AT

Implementación cliente servidor mediante sockets

Client-server implementation using sockets

Héctor Julio Fúquene Ardila*

Fecha de recepción: septiembre 30 de 2011
Fecha de aceptación: noviembre 4 de 2011

Resumen

Este artículo pretende hacer visibles los requerimientos tanto físicos, lógicos y funcionales necesarios para establecer una comunicación entre dos computadores utilizando sockets; se plantean los diferentes protocolos mediante los cuales se puede realizar su implementación; además, se presentan las principales características técnicas a tener en cuenta y se realiza la implementación mediante el uso del lenguaje de programación como C++.

Palabra clave

Programa cliente, programa servidor, comunicaciones, puerto lógico, protocolo, dirección IP.

Abstract

This article aims to make visible the requirements of both physical, logical and functional requirements for communication between two computers using sockets, raised different protocols by which implementation can be done, also presents the main technical features to consider and implementation is done using the programming language C + +.

Keywords

Client (customer) program, server program, communications, logic port, protocol, IP address.

* Ingeniero de sistemas, Magister en teleinformática, miembro del grupo de investigación IAFT, docente de palnta de la Universidad Distrital Francisco José de Caldas. hfuquene@udistrital.edu.co

Introducción

La comunicación entre entidades abstractas (PCs) se logra utilizando diferentes técnicas, una de ellas es utilizando sockets, que consiste en configurar una red cliente servidor para establecer el flujo de información entre transmisor y receptor.

Es posible utilizar un protocolo orientado a conexión como TCP (Transfer Control Protocol) o uno no orientado a conexión como UDP (User Datagram Protocol), la diferencia entre los dos es la fiabilidad que se puede garantizar con un protocolo que garantice la conexión con el extremo receptor, es decir que sea orientado a conexión.

Para que dos aplicaciones puedan intercambiar información entre sí se necesita:

- Un protocolo de comunicación común a nivel de red y a nivel de transporte.
- Una dirección del protocolo de red que identifique a cada uno de los computadores.
- Un número de puerto que identifique la aplicación dentro del computador.

Los socket se utilizan para poder enviar órdenes a un servidor que está atendiendo nuestras peticiones. Por lo tanto, un socket quedaría definido por una dirección IP (La del equipo que actúa como servidor), un protocolo y un número de puerto (El utilizado para acceder a un servicio determinado).

A su vez, los sockets están basados en el protocolo de comunicaciones utilizado en Internet: TCP (Transmisión Control Protocol).

La función del protocolo TCP es, ni más ni menos que la de traer y llevar información desde un servidor a un cliente y viceversa utilizando los diferentes protocolos utilizados por cada servicio.

¿Qué son los sockets?

Podemos encontrar diferentes definiciones, algunas de ellas son:

Los sockets son un método de comunicación entre un programa de cliente y uno de servidor a través de una red. Un socket se define como "el extremo de una conexión".

Un **socket**, es un método para la comunicación entre un programa del cliente y un programa del servidor en una red. Un socket se define como el punto final en una conexión. Los sockets se crean y se utilizan con un sistema de peticiones o de *llamadas de función* a veces llamados interfaz de programación de aplicación de sockets (API, Application Programming Interface).

Un **socket** es también una dirección de Internet, combinando una dirección IP (la dirección numérica única de cuatro octetos que identifica a un computador particular en Internet) y un número de puerto (el número que identifica una aplicación de Internet particular)

Un socket es un punto final de un enlace de comunicación de dos vías entre dos programas que se ejecutan a través de la red.

¿Cómo se establece la comunicación con sockets?

El cliente y el servidor deben ponerse de acuerdo sobre el protocolo que utilizarán.

Hay dos tipos de socket:

Orientado a conexión

Establece un camino virtual entre servidor y cliente, fiable, sin pérdidas de información ni duplicados, la información llega en el mismo orden que se envía. El cliente abre una sesión en el servidor y este guarda un estado del cliente.

