INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE COMPUTO

Laboratorio de redes de computadoras

PRACTICA 7

PROGRAMA SOCKET SERVER

NOMBRE DEL ALUMNO: GARCÍA QUIROZ GUSTAVO IVAN

GRUPO: 5CV4

NOMBRE DEL PROFESOR: ALCARAZ TORRES JUAN JESUS

24/12/2023

Índice

Objetivos:

- Desarrollar un programa de socket en lenguaje C para implementar un servidor de chat.
- Implementar funciones esenciales para la comunicación entre el servidor y los clientes.
- Establecer una conexión exitosa entre el servidor y al menos un cliente.
- Facilitar el intercambio de mensajes en tiempo real entre el servidor y el cliente.
- Identificar y gestionar posibles fallos en la comunicación para asegurar la estabilidad del sistema.

Requerimientos:

- 1 Computadora Personal.
- S. O. Ubuntu 16.04.7.
- Virtualbox

INTRODUCCIÓN

En una red de ordenadores hay varios ordenadores que están conectados entre si por un cable. A través de dicho cable pueden transmitirse información. Es claro que deben estar de acuerdo en cómo transmitir esa información, de forma que cualquiera de ellos pueda entender lo que están transmitiendo los otros, de la misma forma que nosotros nos ponemos de acuerdo para hablar en inglés cuando uno es italiano, el otro francés, el otro español y el otro alemán.

Al "idioma" que utilizan los ordenadores para comunicarse cuando están en red se le denomina protocolo. Hay muchísimos protocolos de comunicación, entre los cuales el más extendido es el TCP/IP. El más extendido porque es el que se utiliza en Internet.

Aunque todo esto pueda parecer complicado y que no podemos hacer mucho con ello, lo cierto es que podemos aprovecharlo para comunicar dos programas nuestros que estén corriendo en ordenadores distintos. De hecho, con C en Linux/Unix tenemos una serie de funciones que nos permiten enviar y recibir datos de otros programas, en C o en otros lenguajes de programación, que estén corriendo en otros ordenadores de la misma red.

En este artículo no se pretende dar una descripción detallada y rigurosa del protocolo TCP/IP y lo que va alrededor de él. Simplemente se darán unas nociones básicas de manera informal, con la intención de que se pueda comprender un pequeño ejemplo de programación en C.

Está, por tanto, orientado a personas que tengan unos conocimientos básicos de C en Linux y deseen o necesiten comunicar dos programas en C que corren simultáneamente en dos ordenadores distintos conectados en red. El ejemplo propuesto puede servir como guía inicial que se pude complicar todo lo que se desee.

LOS SOCKETS

Una forma de conseguir que dos programas se transmitan datos, basada en el protocolo TCP/IP, es la programación de sockets. Un socket no es más que un "canal de comunicación" entre dos programas que corren sobre ordenadores distintos o incluso en el mismo ordenador.

Desde el punto de vista de programación, un socket no es más que un "fichero" que se abre de una manera especial. Una vez abierto se pueden escribir y leer datos de él con las habituales funciones de **read() y write()** del lenguaje C. Hablaremos de todo esto con detalle más adelante.

Existen básicamente dos tipos de "canales de comunicación" o sockets, los orientados a conexión y los no orientados a conexión.

En el primer caso ambos programas deben conectarse entre ellos con un socket y hasta que no esté establecida correctamente la conexión, ninguno de los dos puede transmitir datos. Esta es la parte TCP del protocolo TCP/IP, y garantiza que todos los datos van a llegar de un programa al otro correctamente. Se utiliza cuando la información a transmitir es importante, no se puede perder ningún dato y no importa que los programas se queden "bloqueados" esperando o transmitiendo datos. Si uno de los programas está atareado en otra cosa y no atiende la comunicación, el otro quedará bloqueado hasta que el primero lea o escriba los datos.

En el segundo caso, no es necesario que los programas se conecten. Cualquiera de ellos puede transmitir datos en cualquier momento, independientemente de que el otro programa esté "escuchando" o no. Es el llamado protocolo UDP, y garantiza que los datos que lleguen son correctos, pero no garantiza que lleguen todos. Se utiliza cuando es muy importante que el programa no se quede bloqueado y no importa que se pierdan datos. Imaginemos, por ejemplo, un programa que está controlando la temperatura de un horno y envía dicha temperatura a un ordenador en una sala de control para que éste presente unos gráficos de temperatura. Obviamente es más importante el control del horno que el perfecto refresco de los gráficos. El programa no se puede quedar bloqueado sin atender al horno simplemente porque el ordenador que muestra los gráficos esté ocupado en otra cosa.

En el ejemplo y a partir de este momento nos referimos únicamente a sockets TCP. Los UDP son básicamente iguales, aunque hay pequeñas diferencias en la forma de abrirlos y de enviar los mensajes. Puedes ver un ejemplo de programación de socket UDP.

ARQUITECTURA CLIENTE / SERVIDOR

A la hora de comunicar dos programas, existen varias posibilidades para establecer la conexión inicialmente. Una de ellas es la utilizada aquí. Uno de los programas debe estar arrancado y en espera de que otro quiera conectarse a él. Nunca da "el primer paso" en la conexión. Al programa que actúa de esta forma se le conoce como servidor. Su nombre se debe a que normalmente es el que tiene la información que sea disponible y la "sirve" al que se la pida. Por ejemplo, el servidor de páginas web tiene las páginas web y se las envía al navegador que se lo solcite.

