I. PROGRAMACIÓN CON SOCKETS

II. OBJETIVOS

Al finalizar la presente práctica el estudiante:

1. Comprende y explica la comunicación mediante sockets UDP y TCP tanto en plataformas Windows como Linux
2. Escribe aplicaciones de red que utilizan sockets UDP y TCP para comunicarse entre sí.
3. Escribe aplicaciones de red, que permiten la comunicación en plataformas heterogéneas Windows y Linux.
4. Verifica las comunicaciones de red mediante herramientas de monitoreo.

III. MATERIALES Y EQUIPOS.

Los materiales que utilizaremos en los trabajos de laboratorio son:

1. Computador con sistema operativo Linux y compilador para el lenguaje de programación C.
2. Computador con sistema operativo Windows y compilador para el lenguaje de programación C.
3. Manuales de programación en C para Linux y Windows.
4. Manuales de programación con sockets en Linux y Windows.
5. Herramienta de monitoreo Wireshark para Linux y Windows.

IV. PRE-REQUISITOS

Para mejores resultados, es recomendable que, previamente, el estudiante:

1. Tenga conocimientos de programación con el lenguaje de programación C para Linux
2. Tenga conocimientos de programación con el lenguaje de programación C para Windows
3. Conozca la teoría de los protocolos de comunicación TCP y UDP.

V. MARCO TEÓRICO

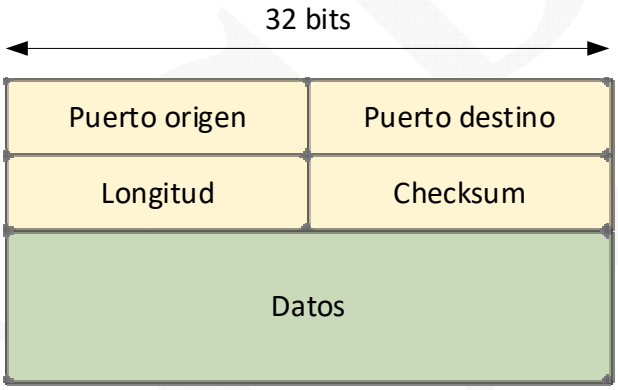
PROTOCOLO UDP

UDP es un protocolo orientado a mensajes, es decir, cada segmento UDP contiene un mensaje íntegro, con un tamaño máximo de 65535 – 8 (encabezado UDP) – 20 (encabezado IPv4) = 65507 bytes.

UDP es un protocolo de "mejor esfuerzo", sin estado (es decir, no requiere mantener variables asociadas a una conexión) y no fiable.

Estas características hacen de UDP un protocolo atractivo para aplicaciones tolerantes a pérdidas o que no requieren la sobrecarga de retransmisiones en caso de errores.

En el gráfico se muestra un segmento UDP



Una descripción más amplia sobre las características de la programación mediante sockets UDP puede obtenerse en los textos de la referencia bibliográfica.

PROTOCOLO TCP

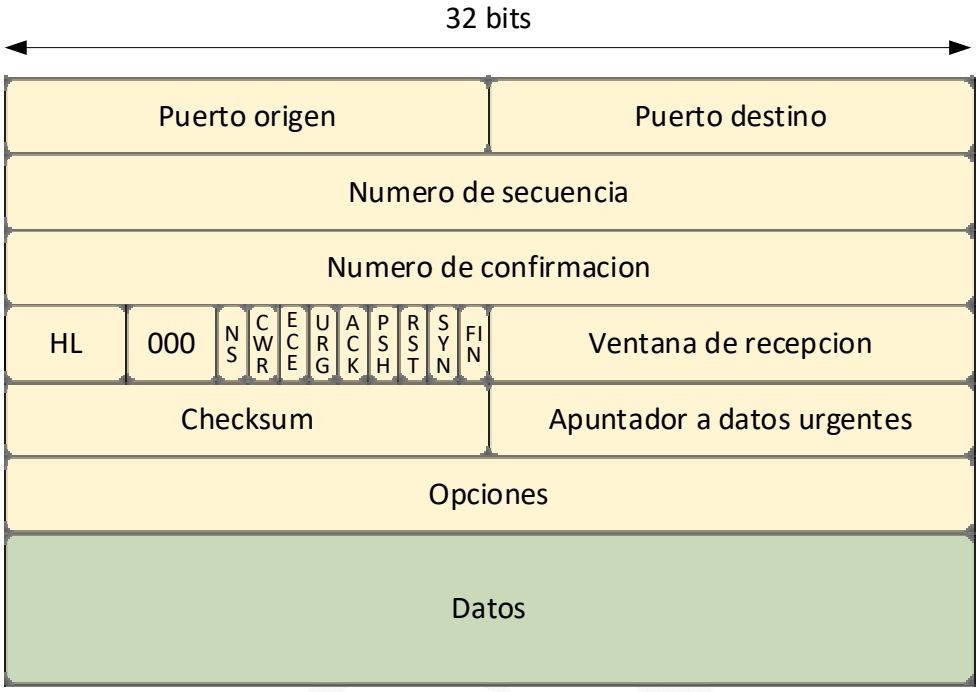
TCP es un protocolo orientado a la conexión, que provee un servicio de comunicación fiable entre procesos remotos.

TCP es un protocolo de transmisión de flujos de bytes que, además, incorpora mecanismos para el control de flujo y el control de congestionamiento

TCP se utiliza cuando se requiere proveer servicios que no toleran pérdidas ni errores como, por ejemplo, la transferencia de archivos o el envío de mensajes de correo.

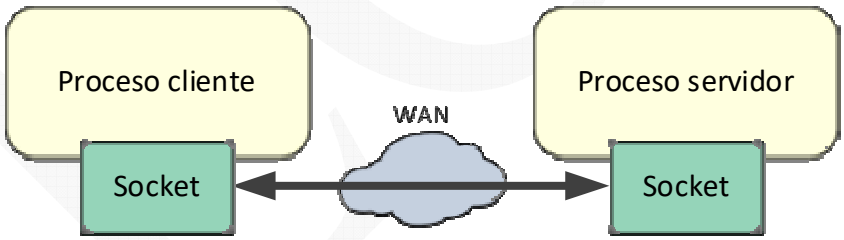
Para proveer los servicios TCP, es necesario que en los extremos de una conexión TCP se mantengan muchas variables que hacen que el protocolo sea complejo y costoso en términos de recursos asociados a las conexiones.

En el gráfico se muestra un segmento TCP

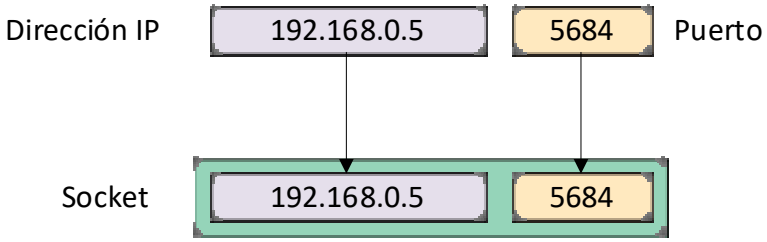


SOCKETS

Un socket es un extremo de una conexión entre aplicaciones de red. Un socket provee la interfaz a través de la cual procesos en equipos extremos pueden comunicarse.