Orientado a no conexión

Envío de datagramas de tamaño fijo. No es fiable, puede haber pérdidas de información y duplicados, y la información puede llegar en distinto orden del que se envía. No se guarda ningún estado del cliente en el servidor, por ello, es más tolerante a fallos del sistema. Los*sockets* no son más que puntos o mecanismos de comunicación entre procesos que permiten que un proceso hable (emita o reciba información) con otro proceso incluso estando estos procesos en distintas máquinas. Esta característica de interconectividad entre máquinas hace que el concepto de socket nos sirva de gran utilidad.

Un socket es al sistema de comunicación entre computadores, lo que un buzón o un teléfono es al sistema de comunicación entre personas: un punto de comunicación entre dos agentes (procesos o personas respectivamente) por el cual se puede emitir o recibir información.

La arquitectura cliente-servidor

Es un modelo de aplicación distribuida en el que las tareas se reparten entre los proveedores de recursos o servicios, llamados servidores, y los demandantes, llamados clientes. Un programa cliente realiza peticiones a otro programa, el servidor, que le da respuesta. Esta idea también se puede aplicar a programas que se ejecutan sobre una sola máquina, aunque es más ventajosa en un sistema operativo multiusuario distribuido a través de una red de computadores.

En esta arquitectura la capacidad de proceso está repartida entre los clientes y los servidores. La separación entre cliente y servidor es una separación de tipo lógico, donde el servidor no se ejecuta necesariamente sobre una sola máquina ni es necesariamente un sólo programa. Los tipos específicos de servidores incluyen los servidores webs, los servidores de archivo, los servidores del correo, entre otros. Mientras que sus propósitos varían de unos servicios a otros, la arquitectura básica seguirá siendo la misma.

La comunicación entre procesos a través de sockets se basa en la filosofía cliente-servidor: un proceso en esta comunicación actuará de proceso servidor creando un socket cuyo nombre conocerá el proceso cliente, el cual podrá "hablar" con el proceso servidor a través de la conexión con dicho socket nombrado.

El proceso crea un socket sin nombre cuyo valor de vuelta es un descriptor sobre el que se leerá o escribirá, permitiéndose una comunicación bidireccional, característica propia de los sockets. El mecanismo de comunicación vía sockets tiene los siguientes pasos:

- El proceso servidor crea un socket con nombre y espera la conexión.
- El proceso cliente crea un socket sin nombre.
- El proceso cliente realiza una petición de conexión al socket servidor.
- El cliente realiza la conexión a través de su socket mientras el proceso servidor mantiene el socket servidor original con nombre.

Es muy común en este tipo de comunicación lanzar un proceso hijo, una vez realizada la conexión, que se ocupe del intercambio de información con el proceso cliente mientras el proceso padre servidor sigue aceptando conexiones. Para eliminar esta característica se cerrará el descriptor del socket servidor con nombre en cuanto realice una conexión con un proceso socket cliente.

El Servidor

A partir de este punto comenzamos con lo que es la programación en C++ de los sockets. Se debe poseer conocimientos de C++, de esta forma será más fácil el atender el procedimiento a seguir.

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

Realizar la apertura de un socket, mediante la función **socket().** Esta función devuelve un descriptor de archivo 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. 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 archivo. Obviamente, tanto cliente como servidor deben saber qué datos esperan recibir, qué datos deben enviar y en qué formato.

Cierre de la comunicación y del socket, por medio de la función **close(),** que es la misma que sirve para cerrar un archivo.

El Cliente

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

Realizar la 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().**

Como se puede apreciar, el procedimiento en el cliente es mucho más sencillo que el servidor, más sin embargo, se debe como mínimo garantizar en los dos extremos, un paso de establecimiento de la comunicación, uno de transferencia de información y uno mediante el cual se libera la comunicación.

Tipos de sockets

Todo socket viene definido por dos características fundamentales: El tipo del socket y el dominio del socket.

El tipo del socket, que indica la naturaleza del mismo, el tipo de comunicación que puede generarse entre los sockets.

El dominio del socket especifica el conjunto de sockets que pueden establecer una comunicación con el mismo.