El otro programa es el que da el primer paso. En el momento de arrancarlo o cuando lo necesite, intenta conectarse al servidor. Este programa se denomina cliente. Su nombre se debe a que es el que solicita información al servidor. El navegador de Internet pide la página web al servidor de Internet.

En este ejemplo, el servidor de páginas web se llama servidor porque está (o debería de estar) siempre encendido y pendiente de que alguien se conecte a él y le pida una página. El navegador de Internet es el cliente, puesto que se arranca cuando nosotros lo arrancamos y solicita conexión con el servidor cuando nosotros escribimos, por ejemplo, www.chuidiang.com

En el juego del Quake, debe haber un servidor que es el que tiene el escenario del juego y la situación de todos los jugadores en él. Cuando un nuevo jugador arranca el juego en su ordenador, se conecta al servidor y le pide el escenario del juego para presentarlo en la pantalla. Los movimientos que realiza el jugador se transmiten al servidor y este actualiza escenarios a todos los jugadores.

Resumiendo, servidor es el programa que permanece pasivo a la espera de que alguien solicite conexión con él, normalmente, para pedirle algún dato. Cliente es el programa que solicita la conexión para, normalmente, pedir datos al servidor.

LA CONEXIÓN

Para poder realizar la conexión entre ambos programas son necesarias varias cosas:

Dirección IP del servidor.

Cada ordenador de una red tiene asignado un número único, que sirve para identificarle y distinguirle de los demás, de forma que cuando un ordenador quiere hablar con otro, manda la información a dicho número. Es similar a nuestros números de teléfono. Si quiero hablar con mi amigo "Josechu", primero marco su número de teléfono y luego hablo con él.

El servidor no necesita la dirección de ninguno de los dos ordenadores, al igual que nosotros, para recibir una llamada por teléfono, no necesitamos saber el número de nadie, ni siquiera el nuestro. El cliente sí necesita saber el número del servidor, al igual que nosotros para llamar a alguien necesitamos saber su número de teléfono.

La dirección IP es un número del estilo 192.100.23.4. ¡Todos lo hemos visto en Internet!. En resumidas cuentas, el cliente debe conocer a qué ordenador desea conectarse. En nuestro navegador de Internet facilitamos la dirección IP del servidor al que queremos conectarnos a través de su nombre (www.chuidiang.com). Obviamente este nombre hay que traducirlo a una dirección IP, pero nuestro navegador e Internet se encargan de eso por nosotros.

Servicio que queremos crear / utilizar.

Si llamamos a una empresa, puede haber en ella muchas personas, cada una con su extensión de teléfono propia. Normalmente la persona en concreto con la que hablemos nos da igual, lo que queremos es alguien que nos atienda y nos de un determinado "servicio", como recoger una queja, darnos una información, tomarnos nota de un pedido, etc.

De la misma forma, en un mismo ordenador pueden estar corriendo varios programas servidores, cada uno de ellos dando un servicio distinto. Por ejemplo, un ordenador puede tener un servidor de Quake y un servidor de páginas web corriendo a la vez. Cuando un cliente desea conectarse, debe indicar qué servicio quiere, igual que al llamar a la empresa necesitamos decir la extensión de la persona con la que queremos hablar o, al menos, decir su nombre o el departamento al que pertenece para que la telefonista nos ponga con la persona adecuada.

Por ello, cada servicio dentro del ordenador debe tener un número único que lo identifique (como la extensión de teléfono). Estos números son enteros normales y van de 1 a 65535. Los número bajos, desde 1 a 1023 están reservados para servicios habituales de los sistemas operativos (www, ftp, mail, ping, etc). El resto están a disposición del programador y sirven para cosas como Quake.

Tanto el servidor como el cliente deben conocer el número del servicio al que atienden o se conectan. El servidor le indica al sistema operativo qué servicio quiere atender, al igual que en una empresa el empleado recién contratado (o alguien en su lugar) debe informar a la telefonista en qué extensión se encuentra.

El cliente, cuando llame a la empresa, debe dar el número de extensión (o nombre de empleado), de forma que la telefonista le ponga con la persona adecuada. En el caso del navegador de Internet, estamos indicando el servicio con la www en www.chuidiang.com, servicio de páginas web. También es posible, por ejemplo "ftp.chuidiang.com", si chuidiang.com admite clientes ftp. Nuestro ordenador es lo suficientemente listo como para saber a qué número corresponden esos servicios habituales.

EL SERVIDOR

A partir de este punto comenzamos con lo que es la programación en C de los sockets. Si no tienes unos mínimos conocimientos de C, es mejor que los adquieras antes de seguir.

Con C en Unix/Linux, los pasos que debe seguir un programa servidor son los siguientes:

- **Apertura de un socket**, mediante la función **socket()**. Esta función devuelve un descriptor de fichero normal, como puede devolverlo **open()**. La función **socket()** no hace absolutamente nada, salvo devolvernos y preparar un descriptor de fichero que el sistema posteriormente asociará a una conexión en red.
- **Avisar al sistema operativo de que hemos abierto un socket y queremos que asocie nuestro programa a dicho socket.** Se consigue mediante la función **bind()**. El sistema todavía no atenderá a las conexiones de clientes, simplemente anota que cuando empiece a hacerlo, tendrá que avisarnos a nosotros. Es en esta llamada cuando se debe indicar el número de servicio al que se quiere atender.
- **Avisar al sistema de que comience a atender dicha conexión de red.** Se consigue mediante la función **listen()**. A partir de este momento el sistema operativo anotará la conexión de cualquier cliente para pasárnosla cuando se lo pidamos. Si llegan clientes más rápido de lo que somos capaces de atenderlos, el sistema operativo hace una "cola" con ellos y nos los irá pasando según vayamos pidiéndolo.
- **Pedir y aceptar las conexiones de clientes al sistema operativo.** Para ello hacemos una llamada a la función **accept()**. Esta función le indica al sistema operativo que nos dé al siguiente cliente de la cola. Si no hay clientes se quedará bloqueada hasta que algún cliente se conecte.
- **Escribir y recibir datos del cliente, por medio de las funciones write() y read()**, que son exactamente las mismas que usamos para escribir o leer de un fichero. Obviamente, tanto cliente como servidor deben saber qué datos esperan recibir, qué datos deben enviar y en qué formato. Puedes ver cómo se pueden poner de acuerdo en estos mensajes en el apartado de mensajes.
- **Cierre de la comunicación y del socket**, por medio de la función **close()**, que es la misma que sirve para cerrar un fichero.

EL CLIENTE

Los pasos que debe seguir un programa cliente son los siguientes:

- **Apertura de un socket**, como el servidor, por medio de la función socket()
- **Solicitar conexión con el servidor** por medio de la función connect(). Dicha función quedará bloqueada hasta que el servidor acepte nuestra conexión o bien

si no hay servidor en el sitio indicado, saldrá dando un error. En esta llamada se debe facilitar la dirección IP del servidor y el número de servicio que se desea.

- **Escribir y recibir datos del servidor** por medio de las funciones write() y read().
- **Cerrar la comunicación por medio de close().**

Desarrollo

Se tiene que usar el programa emulador Oracle VirtualBox ya que servirá para poder usar el compilador de lenguaje c el cual esta en el sistema operativo Ubuntu, y es un requisito indispensable para esta practica tenerlo previamente instalado en el computador.

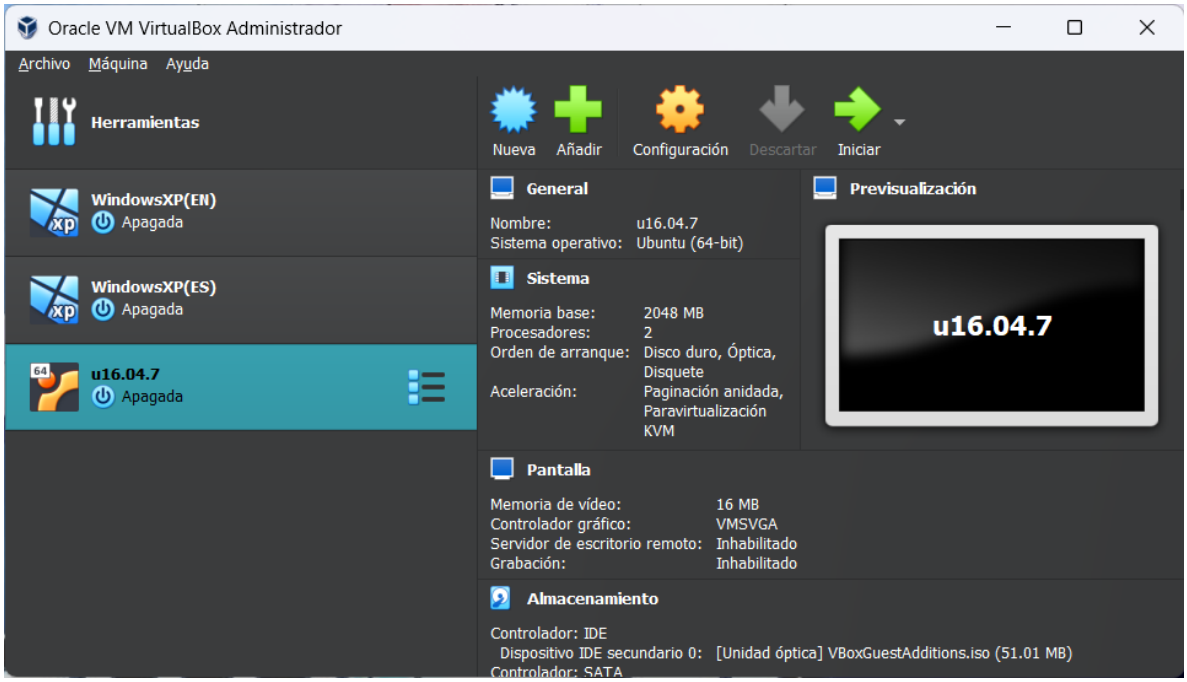


Imagen 1 Oracle VirtualBox

En las opciones que hay en la parte superior escogemos la de Iniciar para encender la maquina virtual usando el sistema operativo.

Imagen 2 Iniciar

Uso de Ubuntu

Una vez iniciado sesión en el sistema operativo vamos a dirigir hacia la dirección del donde se ha guardado el archivo y se abre la carpeta, ya que será necesario para ejecutar los programas mediante el compilador de Ubuntu.