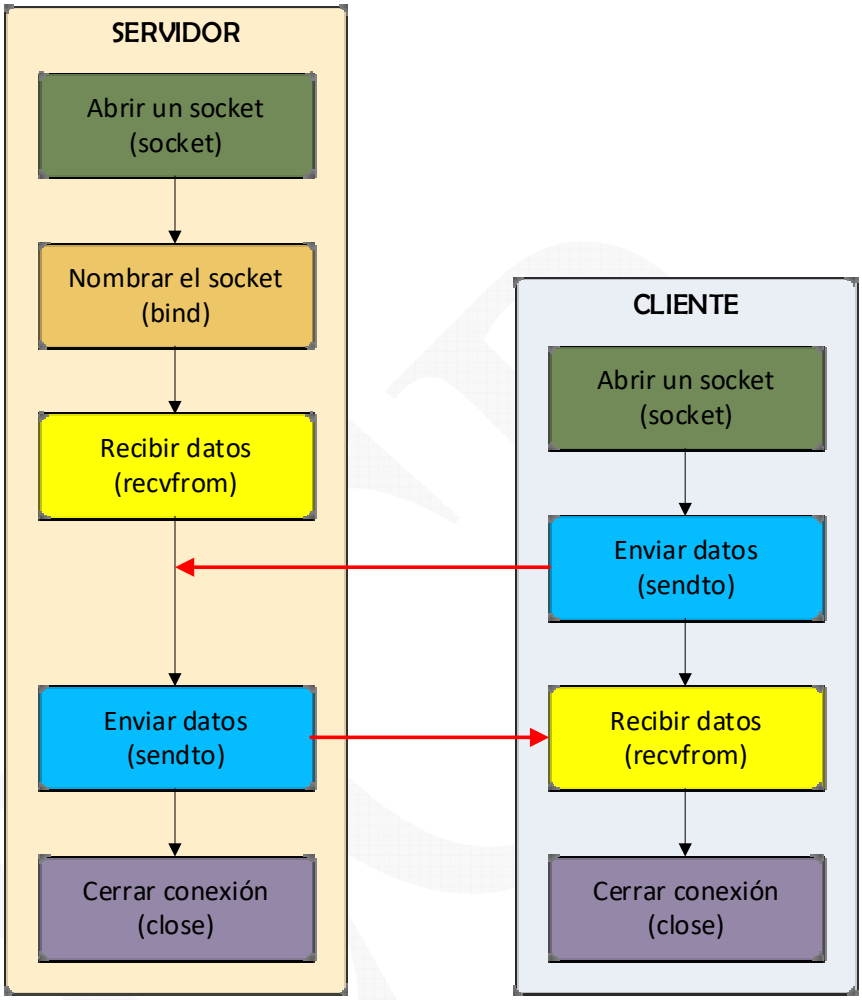


Los sockets comprenden la dirección IP y el número del puerto en elque un proceso espera datos



COMUNICACIÓN ENTRE PROCESOS UTILIZANDO SOCKETS UDP

El siguiente diagrama muestra la comunicación entre procesos remotos utilizando sockets UDP



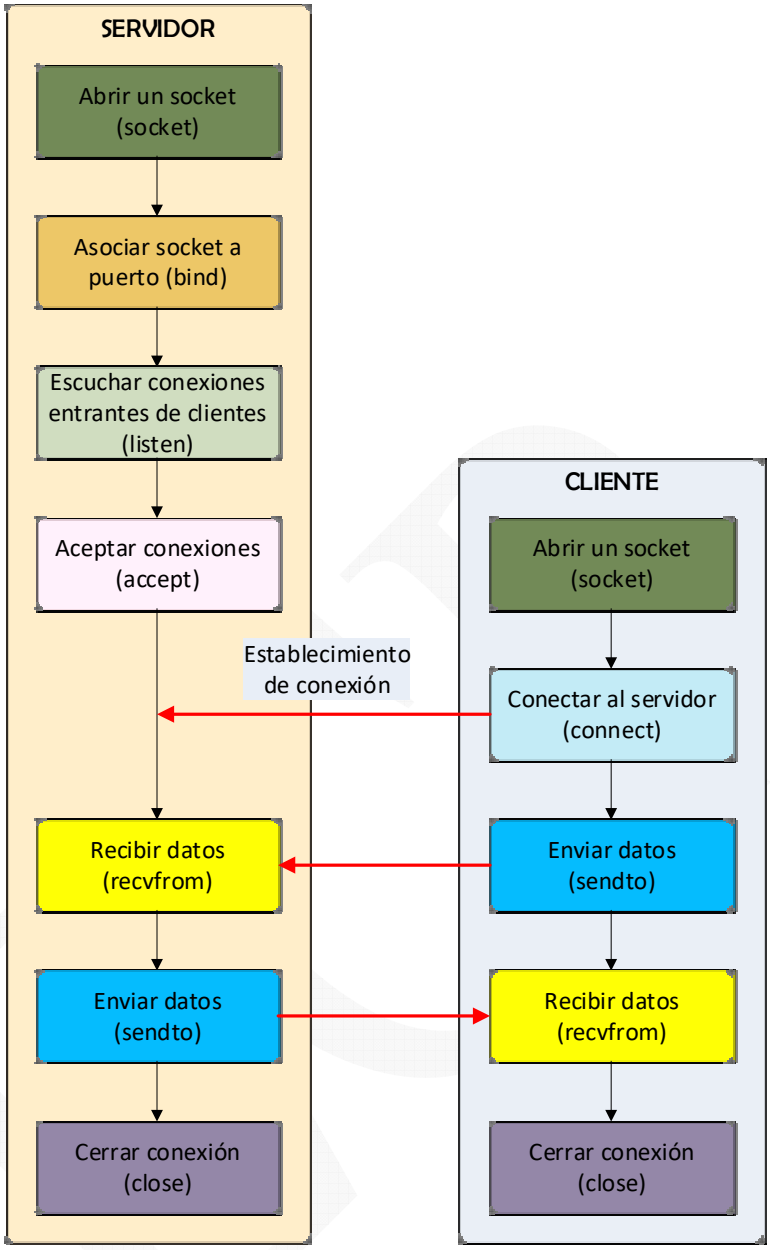
COMUNICACIÓN ENTRE PROCESOS UTILIZANDO SOCKETS TCP

La comunicación entre procesos, mediante sockets TCP, requiere pasos adicionales, comparado con la comunicación entre procesos mediante sockets UDP.

Antes de poder enviar segmentos, se debe establecer una conexión entre el proceso cliente y el proceso servidor y solo después se podrá enviar datos

Al terminar el intercambio TCP, se debe cerrar de forma ordenada la conexión previamente establecida.

El siguiente diagrama muestra la comunicación entre procesos remotos utilizando sockets TCP.



ESTRUCTURAS PARA EL MANEJO DE SOCKETS

Para gestionar la información relacionada con los sockets, se utilizan estructuras de datos definidas en la librería <netinet/in.h>

Estructura in_addr – Almacena una dirección IPv4

```
struct in_addr{
    in_addr_t    s_addr; //Dirección IPv4 en orden de bytes de red
};
```

s_addr – Dirección IPv4 de 32 bits

Estructura sockaddr_in – Registra los atributos relacionados con un socket IPv4

```
struct sockaddr_in {
     uint8_t          sin_len;      //Longitud de la estructura(16)
     sa_family_t      sin_family;   //AF_INET
     in_port_t        sin_port;     //Numero de Puerto TCP o UDP
     struct in_addr   sin_addr;     //Dirección IPv4 de 32 bits
     char             sin_zero[8];  //No se usa. Siempre se pone 0
};
```

sin_len – longitud total de la dirección
sin_family – familia de direcciones asignada al crear el socket. Para TCP/IP es AF_INET
sin_port – el número de puerto TCP o UDP (dirección de transporte) de 16 bits
sin_addr – la dirección IP para el socket
sin_zero[8] – no se usa. Siempre se pone en cero.

