6. Red

6. Red
6. Red
6. Red

6.1 Conceptos

La transmisión de datos a través de una red se estandarizó hace tiempo a nivel de software y hardware. Actualmente se trabaja con el llamado *modelo de capas*. Básicamente, este modelo pretende aislar los diferentes niveles de complejidad de datos, de forma que la sustitución de una capa no afecte al trabajo en otra. Podemos ver hoy en día que es posible, por ejemplo, conectarse a Internet a través de un módem, una línea ADSL o cable ethernet en una red local y, sin embargo, el hecho de tener aparatos tan distintos en funcionamiento no obliga a disponer de un navegador o un cliente de correo diferentes para cada situación. Esto se debe al modelo de capas. El navegador sólo entiende de protocolos de alto nivel, como HTTP o FTP, transfiere la información a un nivel más bajo y, finalmente, el módem convencional, ADSL o la tarjeta de red se ocupan de transportar esos datos según proceda.

Para no entrar en complejidades, ya que es tarea de un libro sobre redes, diremos que hay un par de niveles: los más bajos, conformados por el hardware (módems, tarjetas de red...) y los drivers o módulos del núcleo, que controlan y trocean la información para emitirla o recibirla por dicho hardware; y unos niveles por encima, que son los que más nos interesan a la hora de programar con Gambas aplicaciones de red, y que son independientes del método de transporte que haya por debajo.

El protocolo más extendido hoy en día para distribuir información entre equipos es el protocolo IP, que determina qué destino y qué equipo recibirá cada fragmento de información que circula por una red. Cada equipo tiene un número asignado, llamado dirección IP es su identificador único. Cada paquete IP de información es similar a una carta postal: incluye datos del remitente (IP del equipo de origen) y del destinatario (IP del equipo de destino). Estas direcciones IP tienen la forma *XXX.XXX.XXX.XXX*; son grupos de cuatro números que varían entre 0 y 255, separados por puntos (por ejemplo: 192.168.0.23 o 10.124.35.127).

Dado que estos números son difíciles de recordar, se estableció un sistema que permitía establecer una correspondencia entre nombres y direcciones IP. Este sistema se denomina DNS y, a lo largo y ancho de Internet y de casi todas las redes locales, se encuentran servidores DNS que reciben consultas acerca de nombres de equipos y direcciones IP y las traducen en un sentido u otro. De esta forma, podemos, por ejemplo, indicar al navegador que muestre la página **www.gnulinex.org**, en lugar de tener que indicar la dirección IP del equipo que está sirviendo esta página.

Los paquetes IP pueden contener cualquier tipo de información, pero encima de este nivel se han establecido otros dos también estándar, que son los protocolos TCP y UDP.

1. **TCP** se utiliza para asegurar la conexión entre dos equipos: se sabe en todo momento si el equipo remoto está a la escucha, si ha recibido toda la información correctamente o hay que volverla a emitir, y se añaden sistemas de control de errores para asegurar que un ruido en la línea no ha deformado los mensajes. Es el protocolo más utilizado en Internet, ya que asegura que recibamos una página web o un fichero de forma completa y sin errores.

2. En cuanto a UDP, es un protocolo de transporte mucho más simple que TCP, y no verifica si realmente existe una conexión entre los dos equipos o si la información ha sido realmente recibida o no. Es, por tanto, menos fiable, pero en determinados tipos de transmisiones, como son las de audio y vídeo en tiempo real, lo importante no es que llegue toda la información (puede haber pequeños cortes o errores), sino que el caudal sea constante y lo más rápido posible. Por ello, también se emplea con frecuencia en Internet, sobre todo para los llamados servidores de *streaming*, que nos permiten escuchar radio, ver programas de televisión o realizar videoconferencias por la red.

A la hora de establecer una comunicación entre dos equipos, el modelo indica que ha de abrirse un *socket*. Es algo similar a una bandeja de entrada o salida, como la de los administrativos, que tienen una bandeja con los informes pendientes y otra con los que van terminando. El sistema operativo almacena información en la bandeja de entrada proveniente del sistema remoto, hasta que nuestro programa decide tomar los datos, entonces los procesamos y los dejamos en la bandeja de salida. El sistema operativo se encargará de enviar esa información cuando le sea posible.

Sockets no se emplea sólo para comunicar dos o más equipos, dentro de nuestra propia máquina muchos programas se comunican entre sí utilizando este sistema. Por ejemplo, es el caso de los servidores gráficos o X-Window: el servidor gráfico dibuja lo que le piden los programas clientes o aplicaciones que han establecido un socket con él.