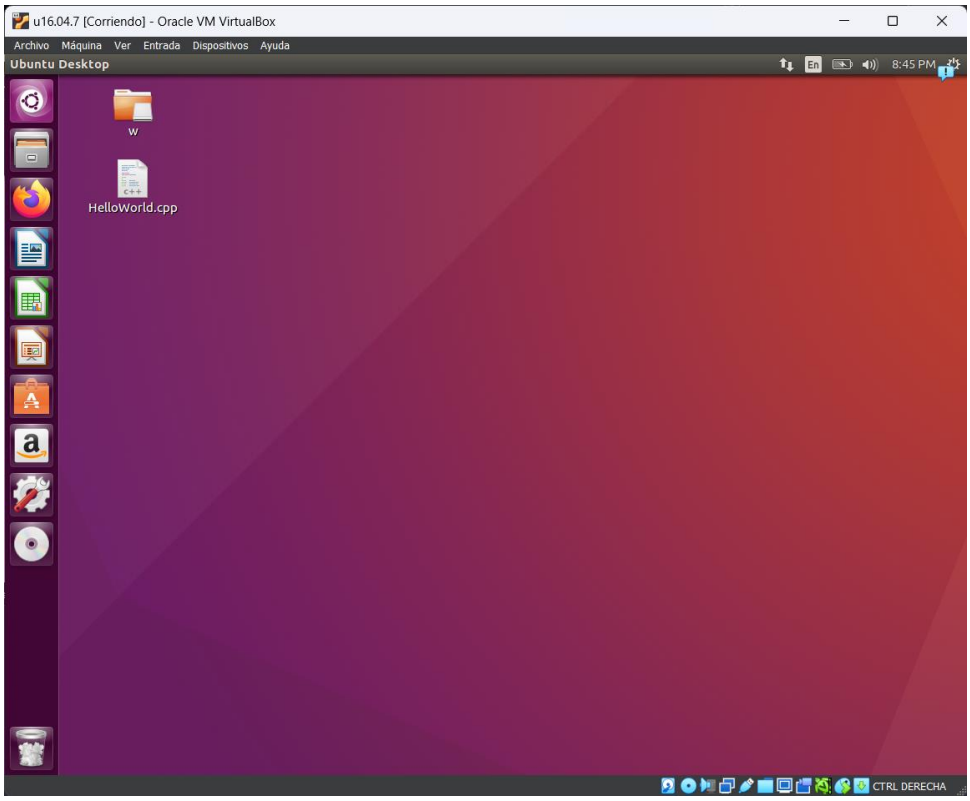
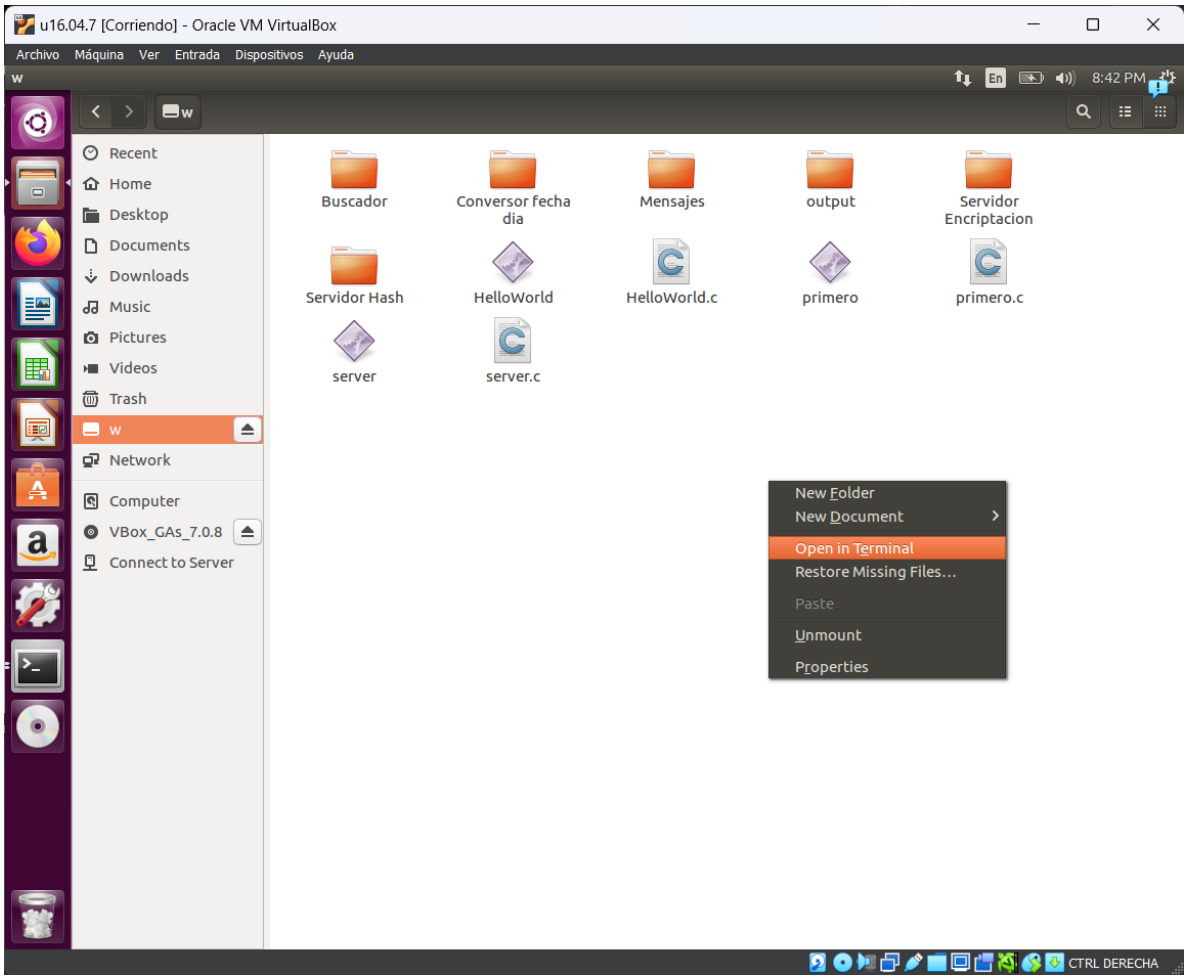


Imagen 3 Ubuntu

Se abre en este caso la carpeta “w” ya que es donde se encuentran los archivos de lenguaje c, que contienen los códigos que se van a usar , y son llamados para el cliente se usa “primero.c” y para el servidor se usa “server.c”.