Estructura hostent – contiene información relevante sobre un host

```
struct  hostent {
   char    *h_name;         //Nombre oficial del host
   char    **h_aliases;     //Lista de alias
   int     h_addrtype;      //Tipo de dirección del host
   int     h_length;        //Longitud de la dirección
   char    **h_addr_list;   //Lista de direcciones desde el servidor
                              de nombres
};
```

h_name – nombre del host
h_aliases – lista de alias
h_addrtype – tipo de dirección del host
h_length – longitud de dirección
h_addr_list – lista de direcciones desde el servidor de nombres

FUNCIONES PARA EL MANEJO DE SOCKETS

CREAR SOCKETS

`socket()` - Crea un terminal de comunicación

Sintáxis

```
int socket(  int domain,
             int type,
             int protocol );
```

domain – Especifica la familia de protocolo que utilizará el socket
PF_INET : para IPv4

PF_INET6 : para IPv6

type – Tipo de socket que se creará:
SOCK_STREAM : socket TCP
SOCK_DGRAM : socket UDP

protocol – Indica el protocolo que se usará con el socket, si es 0, se usa el protocolo por defecto de acuerdo al tipo de socket creado

Al crearse el socket, este se mantiene desconectado.

ASOCIAR UN SOCKET A UN PUERTO

`bind()` – Asocia una dirección de red y un número de puerto a un socket

Sintáxis

```
int bind(  int                     socket,
           const struct sockaddr   *address,
           socklen_t               address_len  );
```

socket – Descriptor de archivo del socket que se vinculará
address – Apunta a una estructura sockaddr que contiene la dirección que se vinculará al socket. La longitud y formato de la dirección depende de la familia de direcciones del socket
address_len – Especifica la longitud de la estructura sockaddr referenciada por el campo address.

Si se utiliza como número de puerto el 0 (cero), el sistema operativo asigna el primer puerto libre disponible.

Si se utiliza como dirección IP el valor INADDR_ANY (0.0.0.0), el socket escuchará en cualquier dirección IP

CERRAR UNA CONEXIÓN

`close()` – Cierra la comunicación entre el cliente y el servidor

Sintáxis

```
int close( int sockfd );
```

sockfd – Descriptor de socket que se desea cerrar

`shutdown()` – Cierra la comunicación entre el cliente y el servidor, permitiendo un nivel de cierre más controlado que `close()`

Sintáxis

```
int shutdown(  int socket,
               int how  );
```

socket – Descriptor de socket
how – Especifica el tipo de cierre, los cuales pueden ser:
- SHUT_RD : Desactiva las operaciones de recepción
- SHUT_WR : Desactiva las operaciones de envío
- SHUT_RDWR : Desactiva toda operación de envío y recepción

Esta función no libera el descriptor de socket, por lo que debe ser seguida de una función `close()`.

ENVIAR MENSAJES A TRAVES DE UN SOCKET

`sendto()`– Envía un mensaje a través de a través de un socket orientado a la conexión o no orientado a la conexión. Si el socket es no orientado a la conexión, el mensaje se debe enviar a la dirección especificada por dest_addr. Si el socket es orientado a la conexión, dest_addr debe ser ignorado.

Sintáxis

```
ssize_t sendto(  int                     socket,
                 const void              *message,
                 size_t                  length,
                 int                     flags,
                 const struct sockaddr   *dest_addr,
                 socklen_t               dest_len );
```

socket – Especifica el descriptor de archivo del socket
message – Apunta a un buffer que contiene el mensaje que se enviará
length – Especifica el tamaño del mensaje en bytes
flags – Especifica el tipo de transmisión del mensaje. Los valores de este argumento se forman aplicando la operación OR cero o más de las siguientes banderas:
- MSG_EOR – Termina un registro
- MSG_OOB – Envía datos fuera de banda en sockets que soporten datos fuera de banda. El significado y semántica de los datos fuera de banda son específicos de los protocolos.
- MSG_NOSIGNAL – Solicita no enviar la señal SIGPIPE si se hace un intento por enviar a través de un socket orientado a flujo que ya no está conectado.

dest_addr – Apunta a una estructura sockaddr que contiene la dirección destino. La longitud y formato de la dirección depende de la familia de direcciones del socket.
dest_len – Especifica la longitud de la estructura sockaddr apuntada por el argumento dest_addr

RECIBIR MENSAJES A TRAVÉS DE UN SOCKET

recvfrom()– Recibe un mensaje de un socket orientado a la conexión a un socket no orientado a la conexión. Normalmente se usa con sockets no orientados a la conexión porque permite a la aplicación recuperar la dirección origen de los datos recibidos.

Sintáxis

```
ssize_t recvfrom(int socket,
                 void             *restrict buffer,
                 size_t           length,
                 int              flags,
                 struct sockaddr  *restrict address,
                 socklen_t        *restrict address_len );
```

socket – especifica el descriptor de archivo del socket
buffer – apunta al buffer donde se almacenará el mensaje
length – especifica la longitud en bytes del buffer apuntado por *buffer*
flags – especifica el tipo de transmisión del mensaje. Los valores de este argumento se forman aplicando la operación OR cero o más de las siguientes banderas:
- MSG_PEEK – examina un mensaje entrante. Los datos se tratan como no leidos y el siguiente recvfrom() o función similar debe aún retornar este datos
- MSG_OOB – solicita datos fuera de banda. El significado y semántica de los datos fuera de banda son específicos de los protocolos.
- MSG_WAITALL – en sockets SOCK_STREAM solicita que la función se bloquee hasta que la totalidad de los datos pueda devolverse. Esta función puede devolver la cantidad mínima de datos si el socket es un socket basado en mensajes, si una señal es capturada, si la conexión es terminada, si MSG_PEEK fue especificada, o si existe un error pendiente del socket.

address – apuntador nulo o apuntador a una estructura sockaddr en el que la dirección remitente se almacena. La longitud y formato de la dirección dependen de la familia de direcciones del socket.
address_len – apuntador a nulo si address es un apuntador a nulo. Apuntador a un objeto socklen_t que en entrada especifica la longitud de la estructura sockaddr y en salida especifica la longitud de la dirección almacenada.

Existen más funciones para el manejo de sockets. Estas pueden consultarse en la bibliografía de referencia.

WINSOCK

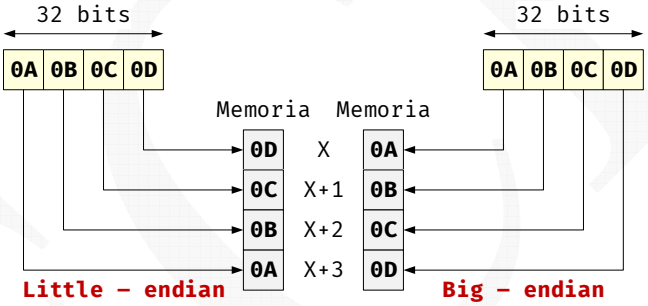
WINSOCK es la implementación de sockets en Windows. Las funciones de manejo de sockets son idénticas a las de las plataformas POSIX (Unix, Linux) y salvo pequeñas modificaciones, el código es portable siempre que se utilice C estándar. Para información más detallada, consulte las referencias incluidas en la presente guía.

ORDEN DE ALMACENAMIENTO DE DATOS EN MEMORIA

La memoria de los computadores puede representarse como un arreglo de bytes con direcciones sucesivas x, x+1, x+2, etc.

Un byte 0x0A puede almacenarse en la posición de memoria X, pero ¿cómo se almacena un elemento de 4 bytes, por ejemplo, 0x0A0B0C0D? Esto dependerá de si el computador es Little – endian o Big – endian.