Existe un tipo especial de socket, llamado *local* o *Unix*, que sólo sirve para comunicar programas dentro de un mismo sistema, y que está optimizado para realizar esa función con gran velocidad y eficacia, permitiendo una comunicación interna varias veces más veloz y con menor consumo de recursos que a través de un socket TCP o UDP.

Subiendo más arriba, tenemos los llamados *protocolos de aplicación*. Ya no nos encargamos del transporte de datos, sino del formado de los datos y de la comunicación. Uno de los protocolos más extendidos es el HTTP que, entre otras cosas, se utiliza

en todo Internet para transmitir y recibir páginas web. Establece el modo de conectar, cómo autenticarse, el formato de los datos a transmitir o recibir (páginas web, imágenes, archivos comprimidos...) y cómo finalizar la comunicación.

Otros protocolos específicos son el FTP, especializado en la transmisión y recepción de archivos; TELNET, para trabajar con un terminal de texto sobre un sistema remoto; SMTP, POP o IMAP para el correo electrónico o los diferentes protocolos de mensajería instantánea, como el JABBER.

Si esta breve introducción nos ha resultado nueva, debemos extender nuestros conocimientos en el área de redes, aprendiendo conceptos como las máscaras de red y divisiones en subredes, los diferentes tipos de proxies, routers y gateways, las conversiones NAT, el futuro protocolo Ipvó y, en general, todo lo que nos ayude a determinar por qué no conseguimos conectar con otro equipo en un momento dado. El mundo de las redes es extenso, pero una buena base nos evitará muchos problemas a la hora de programar sistemas distribuidos entre varios servidores y clientes.

6. 2 Creando un servidor TCP

Nuestra primera tarea consistirá en escribir el código de un servidor TCP: será un servidor que aceptará conexiones remotas, leerá los datos que envían y devolverá un eco a los clientes. Crearemos un programa de consola llamado *MiServidor*. Este programa tendrá un único módulo llamado **ModMain**. En las propiedades del proyecto hemos de seleccionar el componente **gb.net**.

Dentro del módulo **ModMain** tendremos una referencia a un objeto *ServerSocket*. Dichos objetos se comportan como servidores de sockets, es decir, se encuentran a la escucha de peticiones de conexión de clientes remotos o locales en un puerto determinado. Los puertos se numeran del 1 al 65535 y cada programa que actúa como servidor dentro del sistema utiliza uno de ellos.

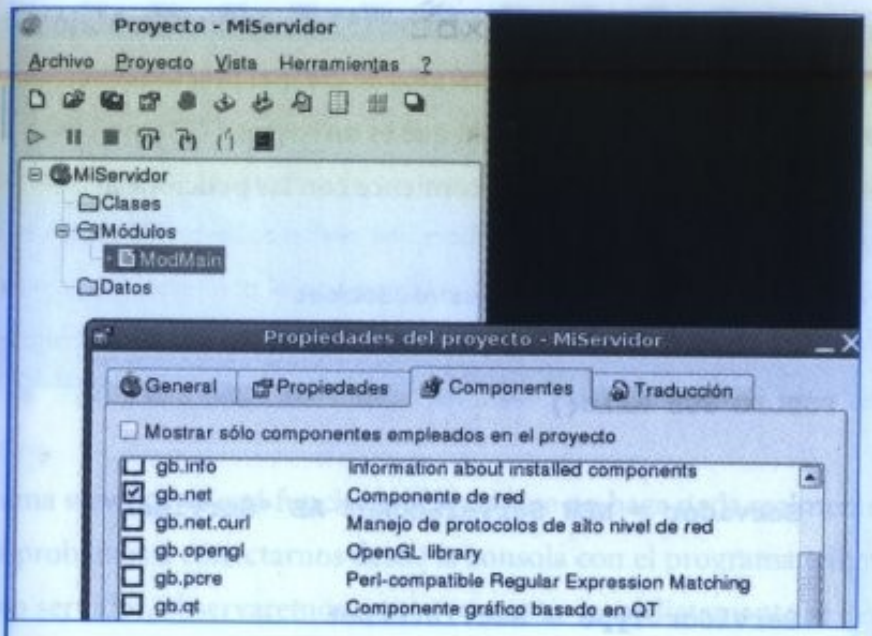

Figura 1. Programa *MiServidor*.