Se va a abrir la terminal habiendo click izquierdo y seleccionando la opción “Open in Terminal”. Con esto hecho se ha terminado de poner el sistema para realizar la compilación que ayudara a observar el programa.

Primero se va a iniciar el programa para el servidor se usa “server.c” y se va a escribir :

```
ls
gcc server.c -o server
./server
```

.y se va a esperar el programa a la respuesta del cliente que después se va a ejecutar.

Entonces se ejecuta en otra ventana de Terminal el archivo que corresponde al cliente y se usa “primero.c” y se usan las siguientes líneas de codigo:

```
ls
gcc primero.c -o primero
./ primero
```

Se mostrara la validación del socket, del bind, y del envío del mensaje al servidor.

```
vboxuser@Ubuntu16:~/Desktop/w$ ./primero
xito al abrir el socket: Success
xito en bindxito al enviarvboxuser@Ubuntu16:~/Desktop/w$
```

Imagen 4 cliente "primero.c"

Se recibe el mensaje en el servidor y se observa el mensaje del cliente que es 'Hola red como estas' al igual que se mostrara la validación del socket, del bind, y del envío del mensaje al servidor.

```
vboxuser@Ubuntu16:~/Desktop/w$ ./server
xito al abrir el socket: Success
xito en bindxito al enviar
Hola red como estas
vboxuser@Ubuntu16:~/Desktop/w$
```

Imagen 5 servidor "server.c".

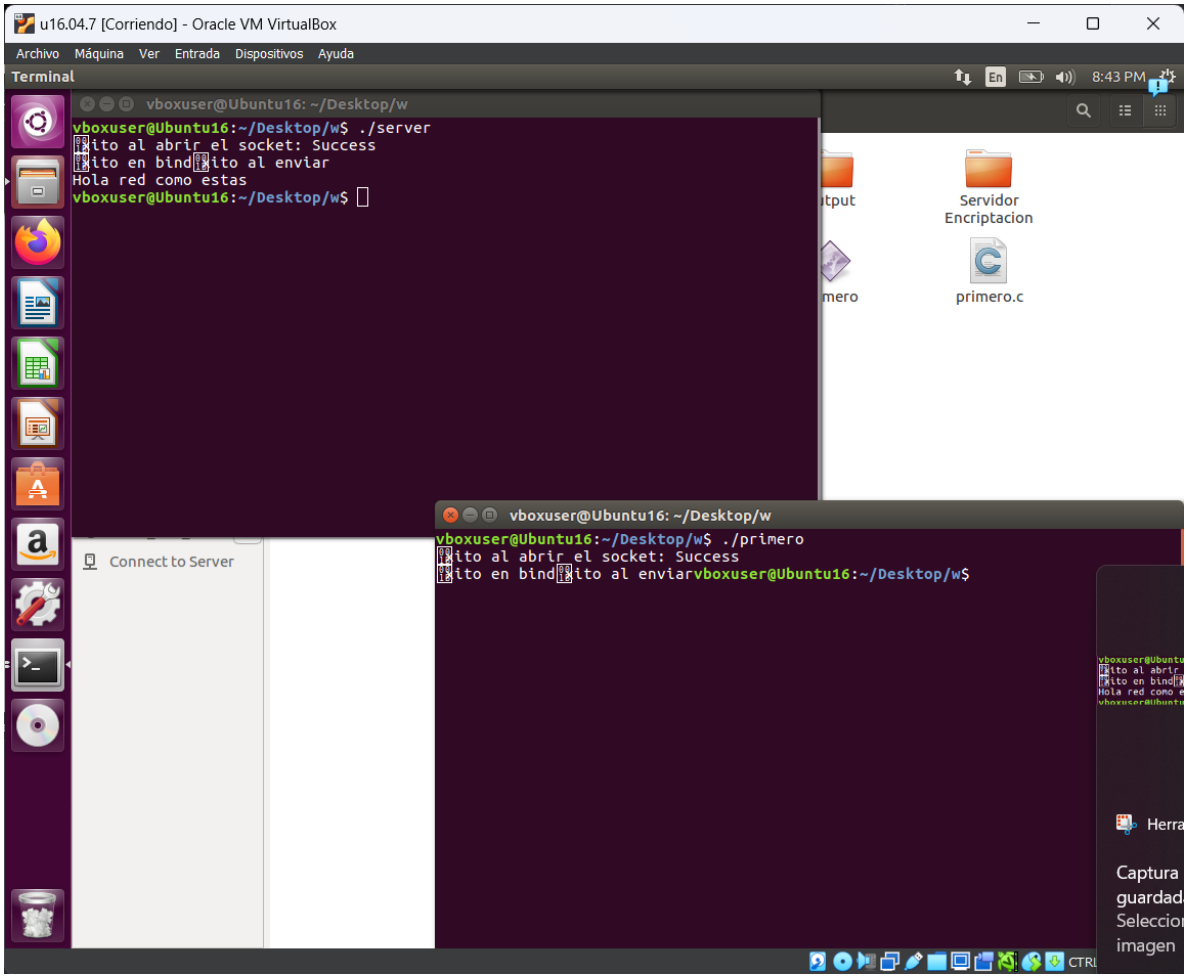


Imagen 6 cliente "primero.c" y para el servidor "server.c"

Explicación del programa

Se debe crear un código para el cliente el cual tiene una serie de paso a seguir.