Las dos alternativas se muestran en el siguiente gráfico:



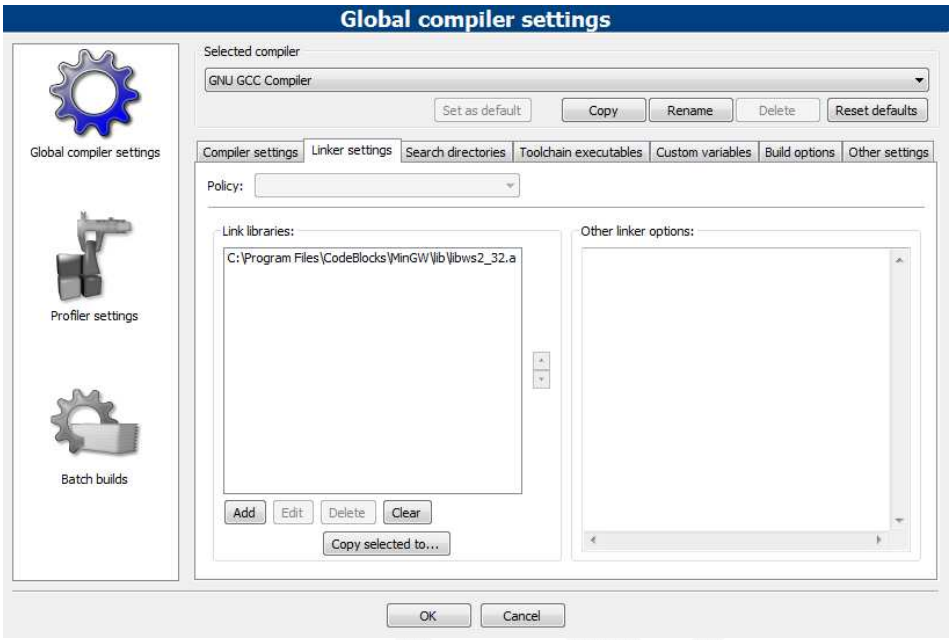
Esta diferencia de esquemas de almacenamiento, crea un problema al desarrollar aplicaciones de red, por lo que, para compatibilizar estos sistemas, se implementa funciones especiales:

htons()	host **to** network **s**hort
htonl()	host **to** network **l**ong
ntohs()	network **to** host **s**hort
ntohl()	network **to** host **l**ong

El esquema *Big-Endian* se denomina *Network Byte Order*, y es el que se utiliza para transmisión de datos en red; mientras que *Host Byte Order* es el esquema utilizado en los equipos terminales. Si el procesador es Intel, el *Host Byte Order* será *Little-Endian*.

ENTORNO DE DESARROLLO DE APLICACIONES

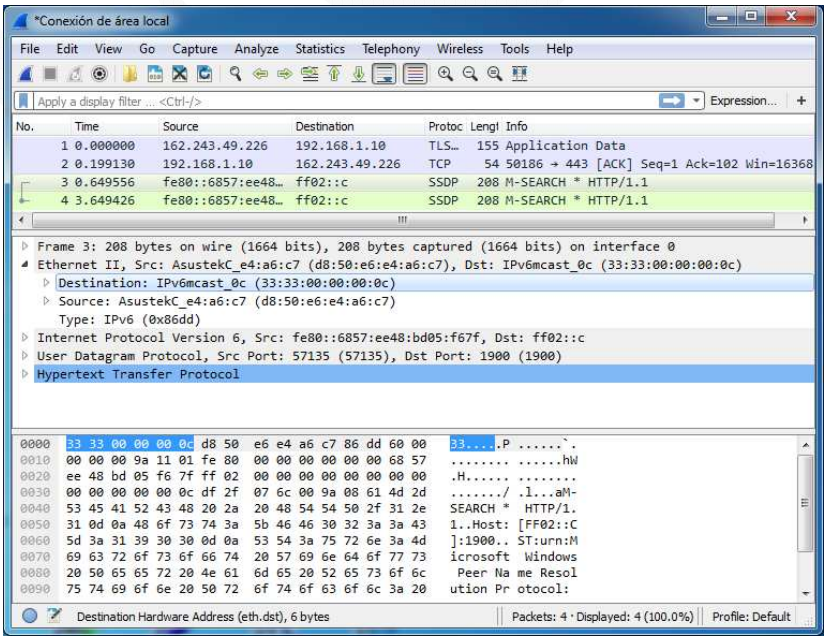
Para el desarrollo de la aplicación para Windows utilizamos el IDE Code::Blocks, en el que, para compilar el programa debemos agregar a las librerías de enlace la librería libws2_32.a ubicada en C:\Program Files\CodeBlocks\MinGW\lib\ desde la opción Settings-> Compiler-> Linker settings-> add.



En caso de utilizar otro IDE para la creación de los programas, deberá documentarse en la necesidad o no de modificar las opciones de compilación.

WIRESHARK

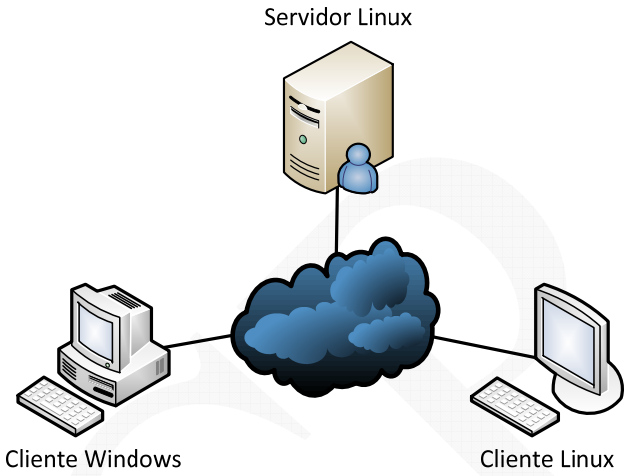
Es una aplicación que permite capturar los datos que se transmite a través de una interfaz de red para su estudio y análisis. La siguiente captura de pantalla muestra la información datos que puede ser analizada con ayuda de esta herramienta:



Mayor información sobre Wireshark, asi como su uso, puede obtenerse en los sitios web www.wireshark.org y http://www.wireshark.org/docs/.

VI. PRACTICAS DE LABORATORIO.

1. Escribir una aplicación Cliente/Servidor utilizando sockets UDP, en la que el cliente envía un mensaje al servidor y este responde con una confirmación de recepción. El servidor debe implementarse en Linux, mientras que los clientes deben operar tanto en Windows como Linux.



Solución

CÓDIGO DEL SERVIDOR

```
//Nombre        : server_udp
//Proposito     : Recibe un mensaje y lo devuelve al cliente
                  usando UDP
//Autor         : Edwin Carrasco (adaptado de
                  http://www.linuxhowtos.org/C_C++/socket.htm)
//FCreacion     : 02-Nov-2010
//FModific.     : 24-06-2013
//Compilacion   : gcc -o servidor server_udp.c
//Ejecucion     : ./servidor <nro_Puerto>


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

void AvisarError(char *mensaje)
{
    perror(mensaje);
    exit(0);
}
```

```
int main(int argc, char *argv[])
{
   //Variables
   int descriptorSocket;
   unsigned int tamSocketServidor;
   unsigned int tamSocketCliente;
   int n;
   struct sockaddr_in socketSrv;
   struct sockaddr_in socketCli;
   char buf[1024];
   int longMsg;
   char *rpta;


   if (argc < 2) {
      fprintf(stderr, "ERROR, no se especifico un puerto\n");
      exit(0);
   }

   //Crear socket de tipo datagrama (UDP)
   descriptorSocket = socket(AF_INET, SOCK_DGRAM, 0);

   if (descriptorSocket < 0)
      AvisarError("Error al abrir el socket");

   tamSocketServidor = sizeof(socketSrv);

   //Pone en cero la estructura
   bzero(&socketSrv, tamSocketServidor);

   socketSrv.sin_family = AF_INET; //socket tipo Internet-IPv4
   socketSrv.sin_addr.s_addr = INADDR_ANY; //No se necesita
                                           conocer IP del host
   socketSrv.sin_port = htons(atoi(argv[1])); //Puerto del
                                                servidor

   if (bind(descriptorSocket,(struct sockaddr *) &socketSrv,
            tamSocketServidor) < 0)
       AvisarError("Error durante binding");

   tamSocketCliente = sizeof(struct sockaddr_in);

   while (1) {
       //Recibir mensaje del cliente
       n = recvfrom(descriptorSocket, buf, 1024, 0, (struct
           sockaddr *) &socketCli, &tamSocketCliente);

       if (n < 0)
         AvisarError("Error en recvfrom");

       write(1, "Se recibio el mensaje: ", 23);
       write(1, buf, n);
       rpta = buf;
```

```
        //Enviar mensaje de confirmacion
        longMsg = n;

        n = sendto(descriptorSocket, rpta, longMsg, 0, (struct
                    sockaddr *) &socketCli, tamSocketCliente);

        if (n < 0)
         AvisarError("Error durante sendto");
    }
    return 0;
 }
```