Los puertos con número del 1 al 1024, se consideran reservados para servicios conocidos (es decir, HTTP, POP, FTP, IMAP...) y no pueden ser utilizados por programas cuyo usuario es distinto del root en sistemas GNU/Linux. Tratar, por tanto, de abrir, por ejemplo, el puerto 523 desde un programa ejecutado por un usuario normal, dará lugar a un error.

Los servicios más conocidos como HTTP o FTP ya tienen un puerto asignado (por ejemplo, 80 en el caso de HTTP) de forma estándar, si bien no hay ninguna razón técnica por la que un servidor web no pueda atender, por ejemplo, al puerto 9854. Esto se debe únicamente a un acuerdo internacional para que cualquier cliente que quiera realizar una petición web, por ejemplo, sepa que, salvo instrucciones específicas en otro sentido, ha de conectarse al puerto 80 de la máquina servidora.

En sistemas GNU/Linux podemos encontrar una lista de los servicios más comunes y sus puertos *oficiales* dentro del fichero de texto */etc/services*.

En nuestro caso, vamos a emplear el puerto 3152. En la función Main del programa, que se ejecuta al inicio de éste, habremos de crear el objeto ServerSocket, especificar que su tipo será Internet (es decir, que es un socket TCP y no UNIX), el puerto al que debe atender e indicarle que comience con las peticiones.

```
PRIVATE Servidor AS ServerSocket

PUBLIC SUB Main()

  Servidor = NEW ServerSocket AS "Servidor"

  Servidor.Type = Net.Internet
  Servidor.Port = 3152
  TRY Servidor.Listen()
  IF ERROR THEN PRINT "Error: el sistema no permite
  atender al puerto especificado"

END
```

Observemos que a la hora de crear el objeto servidor, indicamos que el gestor de eventos será "Servidor", para poder escribir las rutinas en las cuales tratemos las peticiones que llegan de los clientes.

A la hora de indicar al servidor que se ponga a la escucha con el método Listen, lo hacemos protegiendo el código con una instrucción TRY, para gestionar el error si el sistema no permitiese atender al puerto indicado (por ejemplo, si otro servicio ya lo estuviera utilizando). Ya podemos ejecutar el programa.

Como podemos observar, el programa pasa la función Main y sigue ejecutándose a la espera de una conexión de un cliente. Un programa normal de consola, al terminar esta función, hubiera finalizado sin más, pero en este caso el intérprete Gambas está atendiendo las novedades que puedan acaecer en el socket que se ha creado y, por tanto, el programa sigue funcionando, esperando peticiones de clientes.

Cualquier programa Gambas que está vigilando un *descriptor*, es decir, un archivo, un proceso o un socket, no finaliza mientras dicho *descriptor* se encuentre abierto. Esta técnica permite crear programas de consola que no se encuentren en un bucle continuo a la espera de novedades (sea una modificación en un fichero, una entrada de un cliente en un socket o la lectura de datos de un proceso hijo), ahorrando recursos y mejorando la eficacia del proceso.

El programa servidor ya está funcionando, aunque no haga nada realmente útil. De hecho, si probamos a conectarnos desde la consola con el programa telnet a nuestro propio servidor, observaremos que se conecta e inmediatamente se desconecta del servicio.

```
dcampos@gnuLinEx: /home/
Archivo  Editar  Ver  Terminal  Solapas  Ayuda
dcampos@gnuLinEx:~$ telnet 127.0.0.1 3152
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
Connection closed by foreign host.
dcampos@gnuLinEx:~$
```

Figura 2. Prueba de conexión.

El funcionamiento del objeto ServerSocket es precisamente éste: recibe una conexión de un cliente y, si no especificamos en el programa que deseamos atenderla, la cierra.

Cuando un objeto servidor recibe una petición de un cliente, se dispara el evento **Connection**. Dentro de dicho evento, y sólo dentro de él, podemos utilizar el método **Accept**. Dicho método devuelve un objeto **Socket** que representa una conexión con dicho cliente. A la hora de recibir o enviar datos a ese cliente, se hará a través de dicho objeto, de ese modo, se diferencia a cada cliente conectado con un servidor de forma única, evitando, por ejemplo, que enviemos los resultados de un cálculo a un cliente equivocado.