Los sockets definen las propiedades de las comunicaciones en las que se ven envueltos, esto es, el tipo de comunicación que se puede dar entre cliente y servidor. Estas pueden ser:

Fiabilidad de transmisión.
Mantenimiento del orden de los datos.
No duplicación de los datos.
El "Modo Conectado" en la comunicación.
Envío de mensajes urgentes.

Y cómo Logran "hablar" los computadores.

A continuación se describe detalladamente cómo se establece la comunicación entre dos procesos residentes en un mismo PC o en máquinas diferentes. Se describe las funciones utilizadas, haciendo énfasis en los parámetros utilizados y en el orden lógico de invocación. Inicialmente, se hace la descripción del proceso realizado en el servidor y posteriormente lo que ocurre en el cliente.

A continuación se presenta el código en C++ para Windows, utilizando ambiente gráfico. En Windows es indispensable utilizar la librería winsock2.h y utilizar el software Dev C++ generando un proyecto.

Servidor.

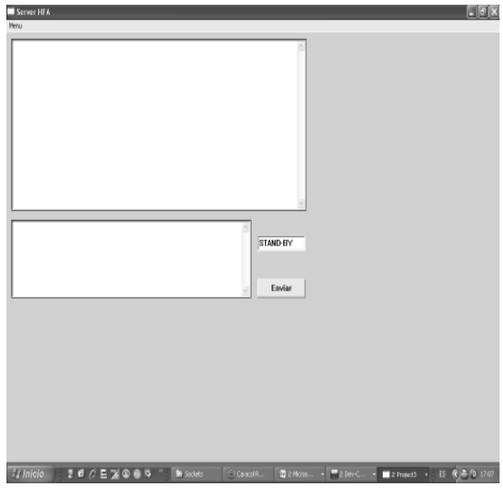
```
//nombrar este código con el nombre server.cpp
#include <windows.h>
#include <iostream>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdlib.h>
#include <stdio.h>
#define DEFAULT_BUFLEN 512
#define DEFAULT_PORT "9999"
#define SERVER_ICON 100
#define CM_INICIAR 101
#define CM_SALIR 102
#define CM_CERRARCONEXION 103
#define CM_ENVIARDATA 104
LRESULT CALLBACK WindowProcedure (HWND, UINT, WPARAM, LPARAM);
char szClassName[ ] = "Server";
void InsertarMenu(HWND);
char enviardata(HWND, char*);
WSADATA wsa;
SOCKET sock;
struct sockaddr_in local;
int len=0;
char Buffer[500000];
int WINAPI WinMain (HINSTANCE hThisInstance, HINSTANCE hPrevInstance,
        LPSTR lpszArgument, int nFun
    sterStil)
{
    HWND hwnd;
    MSG messages;
    WNDCLASSEX wincl;
    wincl.hInstance = hThisInstance;
    wincl.lpszClassName = szClassName;
    wincl.lpfnWndProc = WindowProcedure;
    wincl.style = CS_DBLCLKS;
    wincl.cbSize = sizeof (WNDCLASSEX);
    wincl.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    wincl.hIconSm = LoadIcon (NULL, IDI_APPLICATION);
    wincl.hCursor = LoadCursor (NULL, IDC_ARROW);
    wincl.lpszMenuName = NULL;
    wincl.cbClsExtra = 0;
     wincl.cbWndExtra = 0;
    wincl.hbrBackground = (HBRUSH) COLOR_BACKGROUND;
    if (!RegisterClassEx (&wincl))
        return 0;
    hwnd = CreateWindowEx (0, szClass Name,
    "Server",WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT,       CW_USE
        DEFAULT, 640, 480, HWND_DESKTOP,  NULL,
       hThisInstance,
       NULL
       );
```

```
   InsertarMenu(hwnd);
 ShowWindow (hwnd, nFunsterStil);
 while (GetMessage (&messages, NULL, 0, 0))
 {
 TranslateMessage(&messages);
 DispatchMessage(&messages);
 }
 return messages.wParam;
}
struct thread1
{
    HWND hwnd1;
    HWND hwnd2;
    HWND hwnd3;
};
thread1 obj1;
UINT ThreadProc1(LPVOID lpvoid);
UINT ThreadProc1(LPVOID lpvoid)
{
    thread1 *temp = (thread1*)lpvoid;
    HWND hwnd= temp->hwnd1;
    HWND hwndRECIBIDO = temp->hwnd2;
    HWND hwndSTATUS = temp->hwnd3;
    HDC hdc= GetDC(hwnd);
  WSAStartup(MAKEWORD(2,0),&wsa);
                sock=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
  local.sin_family = AF_INET;
  local.sin_addr.s_addr = INADDR_ANY;
  local.sin_port = htons(9999);
