Error al obtener la dirección del cliente");
            close(clientfd);
            close(sockfd);
            exit(1);
        }
    }
}

```

```

        char client_ip[INET_ADDRSTRLEN];
        inet_ntop(AF_INET, &client_addr.sin_addr, client_ip,
sizeof(client_ip));

        printf("Cliente conectado desde %s:%d\n", client_ip,
ntohs(client_addr.sin_port));

        while (1) {
            // Recibir el comando del cliente
            char comando[MAX_BUFFER_SIZE];
            memset(comando, 0, sizeof(comando));
            ssize_t bytesRecibidos = recv(clientfd, comando,
sizeof(comando) - 1, 0);

            if (bytesRecibidos == -1) {
                perror("Error al recibir el comando del cliente");
                close(clientfd);
                close(sockfd);
                exit(1);
            } else if (bytesRecibidos == 0) {
                // Conexión cerrada por el cliente
                printf("Cliente desconectado desde %s:%d.\n", client_ip,
ntohs(client_addr.sin_port));
                close(clientfd);
                break;
            }

            // Verificar si el cliente envió "exit"
            if (strcmp(comando, "exit") == 0) {
                printf("Cliente %s:%d solicitó cerrar la conexión.\n",
client_ip, ntohs(client_addr.sin_port));
                close(clientfd);
                break;
            }

            // Ejecutar el comando y capturar la salida
            FILE *fp = popen(comando, "r");
            if (fp == NULL) {
                perror("Error al ejecutar el comando");
                close(clientfd);
                close(sockfd);
                exit(1);
            }

            // Leer la salida del comando
            char buffer[MAX_BUFFER_SIZE];
            memset(buffer, 0, sizeof(buffer));
            size_t bytesRead = fread(buffer, 1, sizeof(buffer) - 1, fp);

            // Enviar la salida al cliente
            if (send(clientfd, buffer, bytesRead, 0) == -1) {
                perror("Error al enviar la salida al cliente");
                close(clientfd);
                close(sockfd);
            }
        }
    }
}

```

```

        pclose(fp);
        exit(1);
    }

    // Imprimir en la terminal del servidor que se envió la
respuesta
    printf("Enviada respuesta al cliente %s:%d:\n%s\n", client_ip,
ntohs(client_addr.sin_port), buffer);

    // Cerrar el descriptor de archivo del comando
    pclose(fp);
}

// Cerrar el socket del servidor (esto nunca se alcanza en este bucle
infinito)
close(sockfd);

return 0;
}

```