El código proporcionado es un programa cliente escrito en C para comunicarse mediante sockets UDP (User Datagram Protocol). A continuación, desglosaré el código por partes para una mejor comprensión:

Inclusión de Bibliotecas:

```
#include <sys/socket.h>
   #include <netinet/in.h>
   #include <netinet/ip.h> /* superset of previous */
   #include <stdlib.h>
   #include <unistd.h>
   #include <stdio.h>
   #include <arpa/inet.h>
   #include <string.h>
```

En esta sección, se incluyen las bibliotecas necesarias para trabajar con sockets y realizar operaciones relacionadas con la red y la manipulación de cadenas.

Función principal (main):

```
int main ()
{
int udp_socket, lbind, tam;
struct sockaddr_in local, remota;
unsigned char mensaje[]="Hola red como estas";
udp_socket = socket(AF_INET, SOCK_DGRAM, 0);
if(udp_socket==-1)
{
    perror("\nError al abrir el socket");
    exit(1);
}
else
{
    perror("\Exito al abrir el socket");
    local.sin_family=AF_INET;
    local.sin_port=htons(8081);
    local.sin_addr.s_addr=INADDR_ANY;
    lbind=bind(udp_socket, (struct sockaddr *)&local,sizeof(local));
    if(lbind==-1)
    {
        perror("\nError en bind");
    }
else
{
    printf("\Exito en bind");
```

```
            local.sin_family=AF_INET;
            remota.sin_port=htons(8080);
            inet_aton("10.0.2.15",&remota.sin_addr);
            tam=sendto(udp_socket,mensaje,strlen(mensaje)+1,0,(struct
sockaddr *)&remota,sizeof(remota));
        if(tam==-1)
        {
            perror("\n Error al enviar");
            exit(1);
        }
        else
        {
            printf("\Exito al enviar");
        }
        }

        }
        close(udp_socket);
        return 0;
        }
```

Descripción del Código:

```
        udp_socket = socket(AF_INET, SOCK_DGRAM, 0);
```

Se crea un socket UDP (**SOCK_DGRAM**) y se asigna el descriptor del socket a la variable **udp_socket**.

```
            local.sin_family=AF_INET;
            local.sin_port=htons(8081);
            local.sin_addr.s_addr=INADDR_ANY;
```

Se configura la dirección local del socket, especificando el puerto y la dirección IP.

1. **Asociación del Socket con la Dirección Local (Bind):**

```
c         lbind=bind(udp_socket, (struct sockaddr *)&local,sizeof(local));
```

El socket se asocia con la dirección local configurada. Si hay un error, se muestra un mensaje de error.

2. **Configuración de la Dirección Remota:**

```
3.              local.sin_family=AF_INET;
4.              remota.sin_port=htons(8080);
5.              inet_aton("10.0.2.15",&remota.sin_addr);
```

Se configura la dirección del servidor al que se enviarán los datos, especificando el puerto y la dirección IP.

6. **Envío de Datos al Servidor:**

```
c              tam=sendto(udp_socket,mensaje,strlen(mensaje)+1,0,(struct
sockaddr *)&remota,sizeof(remota));
```

Se envía el mensaje al servidor utilizando la función **sendto**. Si hay un error, se muestra un mensaje de error.

7. **Cierre del Socket:**

```
        close(udp_socket);
```

Finalmente, se cierra el socket después de completar las operaciones.

Este código representa un cliente simple que envía un mensaje a un servidor a través de UDP. Tener un servidor que escuche en el puerto 8080 sirve para recibir los mensajes enviados por este cliente.

Después se crea un código para el servidor.

Este código representa un servidor simple que escucha en el puerto 8080 y recibe mensajes de clientes a través de UDP. A continuación, desglosaré el código por partes para una mejor comprensión:

Inclusión de Bibliotecas:

```
#include <sys/socket.h>
      #include <netinet/in.h>
      #include <netinet/ip.h> /* superset of previous */
      #include <stdlib.h>
      #include <unistd.h>
      #include <stdio.h>
      #include <arpa/inet.h>
        #include <string.h>
```

Se incluyen las bibliotecas necesarias para trabajar con sockets y realizar operaciones relacionadas con la red y la manipulación de cadenas.