     if (bind(sock, (SOCKADDR*) &local, sizeof(local))==-1)
  {
   SendMessage(hwndSTATUS, WM_SETTEXT, 0, (long int) "ERROR" );
  }
  if (listen(sock,1)==-1)
  {
   SendMessage(hwndSTATUS, WM_SETTEXT, 0, (long int) "ERROR" );
  }
  else
  {
   SendMessage(hwndSTATUS, WM_SETTEXT, 0, (long int) "INICIADO" );
  }
  len=sizeof(struct sockaddr);
    sock=accept(sock,(sockaddr*)&local,&len);
   SendMessage(hwndSTATUS, WM_SETTEXT, 0, (long int) "CONECTADO" );
  while (len!=0)
  {
  len=recv(sock,Buffer,499999,0);
  if (len>0)
  {
   Buffer[len]=0;
  SendMessage(hwndRECIBIDO, WM_SETTEXT, 0, (long int) Buffer );
  }
   Sleep(1);
  }
}
LRESULT CALLBACK WindowProcedure (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
static HINSTANCE hInstance;
char textoAENVIAR[500000];
HANDLE hThrd1,hThrd2;
 switch (message)
 {   case WM_CREATE:
    static HWND hwndENVIARDATA = CreateWindow
  (  "button", "Enviar",
   WS_CHILD|WS_VISIBLE|BS_CENTER,
    520, 395, 100, 30, hwnd, (HMENU)CM_ENVIARDATA, hInstance, NULL
  );
                        static HWND hwndRECIBIDO=CreateWindowEx
  (  WS_EX_CLIENTEDGE, "EDIT","",
   WS_CHILD | WS_VSCROLL | ES_MUL-
```

```
TILINE | WS_VISIBLE | WS_BORDER | WS_TABSTOP,
   10, 10, 612, 275, hwnd,  NULL, hInstance, NULL
  );
                        static HWND hwndENVIAR=CreateWindowEx
  (  WS_EX_CLIENTEDGE,  "EDIT","",
   WS_CHILD | WS_VSCROLL | ES_MULTILINE | WS_VISIBLE | WS_BORDER | WS_TABSTOP,
   10, 300, 497, 125, hwnd,  NULL, hInstance, NULL  );
                        static HWND hwndSTATUS=CreateWindowEx
  (  WS_EX_CLIENTEDGE,
   "EDIT","STAND-BY",
    WS_CHILD | WS_VISIBLE | WS_BORDER | WS_TABSTOP,
   520, 325, 100, 25, hwnd,  NULL, hInstance, NULL
  );
  break;
    case WM_COMMAND:
   switch(LOWORD(wParam)) {
   case CM_INICIAR:
            // Run the first Thread
            obj1.hwnd1 = hwnd;
            obj1.hwnd2 = hwndRECIBIDO;
            obj1.hwnd3 = hwndSTATUS;
hThrd1 = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE) ThreadProc1,
            (LPVOID)&obj1,
            CREATE_SUSPENDED,
                    NULL);
            ResumeThread(hThrd1);

  SetThreadPriority(hThrd1,THREAD_PRIORITY_HIGHEST);
   break;
    case CM_SALIR:
   closesocket(sock);
   WSACleanup();
   PostQuitMessage(0);
   break;
    case CM_CERRARCONEXION:
   closesocket(sock);
   WSACleanup();
   break;
    case CM_ENVIARDATA:
   GetWindowText(hwndENVIAR, textoAENVIAR, 500000);
      enviardata(hwndENVIAR, textoAENVIAR);
   break;
   }
   break;
 case WM_DESTROY:
  closesocket(sock);
  WSACleanup();
  PostQuitMessage (0);
  break;
 default:
  return DefWindowProc (hwnd, message, wParam, lParam);
 }
 return 0;
}
char enviardata(HWND hwndENVIAR, char *entrada)
{
send(sock,entrada,strlen(entrada),0);
SendMessage(hwndENVIAR, WM_SETTEXT, 0, (long int) " " );
}
void InsertarMenu(HWND hwnd)
{
HMENU hMenu1, hMenu2;
hMenu1 = CreateMenu();
hMenu2 = CreateMenu();
AppendMenu(hMenu2, MF_STRING, CM_INICIAR, "&Iniciar Server");
AppendMenu(hMenu2, MF_STRING, CM_CERRARCONEXION, "&Cerrar Conexion");
AppendMenu(hMenu2, MF_SEPARATOR, 0, NULL);
```

```
AppendMenu(hMenu2, MF_STRING, CM_
SALIR, "&Salir");
AppendMenu(hMenu1, MF_STRING | MF_
POPUP, (UINT)hMenu2, "&Menu");
SetMenu (hwnd, hMenu1);
}
```