En nuestro programa añadiremos una matriz de objetos donde guardaremos una referencia a cada objeto cliente, y en esa matriz iremos añadiendo los clientes que se conectan.

```
PRIVATE Servidor AS ServerSocket
PRIVATE Clientes AS NEW Object[]

PUBLIC SUB Servidor_Connection(RemoteHostIP AS String)

  DIM Cliente AS Socket
  Cliente = Servidor.Accept()
  Clientes.Add(Cliente)
END

PUBLIC SUB Main()

  Servidor = NEW ServerSocket AS "Servidor"
  Servidor.Type = Net.Internet
  Servidor.Port = 3152
  TRY Servidor.Listen()
  IF ERROR THEN PRINT "Error: el sistema no permite
  atender al puerto especificado"
END
```

En nuestro gestor de eventos Connection utilizamos el método Accept, por lo que siempre aceptamos las conexiones entrantes. No obstante, si lo omitimos en casos determinados, el intento de conexión del cliente se cerrará automáticamente. Observamos, por ejemplo, que Accept nos informa de la dirección IP del cliente que

trata de conectarse a través del parámetro RemoteHostIP. Si deseáramos, por ejemplo, aceptar sólo conexiones desde nuestro propio equipo (que siempre tiene dirección 127.0.0.1), podríamos modificar el código para que se rechacen las conexiones desde otras direcciones IP.

```
PUBLIC SUB Servidor_Connection(RemoteHostIP AS String)

  DIM Cliente AS Socket

  IF RemoteHostIP="127.0.0.1" THEN
      Cliente = Servidor.Accept()
      Clientes.Add(Cliente)
  END IF

END
```

Estos objetos socket, por defecto reciben ya un gestor de eventos llamado Socket, por lo que los eventos de estos clientes de socket se atenderán en el programa por una función llamada *Socket_ + Nombre del evento*.

En nuestro caso atenderemos a los eventos *Read* que se producen cuando se reciben datos desde el cliente, y el evento *Close*, que se produce cuando el cliente cierra la conexión.

En el caso del evento READ, nos valdremos de la palabra clave LAST, que mantiene una referencia al último objeto que ha disparado un evento (en este caso, el cliente de socket) para tratar los datos que provienen de él.

Para ello, leeremos el socket como si fuera un archivo, con la instrucción READ, en la que indicamos una cadena que recibe los datos; leeremos la longitud de los datos disponibles en el socket, determinada por la función Lof() y, después, escribiremos esos mismos datos en el socket con la instrucción WRITE, de forma que el cliente reciba un eco de los datos enviados, pero convertidos a mayúsculas.

```
PUBLIC SUB Socket_Read()

  DIM sCad AS String

  TRY READ #LAST, sCad, Lof(LAST)
  sCad=UCase(sCad)
  TRY WRITE #LAST, sCad, Len(sCad)

END
```

Para **Close**, buscamos el objeto **Socket** dentro de nuestra matriz, y lo eliminamos para que desaparezcan las referencias a dicho cliente dentro de nuestro programa servidor.

```
PUBLIC SUB Socket_Close()

  DIM Ind AS Integer

  Ind = Clientes.Find(LAST)
  IF Ind >= 0 THEN Clientes.Remove(Ind)

END
```

Ya disponemos de un programa servidor de socket con capacidad para atender a múltiples clientes: podemos abrir varias ventanas de terminal, ejecutando en ellas **telnet 127.0.0.1 3152**, y enviar y recibir datos del servidor. Cada vez que escribamos una cadena y la enviemos pulsando **Return**, recibiremos la cadena convertida a mayúsculas como respuesta del servidor.

Aunque el programa telnet se diseñó para controlar un equipo de forma remota, también es un *cliente universal* que puede servir para comprobar el funcionamiento de cualquier servidor que estemos diseñando a mano, antes de disponer de un cliente real.