CÓDIGO DEL CLIENTE EN LINUX

```
//Nombre        : client_udp
//Proposito     : Envia un mensaje a un servidor y muestra el
                  mensaje devuelto por este usando UDP
//Autor         : Edwin Carrasco (adaptado de
                  http://www.linuxhowtos.org/C_C++/socket.htm)
//FCreacion     : 02-Nov-2010
//FModific.     : 24-06-2013
//Compilacion   : gcc -o cliente client_udp.c
//Ejecucion     : ./cliente <IP Servidor> <nro_Puerto>

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

void AvisarError(char *mensaje)
{
     perror(mensaje);
     exit(0);
}


int main(int argc, char *argv[])
{
    int sock;
    unsigned int length;
    int  n;
    struct sockaddr_in servidor, cliente;
    struct hostent *hp;
    char buffer[256];

    if (argc != 3)
    {
     printf("Uso: <IP servidor> <puerto>\n");
     exit(1);
    }
```

```
    //Crear el socket UDP
    sock = socket(AF_INET, SOCK_DGRAM, 0);

    if (sock < 0)
     AvisarError("Error al crear el socket");

    servidor.sin_family = AF_INET;  //Socket tipo Internet-IPv4

    //Obtener datos del servidor
    hp = gethostbyname(argv[1]);

    if (hp==0)
     AvisarError("Host desconocido");

    //establecer direccion del servidor en la estructura
     servidor
    bcopy( (char *) hp->h_addr, (char *) &servidor.sin_addr,
           hp->h_length);

    //Puerto en el que escucha el servidor
    servidor.sin_port = htons(atoi(argv[2]));

    length=sizeof(struct sockaddr_in);

    printf("Por favor ingrese el mensaje: ");

    bzero(buffer, 256);

    //Leer el mensaje de la entrada estandar (teclado)
    fgets(buffer, 255, stdin);

    //Enviar los datos al servidor
    n = sendto(sock, buffer, 256, 0, (struct sockaddr *)
                &servidor, length);

    if (n < 0)
     AvisarError("Error durante Sendto");
    //Leer mensaje del servidor
    n = recvfrom(sock, buffer, 256, 0, (struct sockaddr *)
                  &cliente, &length);
    if (n < 0)
     AvisarError("Error durante recvfrom");

    //Mostrar mensaje devuelto por el servidor
    write(1,"El servidor indica que recibio el mensaje: ",43);
    write(1, buffer, n);

    return 0;
}
```

CÓDIGO DEL CLIENTE WINDOWS

```
//Nombre      : cliente_udp
//Proposito   : Envia una cadena al servidor y muestra la
                cadena devuelta por este usando UDP
//Autor       : Edwin Carrasco (adaptado de
                http://www.linuxhowtos.org/C_C++/socket.htm)
//FCreacion   : 24-06-2013
//FModific.   : --

#include <stdio.h>
#include <winsock.h>
#include <stdlib.h>

void AvisarError(char *mensaje)
{
    perror(mensaje);
    exit(0);
}

int main(int argc, char *argv[])
{
   int sock, length, n;
   struct sockaddr_in servidor, cliente;
   char buffer[256];
   WSADATA wsaData;
   char *serverIP;
   unsigned short puertoServidor;

   if (argc != 3)
   {
    printf("Uso: <IP servidor> <puerto>\n");
    exit(1);
   }

   // Carga el DLL Winsock 2.0
   if (WSAStartup(MAKEWORD(2, 0), &wsaData) != 0)
   {
        fprintf(stderr, "WSAStartup() failed");
        exit(1);
   }

   //Crear el socket UDP
   if ((sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
    AvisarError("Error al crear el socket");

   serverIP = argv[1];
   puertoServidor = atoi(argv[2]);

   //Obtener datos del servidor
   memset(&servidor,0,sizeof(servidor));
   servidor.sin_family = AF_INET;
   servidor.sin_port = htons(puertoServidor);
   servidor.sin_addr.S_un.S_addr = inet_addr(serverIP);
```

```
    length=sizeof(struct sockaddr_in);

    printf("Por favor ingrese el mensaje: ");
    memset(buffer,'\0',256);

    //Leer el mensaje de la entrada estandar (teclado)
    fgets(buffer, 255, stdin);

    //Enviar los datos al servidor
    n = sendto(sock, buffer, strlen(buffer), 0, (struct
                sockaddr *) &servidor, length);

    if (n < 0)
     AvisarError("Error durante Sendto");

    //Leer mensaje del servidor
    n = recvfrom(sock, buffer, 256, 0, (struct sockaddr *)
                  &cliente, &length);

    if (n < 0)
     AvisarError("Error durante recvfrom");

    //Mostrar mensaje devuelto por el servidor
    fprintf(stderr,"Recibi una confirmacion: ");
    puts(buffer);

    closesocket(sock);
    WSACleanup();
    return 0;
}
```

PRUEBAS DE FUNCIONAMIENTO Y VALIDACIÓN

Para verificar que las aplicaciones funcionen debe ejecutar el servidor en una ventana de la consola y el cliente en otra ventana, pasando a cada uno de ellos los parámetros requeridos.

EJECUCIÓN DEL SERVIDOR

```
./server 10000
```

EJECUCIÓN DEL CLIENTE EN LINUX

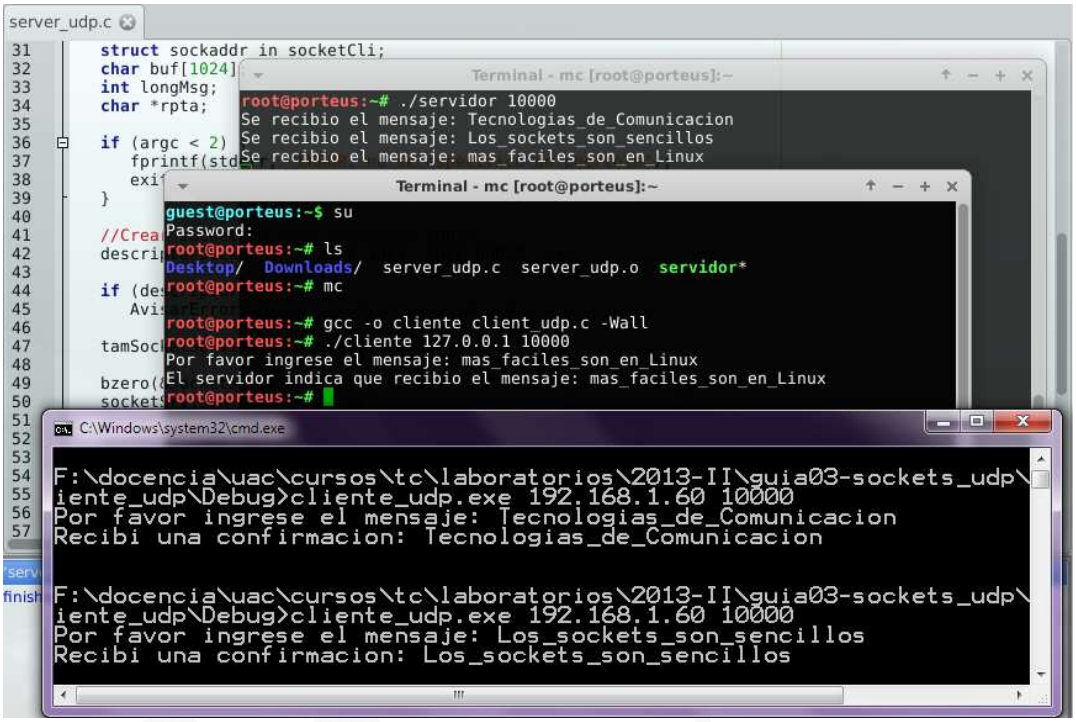
```
./cliente 127.0.0.1 10000 (en Linux)
```