Función principal (main):

```
      int main ()
      {
      int udp_socket, lbind, tam;
      struct sockaddr_in servidor, cliente;
      unsigned char mensaje[]="Hola red dns";
      unsigned char paqrec[512];
      udp_socket = socket(AF_INET, SOCK_DGRAM, 0);
      if(udp_socket==-1)
```

```
        {
            perror("\nError al abrir el socket");
            exit(1);
        }
        else
        {
            perror("\Exito al abrir el socket");
            servidor.sin_family=AF_INET;
            servidor.sin_port=htons(8080);
            servidor.sin_addr.s_addr=INADDR_ANY;
            lbind=bind(udp_socket, (struct sockaddr
*)&servidor,sizeof(servidor));
            if(lbind==-1)
            {
                perror("\nError en bind");
            }
        else
        {
            printf("\Exito en bind");
            //servidor.sin_family=AF_INET;
            //cliente.sin_port=htons(53);
            //inet_aton("8.8.8.8",&cliente.sin_addr);
            socklen_t lrecv = sizeof(cliente);
            tam=recvfrom(udp_socket,paqrec,512,0,(struct sockaddr
*)&cliente, &lrecv);

        if(tam==-1)
        {
            perror("\n Error al enviar");
            exit(1);
        }
        else
        {
            printf("\Exito al enviar");
            printf("\n%s\n",paqrec);
        }
        }

        }
        close(udp_socket);
        return 0;
        }
```

Descripción del Código:

1. **Creación del Socket:**

```
        udp_socket = socket(AF_INET, SOCK_DGRAM, 0);
```

Se crea un socket UDP (**SOCK_DGRAM**) y se asigna el descriptor del socket a la variable **udp_socket**.

2. **Configuración de la Dirección del Servidor:**

```
        servidor.sin_family=AF_INET;
            servidor.sin_port=htons(8080);
            servidor.sin_addr.s_addr=INADDR_ANY;
```

Se configura la dirección del servidor, especificando el puerto y la dirección IP.

3. **Asociación del Socket con la Dirección del Servidor (Bind):**

```
        lbind=bind(udp_socket, (struct sockaddr
*)&servidor,sizeof(servidor));
```

El socket se asocia con la dirección del servidor configurada. Si hay un error, se muestra un mensaje de error.

4. **Recepción de Datos del Cliente:**

```
    tam=recvfrom(udp_socket,paqrec,512,0,(struct sockaddr *)&cliente,
&lrecv);
```

Se recibe el mensaje del cliente utilizando la función **recvfrom**. Si hay un error, se muestra un mensaje de error.

5. **Cierre del Socket:**

```
    close(udp_socket);
```

Finalmente, se cierra el socket después de completar las operaciones.

Este código representa un servidor UDP básico que escucha en el puerto 8080 y muestra en la consola el mensaje recibido del cliente. Asegúrate de tener un cliente que envíe mensajes al servidor en el mismo puerto para probar la comunicación.

Anexo código fuente del servidor

Server.c

```
#include <sys/socket.h>
      #include <netinet/in.h>
      #include <netinet/ip.h> /* superset of previous */
      #include <stdlib.h>
      #include <unistd.h>
      #include <stdio.h>
      #include <arpa/inet.h>
        #include <string.h>

       int main ()
       {
       int udp_socket, lbind, tam;
       struct sockaddr_in servidor, cliente;
       unsigned char mensaje[]="Hola red dns";
       unsigned char paqrec[512];
       udp_socket = socket(AF_INET, SOCK_DGRAM, 0);
       if(udp_socket==-1)
       {
           perror("\nError al abrir el socket");
           exit(1);
       }
       else
       {
           perror("\Exito al abrir el socket");
           servidor.sin_family=AF_INET;
           servidor.sin_port=htons(8080);
           servidor.sin_addr.s_addr=INADDR_ANY;
           lbind=bind(udp_socket, (struct sockaddr
*)&servidor,sizeof(servidor));
           if(lbind==-1)
           {
               perror("\nError en bind");
           }
       else
       {
           printf("\Exito en bind");
           //servidor.sin_family=AF_INET;
           //cliente.sin_port=htons(53);
           //inet_aton("8.8.8.8",&cliente.sin_addr);
           socklen_t lrecv = sizeof(cliente);
           tam=recvfrom(udp_socket,paqrec,512,0,(struct sockaddr
*)&cliente, &lrecv);
```

```
    if(tam==-1)
    {
        perror("\n Error al enviar");
        exit(1);
    }
    else
    {
        printf("\Exito al enviar");
        printf("\n%s\n",paqrec);
    }
    }

    }
    close(udp_socket);
    return 0;
    }
```

Cliente.c

```
#include <sys/socket.h>
   #include <netinet/in.h>
   #include <netinet/ip.h> /* superset of previous */
   #include <stdlib.h>
   #include <unistd.h>
   #include <stdio.h>
   #include <arpa/inet.h>
   #include <string.h>


    int main ()
    {
    int udp_socket, lbind, tam;
    struct sockaddr_in local, remota;
    unsigned char mensaje[]="Hola red como estas";
    udp_socket = socket(AF_INET, SOCK_DGRAM, 0);
    if(udp_socket==-1)
    {
        perror("\nError al abrir el socket");
        exit(1);
    }
    else
    {
        perror("\Exito al abrir el socket");
        local.sin_family=AF_INET;
```

```
            local.sin_port=htons(8081);
            local.sin_addr.s_addr=INADDR_ANY;
            lbind=bind(udp_socket, (struct sockaddr *)&local,sizeof(local));
            if(lbind==-1)
            {
                perror("\nError en bind");
            }
        else
        {
            printf("\Exito en bind");
            local.sin_family=AF_INET;
            remota.sin_port=htons(8080);
            inet_aton("10.0.2.15",&remota.sin_addr);
            tam=sendto(udp_socket,mensaje,strlen(mensaje)+1,0,(struct
sockaddr *)&remota,sizeof(remota));
        if(tam==-1)
        {
            perror("\n Error al enviar");
            exit(1);
        }
        else
        {
            printf("\Exito al enviar");
        }
        }

        }
        close(udp_socket);
        return 0;
        }
```