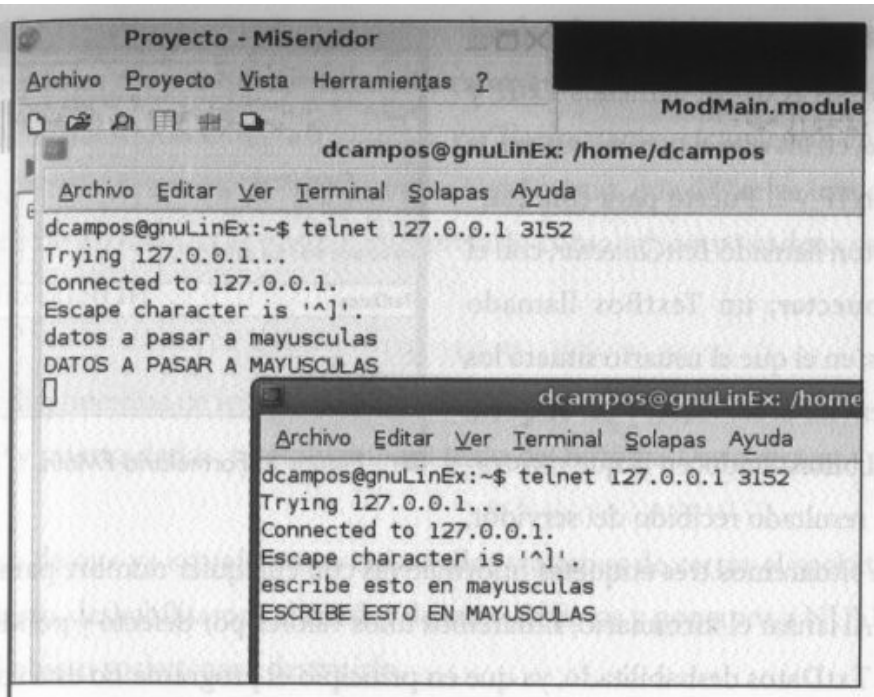

Figura 3. Programa servidor de socket.

6.3 Un cliente TCP

Ahora que ya disponemos de un servidor, vamos a crear un programa cliente.

Diseñaremos un programa gráfico llamado *MiCliente*, con un formulario **FMain**, y una referencia al componente **gb.net** como en el caso anterior.

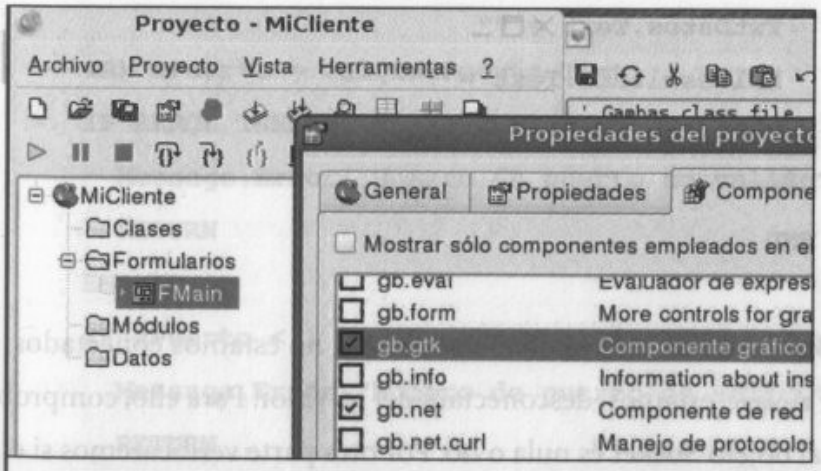

Figura 4. Proyecto *MiCliente*.

Dentro del formulario **FMain** situaremos dos controles TextBox, llamados *TxtIP* y *TxtPuerto*, en los que el usuario situará la **Dirección IP** y el **Puerto** para conectarse; un botón llamado *BtnConectar*, con el texto **Conectar**; un TextBox llamado *TxtDatos*, en el que el usuario situará los datos a enviar al servidor, y un etiqueta llamada **LblResultado**, en la que mostraremos el resultado recibido del servidor.

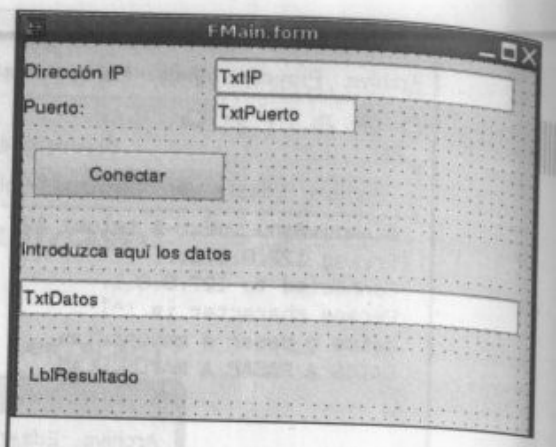

Figura 5. Formulario *FMain*.