EJECUCIÓN DEL CLIENTE EN WINDOWS

El cliente Windows puede ejecutarse desde la ventana de comandos.

```
cliente_udp 192.168.1.60 10000 (en Windows)
```

El resultado de las pruebas puede apreciarse en la siguiente captura de imagen:



Puede verificarse que el programa servidor se ejecuta como una aplicación Linux, así como uno de los clientes, que para el caso de las pruebas se ejecuta en la misma máquina que el servidor (de ahí la dirección 127.0.0.1). Por otra parte, el cliente Windows se ejecuta desde la ventana de comandos y en una máquina diferente (con dirección IP 192.168.1.2).

Conviene notar que tanto el cliente Linux como el cliente Windows, se dirigen al puerto 10 000, el cual es el número de puerto definido por el servidor UDP.

2. Escribir una aplicación Cliente/Servidor utilizando sockets TCP, en la que el cliente envía un mensaje de una palabra al servidor y este responde con el mismo mensaje, que debe ser mostrado por el cliente. Implemente el servidor en Linux y en Windows.

Solución

La aplicación se escribirá en dos partes: una que cumplirá la función de servidor (una aplicación por sistema operativo) y otra la de cliente. Para las pruebas, la aplicación cliente se ejecutará en un equipo y la aplicación servidor en otro equipo. Excepcionalmente se probarán ambos programas en un solo equipo.

CÓDIGO DEL SERVIDOR LINUX

```
// Nombre      : servidor
// Proposito   : Recibe un mensaje y lo envia de vuelta al
                 cliente
// Autor       : Edwin Carrasco (Adaptado de [1])
// FCreacion   : --
// FModific.   : 24/04/2009
// Compilacion : gcc -o Servidor Servidor.c

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

#define MAXPENDING 5    /* Nro maximo de conexiones */
#define BUFFSIZE 32

void AvisarError(char *mensaje)
{
    perror(mensaje);
    exit(1);
}

void AtenderCliente(int sock)
{
    // Variables
  char buffer[BUFFSIZE];
  int received = -1;
    // Recibir mensaje
    if ((received = recv(sock, buffer, BUFFSIZE, 0)) < 0)
    {
      AvisarError("No se pudo recibir bytes iniciales del
                   cliente");
    }

    // Envia bytes y verifica mas datos entrantes
```

```
    while (received > 0)
    {
      // Enviar de vuelta los datos recibidos
      if (send(sock, buffer, received, 0) != received)
      {
        AvisarError("No se pudo enviar bytes al cliente");
      }

      // Verificar mas datos
      if ((received = recv(sock, buffer, BUFFSIZE, 0)) < 0)
      {
        AvisarError("No se pudo recibir bytes adicionales del
                    cliente");
      }
    }
    close(sock);
}

int main(int argc, char *argv[])
{
  int serversock;
  int clientsock;
  struct sockaddr_in echoserver;
  struct sockaddr_in echoclient;

  if (argc != 2)
  {
    fprintf(stderr, "USO: servidor <puerto>\n");
    exit(1);
  }

  // Crear un socket TCP
  if ((serversock = socket(PF_INET, SOCK_STREAM,
  IPPROTO_TCP)) < 0)
  {
    AvisarError("No se pudo crear socket");
  }

  /* Construir la estructura sockaddr_in del servidor */

  // Limpiar la estructura
  memset(&echoserver, 0, sizeof(echoserver));

  // Protocolo Internet/IP
  echoserver.sin_family = AF_INET;

  // Direccion de ingreso
  echoserver.sin_addr.s_addr = htonl(INADDR_ANY);

  // Puerto en que escucha el servidor
  echoserver.sin_port = htons(atoi(argv[1]));

  // Enlazar el socket del servidor
```

```
  if (bind(serversock, (struct sockaddr *) &echoserver,
          sizeof(echoserver)) < 0)
  {
    AvisarError("No se pudo enlazar el socket de
          servidor");
  }

  // Escuchar en el socket del servidor
  if (listen(serversock, MAXPENDING) < 0)
  {
    AvisarError("No se pudo escuchar en el socket del
           servidor");
  }

  // Ejecutar hasta que se cancele
  while (1)
  {
    unsigned int clientlen = sizeof(echoclient);

    // Esperar conexion de cliente
    if ((clientsock=accept(serversock,(struct sockaddr *)
              &echoclient, &clientlen)) < 0 )
    {
      AvisarError("No se pudo aceptar conexion de
           cliente");
    }

    fprintf(stdout, "Cliente conectado: %s\n",
           inet_ntoa(echoclient.sin_addr));
    AtenderCliente(clientsock);
  }
}
```

CÓDIGO DEL CLIENTE LINUX

```
// Nombre      : Cliente
// Proposito   : Envia un mensaje y muestra el mensaje
                 devuelto por el servidor
// Autor       : Edwin Carrasco (Adaptado de [1])
// FCreacion   : --
// FModific.   : --
// Compilacion : gcc -o Cliente Cliente.c

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

#define BUFFSIZE 32
```

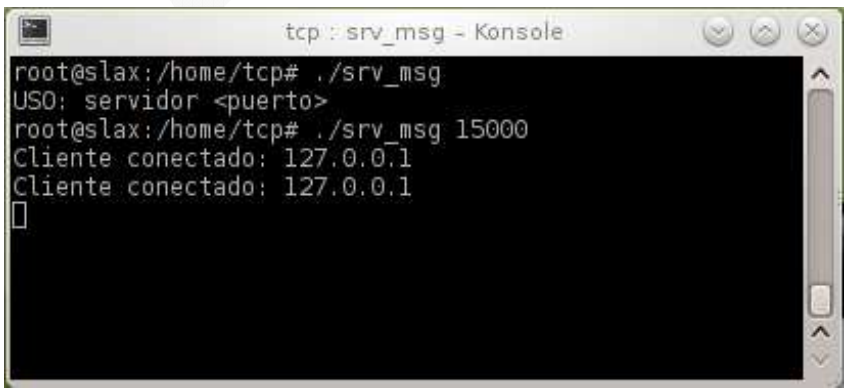
```
void AvisarError(char *mensaje)
{
  perror(mensaje);
  exit(1);
}


int main(int argc, char *argv[])
{
  // Variables
  int sock;
  struct sockaddr_in echoserver;
  char buffer[BUFFSIZE];
  unsigned int echolen;
  int received = 0;

  // Verificar argumentos necesarios para correr programa
  if (argc != 4)
  {
    fprintf(stderr, "USO: Cliente <ip de servidor>
      <mensaje>  <puerto>\n");
    exit(1);
  }

  // Crear el socket TCP
  if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP))< 0)
  {
    AvisarError("No se pudo crear socket");
  }


  /* Construir la estructura sockaddr_in del servidor */

  // Limpiar la estructura
  memset(&echoserver, 0, sizeof(echoserver));

  // Protocolo de Internet/IP
  echoserver.sin_family = AF_INET;

  // Direccion IP
  echoserver.sin_addr.s_addr = inet_addr(argv[1]);

  // Puerto del servidor
  echoserver.sin_port=htons(atoi(argv[3]));

  // Establecer la conexion
  if (connect(sock,(struct sockaddr *) &echoserver,
          sizeof(echoserver)) < 0 )
  {
    AvisarError("No se pudo conectar al servidor");
  }

  // Enviar en mensaje al servidor
  echolen = strlen(argv[2]);
```

```
    if (send(sock, argv[2], echolen, 0) != echolen)
    {
      AvisarError("Inconsistencia en el numero de bytes
                enviados");
    }

    // Recibir el mensaje de vuelta desde el servidor
    fprintf(stdout, "Recibido: ");

    while (received < echolen)
    {
      int bytes = 0;
      if ((bytes = recv(sock, buffer, BUFFSIZE-1, 0)) < 1)
      {
        AvisarError("No se pudo recibir bytes desde el
                servidor");
      }

      received += bytes;
      buffer[bytes]='\0'; // Cerciorarse que la cadena
                          termine en \0
      fprintf(stdout, buffer);
    }

    fprintf(stdout, "\n");
    close(sock);
    exit(0);
}
```