Como se están ejecutando las dos aplicaciones en la misma máquina (cliente y servidor) se debe iniciar de forma independiente. En la gráfica anterior aparece la interfaz inicial del servidor.

El programa cliente se presenta a continuación:

```
//nombrar este código con el nombre cliente.cpp
#include <windows.h>
#include <iostream>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdlib.h>
#include <stdio.h>
#define DEFAULT_BUFLEN 512
#define DEFAULT_PORT "9999"
#define CLIENT_ICON 100
#define CM_CONECTAR 101
#define CM_SALIR 102
#define CM_CERRARCONEXION 103
#define CM_ENVIARDATA 104
LRESULT CALLBACK WindowProcedure (HWND, UINT, WPARAM, LPARAM);
char szClassName[ ] = "Client";
void InsertarMenu(HWND);
char enviardata(HWND, char*);
WSADATA wsa;
SOCKET sock;
int len;
char Buffer[500000];
struct hostent *host;
struct sockaddr_in direc;
int conex;
int WINAPI WinMain (HINSTANCE hThisInstance,
    HINSTANCE hPrevInstance,
    LPSTR lpszArgument,
    int nFunsterStil)
{
 HWND hwnd;
 MSG messages;
 WNDCLASSEX wincl;
 wincl.hInstance = hThisInstance;
 wincl.lpszClassName = szClassName;
 wincl.lpfnWndProc = WindowProcedure;
 wincl.style = CS_DBLCLKS;
 wincl.cbSize = sizeof (WNDCLASSEX);
 wincl.hIcon = LoadIcon (NULL, IDI_APPLICATION);
  wincl.hIconSm = LoadIcon (NULL, IDI_APPLICATION);
 wincl.hCursor = LoadCursor (NULL, IDC_ARROW);
 wincl.lpszMenuName = NULL;
 wincl.cbClsExtra = 0;
 wincl.cbWndExtra = 0;
  wincl.hbrBackground = (HBRUSH) COLOR_BACKGROUND;
 if (!RegisterClassEx (&wincl))
  return 0;
 hwnd = CreateWindowEx (0, szClassName,
 "Client",
```