También situaremos tres etiquetas informativas con cualquier nombre para guiar al usuario. Al lanzar el formulario, situaremos unos valores por defecto y pondremos el TextBox **TxtDatos** deshabilitado, ya que en principio el programa no está conectado al servidor. También situaremos al principio del código una variable global de tipo **Socket**, que representará al cliente con el que nos conectaremos al servidor.


```
' Gambas class file
PRIVATE Cliente AS Socket

PUBLIC SUB Form_Open()

  TxtIp.Text = "127.0.0.1"
  TxtPuerto.Text = "3152"
  TxtDatos.Text = ""
  LblResultado.Text = ""
  TxtDatos.Enabled = FALSE

END
```

El botón **BtnConectar** tendrá dos funciones: si no estamos conectados, tratará de conectar, y si ya lo estamos, desconectará del servidor. Para ello, comprobamos si la referencia al objeto **Socket** es nula o no. Por otra parte verificaremos si el dato de la dirección IP es válido, así como el del puerto indicado.

Para verificar el número de puerto usamos dos funciones: **Val()**, devuelve un número a partir de un texto si éste contiene sólo caracteres numéricos. Si no es así, devuelve NULL. Si hemos obtenido un número, verificamos que se encuentre en el rango 1-65535. Para la dirección IP, emplearemos **Net.Format**, que devuelve una dirección IP a partir de un texto, si es posible interpretarlo como tal, o una cadena vacía si no es una dirección IP.

Una vez disponemos de los datos, creamos el objeto **Socket** y tratamos de conectar con la IP y puerto dados, y cambiamos el texto del botón a *Desconectar*.

En el caso de que ya estuviéramos conectados tratamos de cerrar el socket, si estuviera abierto, deshabilitamos el cuadro de texto de datos y ponemos a NULL la referencia al objeto socket, para destruirlo.

```
PUBLIC SUB BtnConectar_Click()

  DIM nPuerto AS Integer
  DIM sIP AS String

  LblResultado.Text = ""
  TxtDatos.Text = ""

  IF Cliente = NULL THEN

    TRY nPuerto = Val(TxtPuerto.Text)
    IF ERROR THEN
      Message.Error("Número de puerto no válido")
      RETURN
    END IF
    IF nPuerto < 1 OR nPuerto > 65535 THEN
      Message.Error("Número de puerto no válido")
      RETURN
    END IF
```

```
    sIP = Net.Format(TxtIP.Text)
    IF sIP = "" THEN
      Message.Error("Dirección IP no válida")
      RETURN
    END IF

    Cliente = NEW Socket AS "Cliente"
    Cliente.Host = sIP
    Cliente.Port = nPuerto
    Cliente.Connect()

  ELSE

    TRY CLOSE #Cliente
    Cliente = NULL
    TxtDatos.Enabled = FALSE

  END IF

END
```


Entre el intento de conexión y la conexión real puede pasar un intervalo de tiempo. Para saber cuándo hemos conectado realmente con el servidor, emplearemos el evento **Ready**, que se produce cuando el servidor acepta nuestra conexión. En este evento, habilitaremos el cuadro de texto de datos para que el usuario pueda escribir allí. También puede suceder que se produzca un error (por ejemplo, una conexión rechazada o un nombre de host no encontrado), en cuyo caso se dispara el evento **Error**, que aprovechamos para devolver la interfaz a su estado desconectado y eliminar el socket fallido.

```
PUBLIC SUB Cliente_Ready()

  TxtDatos.Enabled = TRUE
```

```
END

PUBLIC SUB Cliente_Error()

  TRY CLOSE #Cliente
  Cliente = NULL
  TxtDatos.Enabled = FALSE
  BtnConectar.Text = "Conectar"
  Message.Error ("Conexión cerrada")

END
```

En cuanto al envío de datos, lo realizaremos cuando el usuario pulse la tecla **Return**, tras escribir un texto en el cuadro **TxtDatos**.

Para ello introduciremos el código dentro del evento **KeyPress** del cuadro de texto.

```
PUBLIC SUB TxtDatos_KeyPress()

  IF Key.Code = Key.Return THEN

    IF Len(TxtDatos.Text) > 0 THEN

      LblResultado.Text = ""
      TRY WRITE #Cliente, TxtDatos.Text,
      Len(TxtDatos.Text)

    END IF

  END IF

END
```

En cuanto a la recepción de datos del servidor, puede llegarnos en varios fragmentos, cada uno de los cuales recibiremos en el evento **Read** del objeto socket, y empalmaremos en la etiqueta **LblResultado**.