EJECUCIÓN DE LAS APLICACIONES

Para verificar que la aplicación funcione correctamente, ejecutamos el lado servidor y el lado cliente, respetando los argumentos con lo que se debe ejecutar cada uno de ellos.

Como puede verse en la siguiente figura, la ejecución del cliente requiere de un argumento: el número de puerto en el que el servidor esperará las peticiones de conexión de los clientes.

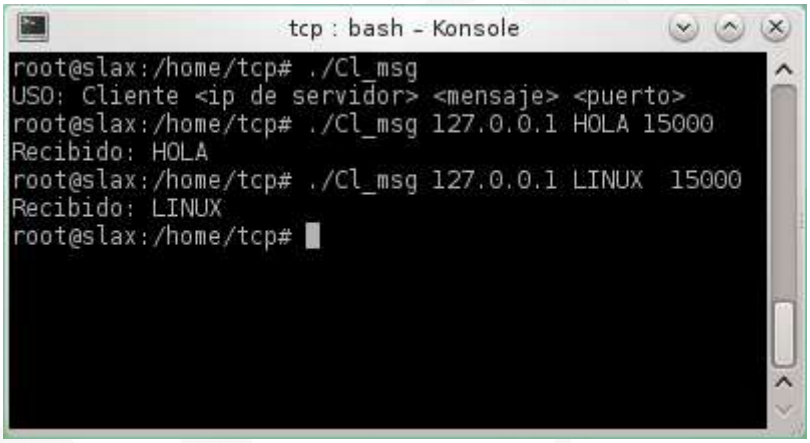


Por cada conexión entrante, el servidor muestra la dirección IP del cliente.

Por su parte el lado cliente requiere tres argumentos: la dirección IP del servidor, el mensaje que se enviará al servidor y el puerto en el cual el servidor escucha las peticiones de conexión.

Si no se conoce la dirección IP del servidor o el puerto en el cual éste escucha, no será posible que las aplicaciones se comuniquen.

Como puede verse en la figura, el cliente envía una palabra como mensaje y el servidor devuelve la misma. Al recibir el mensaje de vuelta, este es mostrado por la consola por el cliente:



CÓDIGO DEL SERVIDOR WINDOWS

```
// Nombre    : serv_tcp
// Proposito : Recibe un mensaje y lo envia de vuelta al
               cliente
// Autor     : Edwin Carrasco (Adaptado de [1])
// FCreacion : 08/07/2013
// FModific. : ---


#include <stdio.h>
#include <winsock.h>
#include <stdlib.h>

#define MAXPENDING 5    /* Numero maximo de conexiones */
#define BUFFSIZE 32

void AvisarError(char *mensaje)
{
    perror(mensaje);
    exit(1);
}
```

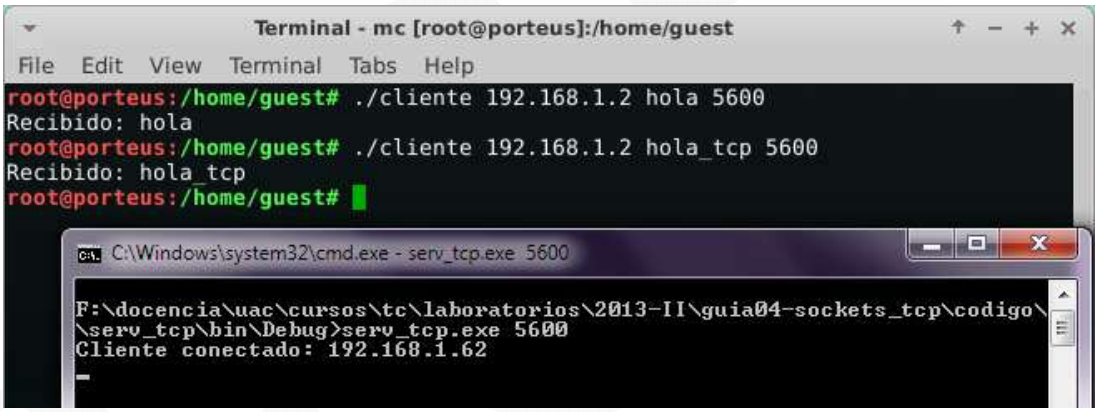
```
void AtenderCliente(int sock)
{
    //Variables
    char buffer[BUFFSIZE];
    int received = -1;

    // Recibir mensaje
    if ((received = recv(sock, buffer, BUFFSIZE, 0)) < 0)
    {
        AvisarError("No se pudo recibir bytes iniciales
                del cliente");
    }

    // Envia bytes y verifica mas datos entrantes
    while (received > 0)
    {
        // Enviar de vuelta los datos recibidos
        if (send(sock, buffer, received, 0) != received)
        {
          AvisarError("No se pudo enviar bytes al
                cliente");
        }

        // Verificar mas datos
        if ((received =recv(sock, buffer, BUFFSIZE, 0))<0)
        {
            AvisarError("No se pudo recibir bytes
                  adicionales del cliente");
        }
    }
    //Cerrar socket
    closesocket(sock);
}

int main(int argc, char *argv[])
{
    //Variables
    int serversock, clientsock;
    struct sockaddr_in echoserver, echoclient;
    WSADATA wsaData;

    //Verificar parametros requeridos
    if (argc != 2)
    {
        fprintf(stderr, "USO: servidor <puerto>\n");
        exit(1);
    }

    //Cargar Winsock 2.0 dll
    if (WSAStartup(MAKEWORD(2, 0), &wsaData) != 0)
    {
        fprintf(stderr, "WSAStartup() failed");
        exit(1);
    }
```

```
// Crear un socket TCP
if ((serversock = socket(PF_INET, SOCK_STREAM,
       IPPROTO_TCP)) < 0)
{
     AvisarError("No se pudo crear socket");
}


/* Construir la estructura sockaddr_in del servidor */

// Limpiar la estructura
memset(&echoserver, 0, sizeof(echoserver));

// Familia de protocolos: Internet/IP
echoserver.sin_family = AF_INET;


// Direccion de ingreso
echoserver.sin_addr.s_addr = htonl(INADDR_ANY);


// Puerto en el que escucha el servidor
echoserver.sin_port = htons(atoi(argv[1]));


// Enlazar el socket del servidor
if (bind(serversock, (struct sockaddr *) &echoserver,
         sizeof(echoserver)) < 0)
{
     AvisarError("No se pudo enlazar el socket de
           servidor");
}


// Escuchar en el socket del servidor
if (listen(serversock, MAXPENDING) < 0)
{
     AvisarError("No se pudo escuchar en el socket del
            servidor");
}

// Ejecutar hasta que se cancele
while (1)
{
     int clientlen = sizeof(echoclient);

     // Esperar conexion de cliente
     if ((clientsock = accept(serversock,(struct
     sockaddr *)&echoclient, &clientlen)) < 0 )
     {
       AvisarError("No se pudo aceptar conexion de
             cliente");
     }
```

```
            fprintf(stdout, "Cliente conectado: %s\n",
              inet_ntoa(echoclient.sin_addr));

            AtenderCliente(clientsock);
        }

        //Cerrar socket
        closesocket(serversock);
        WSACleanup();
        return 0;
}
```