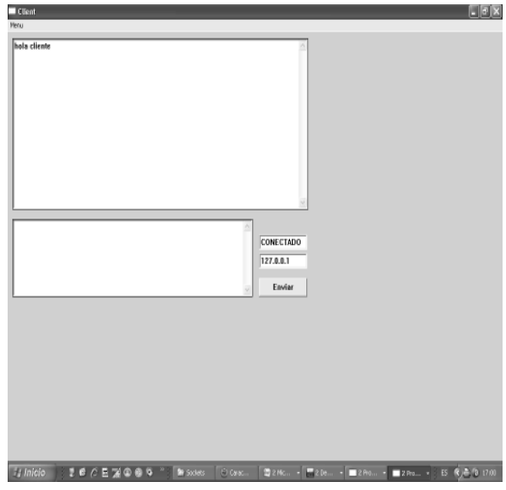
```
    WS_OVERLAPPEDWINDOW,
   CW_USEDEFAULT, CW_USEDEFAULT,
640, 480,      HWND_DESKTOP,
    LoadMenu(hThisInstance, "Menu"),
    hThisInstance,
    NULL
    );
 InsertarMenu(hwnd);
 ShowWindow (hwnd, nFunsterStil);
 while (GetMessage (&messages, NULL, 0,
0))
 {
  TranslateMessage(&messages);
  DispatchMessage(&messages);
 }
 return messages.wParam;
}
struct thread1
{
     HWND hwnd1;
     HWND hwnd2;
     HWND hwnd3;
     HWND hwnd4;
};
thread1 obj1;
UINT ThreadProc1(LPVOID lpvoid);
UINT ThreadProc1(LPVOID lpvoid)
{
     thread1 *temp = (thread1*)lpvoid;
     HWND hwnd= temp->hwnd1;
     HWND hwndSTATUS = temp->hwnd2;
     HWND hwndIP = temp->hwnd3;
     HWND hwndRECIBIDO = temp-
>hwnd4;
     HDC hdc= GetDC(hwnd);
  WSAStartup(MAKEWORD(2,2),&wsa);
   //resuelve el nombre de dominio local-
host, esto se resolverá a 127.0.0.1
  //host=gethostbyname("127.0.0.1");
  char ip[30];
  GetWindowText(hwndIP, ip, 30);
  host=gethostbyname(ip);
  //se crea el socket
                sock=socket(AF_INET,SOCK_
STREAM,IPPROTO_TCP);
  if (sock==-1)
  {
   SendMessage(hwndSTATUS, WM_SET-
TEXT, 0, (long int) "ERROR" );;
  }
   // Se define la dirección a conectar que
hemos recibido desde el gethostbyname
  //y se decide que el puerto al que deberá
conectar es el 9999 con el protocolo ipv4
  direc.sin_family=AF_INET;
  direc.sin_port=htons(9999);
   direc.sin_addr = *((struct in_addr *)host-
>h_addr);
  memset(direc.sin_zero,0,8);
  //Intentamos establecer la conexión
    conex=connect(sock,(sockaddr *)&direc,
sizeof(sockaddr));
  if (conex==-1)  //si no se ha podido conec-
tar porque no se ha encontrado el host o no
    //está el puerto abierto
  {
   SendMessage(hwndSTATUS, WM_SET-
TEXT, 0, (long int) "ERROR" );
  }
  else
  {
   SendMessage(hwndSTATUS, WM_SET-
TEXT, 0, (long int) "CONECTADO" );
}
 len=sizeof(struct sockaddr);
  while (len!=0) //mientras estemos conec-
tados con el otro pc
  {
   len=recv(sock,Buffer,499999,0);  //recibi-
mos los datos
  if (len>0)  //si seguimos conectados
  {
  Buffer[len]=0; //le ponemos el final de ca-
dena
  SendMessage(hwndRECIBIDO, WM_SET-
TEXT, 0, (long int) Buffer );
```

```
  }
  Sleep(1);
  }
}
LRESULT CALLBACK WindowProcedure (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
static HINSTANCE hInstance;
char textoAENVIAR[500000];
HANDLE hThrd1,hThrd2;
 switch (message)
 {
  case WM_CREATE:
      static HWND hwndENVIARDATA = CreateWindow
   (  "button", "Enviar",
    WS_CHILD|WS_VISIBLE|BS_CENTER,
     520, 395, 100, 30, hwnd, (HMENU)CM_ENVIARDATA, hInstance, NULL
   );