Resultados

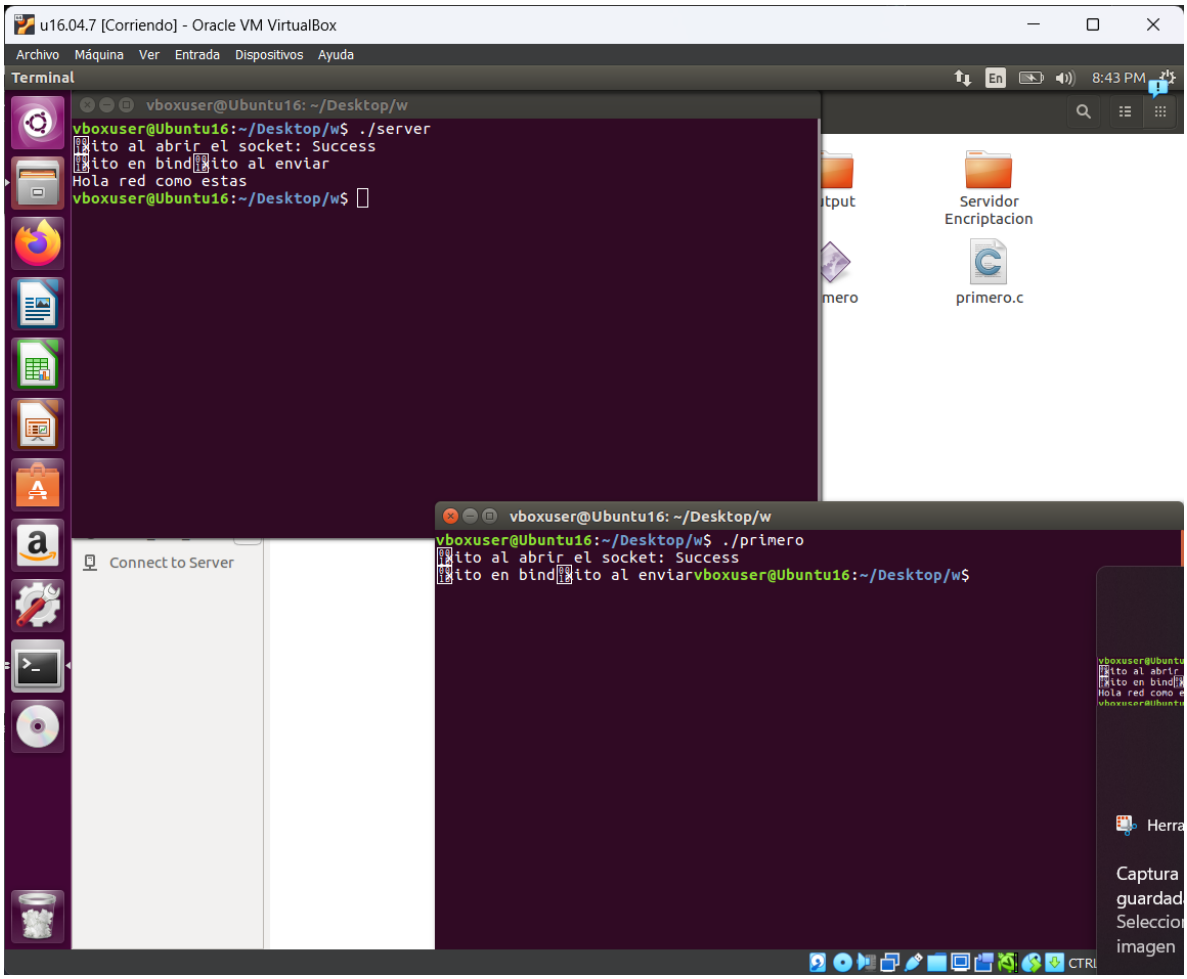


Imagen 7 cliente "primero.c" y para el servidor "server.c"

Conclusión

La práctica del uso de sockets en C para implementar una comunicación cliente-servidor mediante el protocolo UDP ha proporcionado una valiosa experiencia en el desarrollo de aplicaciones de red. El empleo de sockets ha permitido establecer una conexión eficiente y escalable entre un cliente y un servidor, facilitando la transmisión de datos de manera rápida y fiable.

Aprendí a diseñar y desarrollar aplicaciones de red utilizando sockets en C, centrándome en el protocolo UDP. Entendí cómo establecer la comunicación entre un cliente y un servidor, configurar direcciones locales y remotas, y gestionar la transmisión eficiente de datos. Además, adquirí experiencia en la detección y manejo de errores, así como en la importancia de una configuración precisa para lograr una conexión exitosa. En resumen, la práctica con sockets UDP en C me proporcionó habilidades fundamentales para la programación de red y una comprensión más profunda de los aspectos prácticos de la comunicación en entornos distribuidos.

Bibliografía

Canonical. (s/f). *Ubuntu releases*. Ubuntu.com. Recuperado el 25 de diciembre de 2023, de https://releases.ubuntu.com/

dieguiariel [@dieguiariel]. (2020, septiembre 5). *Progamas Ejemplo Sockets en C en Linux Cliente Servidor*. Youtube. https://www.youtube.com/watch?v=AfDfmsh8OGo

Hasenmueller, A. (s/f). *Oracle VM VirtualBox*. Virtualbox.org. Recuperado el 25 de diciembre de 2023, de http://virtualbox.org

Roxas, C. (s/f). *Sockets en C de Unix/Linux*. Chuidiang.org. Recuperado el 25 de diciembre de 2023, de https://old.chuidiang.org/clinux/sockets/sockets_simp.php