EJECUCIÓN DE LAS APLICACIONES

Para verificar que la aplicación funcione correctamente, ejecutamos el lado servidor en Windows y el lado cliente en el equipo Linux, debiendo mostrarse un resultado similar al que se muestra en la siguiente figura:

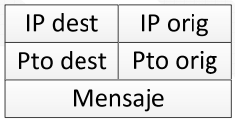


VII. TRABAJOS DE LABORATORIO.

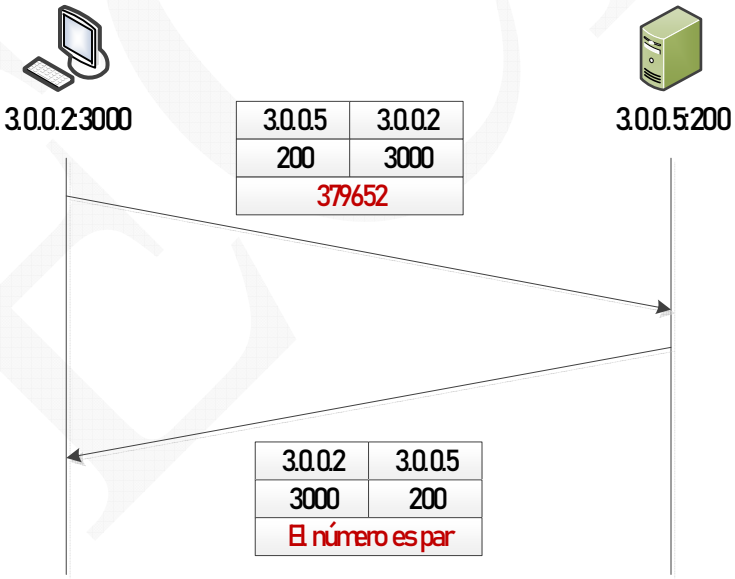
1. Escriba una aplicación cliente servidor, utilizando sockets UDP, en la que el cliente envíe un número entero al servidor. El servidor debe determinar si el número es primo y devolver un mensaje con la respuesta al cliente. El servidor debe implementarse en Linux y el cliente en Windows. El puerto en el que escucha el servidor debe pasarse como parámetro, tal como se muestra en los ejemplos.

2. Utilizando Wireshark, muestre los encabezados de los segmentos intercambiados entre las aplicaciones cliente y servidor UDP. Para cada caso indique las direcciones IP y los puertos utilizados. Represente la secuencia de segmentos intercambiados mediante un diagrama que exprese la temporalidad de cada evento. Respalde su diagrama con capturas de los paquetes en Wireshark.

   El formato de representación debe ser similar a los mostrados en el gráfico:



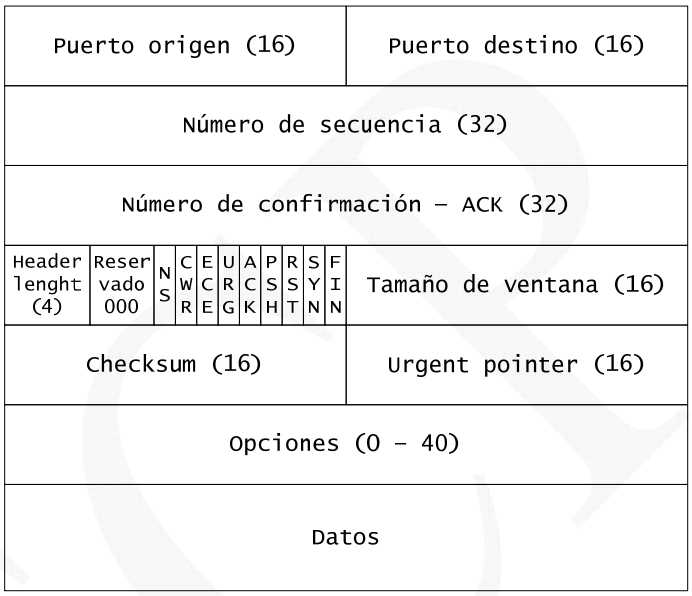
   El diagrama de intercambio debe ser similar al indicado:



3. Escriba una aplicación cliente servidor TCP, en la que el cliente envíe un mensaje de texto y el servidor devuelve el mensaje cifrado. Para el cifrado puede utilizar cualquier algoritmo. El servidor debe implementarse en Linux y el cliente en Windows. El puerto en el que escucha el servidor debe pasarse como parámetro, tal como se muestra en los ejemplos. Tanto el cliente como el servidor deben utilizar direcciones IPv6 solamente.

4. Utilizando Wireshark, muestre los encabezados de los segmentos intercambiados entre las aplicaciones cliente y servidor TCP. Para cada caso indique las direcciones IP y los puertos utilizados. Represente el establecimiento de la conexión, la secuencia de segmentos intercambiados y el cierre de conexión mediante un diagrama que exprese la temporalidad de cada evento. Respalde su diagrama con capturas de los paquetes en Wireshark.

   El formato de representación de los segmentos TCP debe ser similar a los mostrados en el gráfico:



   El diagrama de intercambio debe ser similar al utilizado en el ejercicio de UDP, al cual se debe agregar las fases de establecimiento de conexión y cierre de conexión.

VIII. CRITERIO DE EVALUACIÓN

La evaluación de las actividades realizadas en la presente guía de práctica se hará en función de la siguiente tabla:

ACTIVIDAD	PROCEDIMENTAL		
	SESIÓN 01	SESIÓN 02	SESIÓN 03
Resolución del ejercicio propuesto 01	--	12	--
Resolución del ejercicio propuesto 02	--	08	--
Resolución del ejercicio propuesto 03	--	--	14
Resolución del ejercicio propuesto 04	--	--	06
NOTA	**--**	**20**	**20**

IX. BIBLIOGRAFIA

1. Comer, D. "*Internetworking With TCP/IP Volumen 1*". Ed. Prentice Hall 4ed
2. Comer, D. "*Internetworking With TCP/IP Volumen 2*". Ed. Prentice Hall 3ed
3. Hall, B. "*Beej´s Guide to Network Programming*" http://www.beej.us/guide/bgnet/
4. Sockets Tutorial. https://www.linuxhowtos.org/C_C++/socket.htm
5. López, Novo; "*Protocolos de Internet*" Ed. AlfaOmega
6. Microsoft. Directorios de VC++ (Página De Propiedades) http://msdn.microsoft.com/es-es/library/vstudio/ee855621(v=vs.110).aspx
7. Microsoft. "*Winsock Reference*". http://msdn.microsoft.com/en-us/library/windows/desktop/ms741416%28v=vs.85%29.aspx (F. V. 24/06/2013)
8. Tenouk. "*C, Winsok2 and Visual Studio 2008 (VC++) Professional Edition:. The How-to Build Guide Part 1*" http://www.tenouk.com/Winsock/visualstudio2008ncnwinsock2.html 2012.
9. Tenouk. "*C, Winsok2 and Visual Studio 2008 (VC++) Professional Edition:. The How-to Build Guide Part 2*" http://www.tenouk.com/Winsock/visualstudio2008ncnwinsock2a.html 2012
10. winsocketdotnetworkprogramming.com. "*Tutorials on 'Advanced' winsock 2 Network Programming*" http://www.winsocketdotnetworkprogramming.com/winsock2programming/