                        static HWND hwndRECIBIDO=CreateWindowEx
   (  WS_EX_CLIENTEDGE, "EDIT","",
WS_CHILD | WS_VSCROLL | ES_MULTILINE | WS_VISIBLE | WS_BORDER | WS_TABSTOP,
    10, 10, 612, 275, hwnd,  NULL, hInstance, NULL
   );
                        static HWND hwndENVIAR=CreateWindowEx
   (  WS_EX_CLIENTEDGE, "EDIT","",
    WS_CHILD | WS_VSCROLL | ES_MULTILINE | WS_VISIBLE | WS_BORDER | WS_TABSTOP,
   10, 300, 497, 125, hwnd,  NULL, hInstance, NULL
   );
   static HWND hwndIP=CreateWindowEx
   (  WS_EX_CLIENTEDGE,
    "EDIT","127.0.0.1",
     WS_CHILD | WS_VISIBLE | WS_BORDER | WS_TABSTOP,
   520, 355, 100, 25, hwnd,  NULL, hInstance, NULL
   );
                        static HWND hwndSTATUS=CreateWindowEx
   (  WS_EX_CLIENTEDGE,
    "EDIT","STAND-BY",
     WS_CHILD | WS_VISIBLE | WS_BORDER | WS_TABSTOP,
   520, 325, 100, 25, hwnd,  NULL, hInstance, NULL
   );
   break;
    case WM_COMMAND:
     switch(LOWORD(wParam)) {
     case CM_CONECTAR:
              obj1.hwnd1 = hwnd;
              obj1.hwnd2 = hwndSTATUS;
              obj1.hwnd3 = hwndIP;
              obj1.hwnd4 = hwndRECIBIDO;
           hThrd1 = CreateThread(NULL, 0,
                        (LPTHREAD_START_ROUTINE) ThreadProc1,
           (LPVOID)&obj1, // param to thread func
           CREATE_SUSPENDED, // creation flag
           NULL);   //
               ResumeThread(hThrd1);

   SetThreadPriority(hThrd1,THREAD_PRIORITY_HIGHEST);
    break;
     case CM_SALIR:
     closesocket(sock);
     WSACleanup();
    PostQuitMessage(0); // envía un mensaje WM_QUIT a la cola de mensajes
     break;
     case CM_CERRARCONEXION:
     closesocket(sock);
     WSACleanup();
     break;
```

```
    case CM_ENVIARDATA:
     GetWindowText(hwndENVIAR, textoA-
ENVIAR, 500000);
       enviardata(hwndENVIAR, textoAEN-
VIAR);
       //SendMessage(hwndENVIAR, WM_
SETTEXT, 0, (long int) textoAENVIAR );
    break;
    }
    break;
 case WM_DESTROY:
  PostQuitMessage (0);
  break;
 default:
   return DefWindowProc (hwnd, message,
wParam, lParam);
 }
 return 0;
}
char enviardata(HWND hwndENVIAR,
char *entrada) //FUNCION DE ENVIAR
{
send(sock,entrada,strlen(entrada),0);
SendMessage(hwndENVIAR, WM_SET-
TEXT, 0, (long int) " " );
}
void InsertarMenu(HWND hwnd)
{
HMENU hMenu1, hMenu2;
hMenu1 = CreateMenu();
hMenu2 = CreateMenu();
AppendMenu(hMenu2, MF_STRING, CM_
CONECTAR, "&Conectar");
AppendMenu(hMenu2, MF_STRING, CM_
CERRARCONEXION, "&Cerrar Conex-
ion");
AppendMenu(hMenu2, MF_SEPARATOR,
0, NULL);
AppendMenu(hMenu2, MF_STRING, CM_
SALIR, "&Salir");
AppendMenu(hMenu1, MF_STRING | MF_
POPUP, (UINT)hMenu2, "&Menu");
SetMenu (hwnd, hMenu1);
}
```