Cuando se envían y reciben datos en una red, nunca se puede asegurar qué cantidad de datos se reciben de una sola vez, siempre hay que tener en cuenta la posibilidad de unir fragmentos de los datos totales.

```
PUBLIC SUB Cliente_Read()

  DIM sCad AS String

  TRY READ #Cliente, sCad, Lof(Cliente)
  LblResultado.Text = LblResultado.Text & sCad

END
```


Ya tenemos nuestro cliente listo. Arrancamos el servidor que creamos anteriormente y una o varias instancias de este programa para comprobar los resultados.

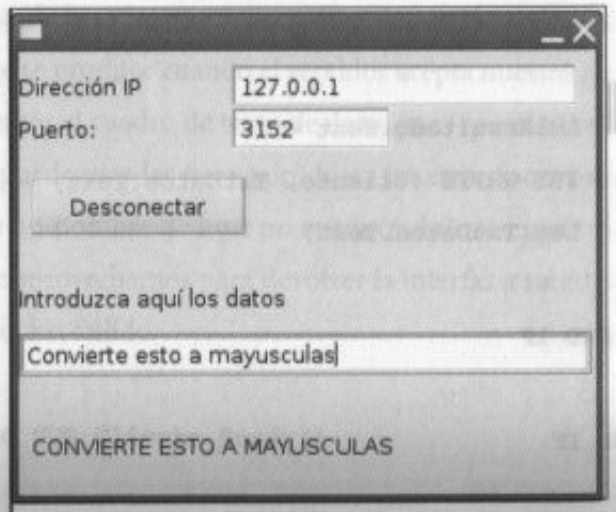

Figura 6. Resultado final de nuestro cliente.

A la hora de crear clientes o servidores para producción, debemos tener en cuenta la codificación de los caracteres empleada por el servidor y el cliente. Muchos errores pueden provenir de no tener en cuenta, simplemente, que Gambas y muchos otros programas usan internamente UTF-8 como codificación predeterminada, mientras que otros emplean ISO-8859-1 o UTF-16. Este programa, por ejemplo, puede tener problemas con la *ñ* y vocales acentuadas. Podemos mejorarlo como ejercicio, convirtiendo las cadenas enviadas y recibidas entre UTF-8 e ISO-8859-1 o ISO-8859-15.

6. 4 Clientes y servidores locales

Al margen de los sockets TCP, diseñados para conectar equipos remotos, los sistemas GNU/Linux, y en general cualquier sistema que siga la filosofía de la familia de sistemas UNIX™, disponen de otro tipo de sockets, que sólo permiten la conexión dentro del propio equipo y cuya misión es optimizar la funcionalidad de los sockets cuando el cliente y servidor se encuentran en la misma máquina.

Estos sockets son unos ficheros especiales que el servidor crea dentro del sistema de archivos, y que los clientes tratan de abrir para lectura y escritura de datos. Hemos de tener en cuenta que situar uno de estos ficheros especiales dentro de una unidad de red no sirve para conectar dos equipos diferentes: un socket UNIX o local, sólo se puede emplear dentro de un mismo equipo.

Todo lo explicado hasta aquí para clientes y servidores TCP, es aplicable a los clientes y servidores locales, la única diferencia se da en el modo inicial de configurar el servicio.

En el caso del servidor, antes de conectar hemos de especificar que el tipo de servidor es local y una ruta a un archivo que representará el socket servidor.

```
' Servidor TCP:
PUBLIC SUB Main()
```

```
    Servidor = NEW ServerSocket AS "Servidor"

    Servidor.Type = Net.Internet
    Servidor.Port = 3152
    TRY Servidor.Listen()
    IF ERROR THEN PRINT "Error: el sistema no permite
    atender al puerto especificado"

END

' Servidor "Local" o "UNIX"
PUBLIC SUB Main()

    Servidor = NEW ServerSocket AS "Servidor"

    Servidor.Type = Net.Local
    Servidor.Path = User.Home & "/misocket"
    TRY Servidor.Listen()
    IF ERROR THEN PRINT "Error: el sistema no permite
    atender al puerto especificado"

END
```

Si ejecutamos el código con esta modificación, observaremos que en nuestra carpeta personal se crea un archivo especial de tipo socket llamado misocket. En la parte cliente hemos de especificar que la conexión se realiza con una ruta dentro del sistema de archivos, en lugar de un puerto TCP, para lo cual se indica el puerto especial Net.Local y la propiedad Path del socket.

```
Cliente = NEW Socket AS "Cliente"
Cliente.Path = User.Home & "/misocket"
Cliente.Port = Net.Local
Cliente.Connect()
```

A partir de aquí el resto del código funcionará sin modificaciones.