En la gráfica siguiente visualizamos el primer pantallazo que presenta el programa cliente. El programa se puede correr en una sola máquina con la dirección 127.0.0.1 o en una diferente para lo cual se debe introducir la dirección IP correspondiente al servidor.



En este tipo de aplicaciones, podemos direccionar datos en ambas direcciones, es decir el cliente y el servidor pueden hacer las veces de transmisor y de receptor. En la pantalla anterior visualizamos el mensaje recibido, lo que transmitió la aplicación servidor.

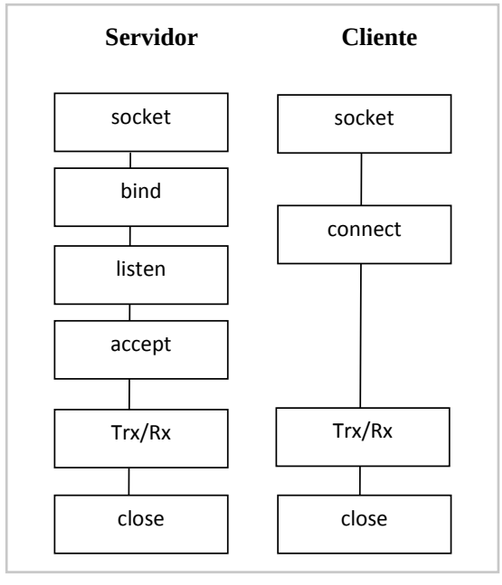
Conclusiones.

La implementación va a depender siempre del sistema operativo que se esté utilizando. Si el sistema es Linux se debe utilizar librerías específicas, con las cuales realizar el proceso. Funcionalmente el procedimiento no sufre cambios significativos.

La mayoría de lenguajes de programación permiten hacer este tipo de implementaciones, pues poseen las librerías y funciones con las cuales trabajar.

En la actualidad existe una gran variedad de aplicaciones que utilizan este tipo de arquitectura (cliente/servidor) para interactuar; pues se logran optimizar los recursos y distribuir los procesos.

El proceso realizado en el cliente lo podemos considerar más sencillo que el realizado en el servidor según se esquematiza en el siguiente diagrama.



El programar la WIN-API de Windows nos facilita enormemente el trabajo; nos da la posibilidad del uso de interfaces estándar y permite una mayor interacción entre el usuario y la aplicación; identificando plenamente los dos extremos de la transmisión, el cliente y el servidor.

Trabajos futuros

El programa se puede mejorar para que sea un poco más didáctico para el usuario final, de tal forma que se pueda identificar el tipo de transmisión, el modo de transmisión y las velocidades de transmisión utilizadas.

Se puede mejorar el programa para que permita la transmisión de archivos, pues actualmente se transmiten los caracteres de forma individual.

Referencias Bibliográficas

DEITEL, Harvey M. Cómo programar en C/C++ y Java. México: Prentice Hall. 2004

DONOHOO Michael J., CALVERT, L Kenneth. TCP/IP sockets in C bundle. Editorial PrenticeHall. 2009

DONAHOO, Michael. TCP/IP Sockets in C: Practical guide for programmers. Estados Unidos: Morgan Kaufmann. 2009

FOROUNZAN Behrouz A. Transmisión de Datos y Redes de Comunicaciones. McGrawHill. 2009

KENDALL, Kenneth. Análisis y Diseño de Sistemas. México: Prentice Hall. 2005

MÁRQUEZ García Manuel, Unix, Programación avanzada, editorial Ra-ma. 2005

RUMBAUGH James, Modelado y diseño orientado a objetos. Metodología OMT, Editorial Prentice Hall. 2006

STALLING, William. Comunicaciones y Redes de Computadores. Editorial Prentice-Hall. 2000

Fuentes Electrónicas

http://www.winprog.org/tutorial/es/index.html

http://es.tldp.org/Universitarios/seminario-2-sockets.html

http://www.chuidiang.com/varios/Libros/libros.php

http://www.arrakis.es/~dmrq/beej/index.html

http://www.mitecnologico.com/Main/ComunicacionClienteServidorSockets