Es una buena idea plantear la posibilidad de que servidores y clientes funcionen con sockets locales o TCP, según los parámetros de configuración de la aplicación, con el fin de optimizar el funcionamiento cuando cliente y servidor se encuentran en la misma máquina. Los servidores X y los de fuentes gráficas emplean este modo de trabajo.

6.5 UDP

El trabajo con UDP al principio puede resultar algo más complejo al principiante. En UDP no existe una conexión entre cliente y servidor, tan sólo llegan datos por un puerto, o se envían, pero sin conocer nunca si al otro lado hay alguien a la escucha. De hecho, no existen servidores o clientes realmente, aunque en la práctica sí haremos esta distinción a la hora de diseñar una aplicación servidora o cliente. Puesto que no existe conexión, cada paquete se identifica con su dirección IP y puerto de origen, y este dato ha de ser tenido en cuenta siempre a la hora de devolver una respuesta al lado remoto.

Gambas da acceso al protocolo UDP a través de los objetos de la clase **UdpSocket**, que sirven tanto para crear programas servidores como programas cliente. Para iniciar el trabajo con un objeto de esta clase, se ha de llamar al método *Bind*, en el cual se ha de indicar el puerto local al que estará ligado. Los servidores habrán de estar unidos a un puerto concreto definido, y los clientes a cualquier puerto, ya que el interés del cliente es enviar datos a un servidor remoto en un puerto dado, independientemente del puerto local que emplee para ello.

UdpSocket proporciona varias propiedades para identificar los paquetes. Cuando llega un paquete al socket procedente de una ubicación remota, se genera un evento **Read** y en él podemos consultar la IP de procedencia, con la propiedad **SourceHost**, y el puerto de origen, con la propiedad **SourcePort**. Para enviar un paquete a un sistema remoto, hemos de rellenar previamente el valor de las propiedades **TargetHost**, con la IP de destino, y **TargetPort**, con el puerto de destino.

Crearemos un servidor muy sencillo: atendemos al puerto UDP 3319, recibe los datos y envía como respuesta el texto ADIOS al host que envió el paquete. Para implementarlo crearemos un proyecto de consola llamado MiServidorUDP, con un módulo *ModMain* y una referencia al componente gb.net. El código es el siguiente:

```
' Gambas module file
PRIVATE MiServidor AS UdpSocket

PUBLIC SUB Servidor_Read()

  DIM sCad AS String

  READ #MiServidor, sCad, Lof(MiServidor)

  MiServidor.TargetHost = MiServidor.SourceHost
  MiServidor.TargetPort = MiServidor.SourcePort
  WRITE #MiServidor, "ADIOS", 5

END

PUBLIC SUB Main()

  MiServidor = NEW UdpSocket AS "Servidor"
  MiServidor.Bind(3319)

END
```

Dispondremos de un objeto de la clase UdpSocket que creamos en la función Main. Hacemos una llamada a Bind para enlazarlo con el puerto UDP 3319. Como en el caso del servidor TCP, el intérprete Gambas queda a la espera de algún evento en el socket. Cuando llega un paquete de datos, se lee como en el caso de los sockets TCP, haciendo uso de la instrucción READ, a continuación situamos las propiedades

TargetHost y TargetPort con los valores correspondientes a la procedencia del paquete de datos, y enviamos la cadena ADIOS, que llegará al host especificado. En cuanto a la parte cliente, se tratará de otro programa de consola que llamaremos MiClienteUDP, con referencia al componente gb.net, y que tendrá un único módulo llamado *ModMain*. El cliente tratará de conectar con el servidor, enviará una cadena HOLA y cuando reciba una cadena completa ADIOS, cerrará el socket y acabará su funcionamiento.

```
' Gambas module file
PRIVATE MiCliente AS UdpSocket
PRIVATE Buffer AS String

PUBLIC SUB Cliente_Read()

  DIM sBuf AS String

  READ #MiCliente, sBuf, Lof(MiCliente)
  Buffer = Buffer & sBuf
  IF Buffer = "ADIOS" THEN CLOSE #MiCliente

END

PUBLIC SUB Main()

  MiCliente = NEW Udpsocket AS "Cliente"

  MiCliente.Bind(0)

  MiCliente.TargetHost = "127.0.0.1"
  MiCliente.TargetPort = 3319
  WRITE #MiCliente, "HOLA", &
  "
END
```

El control **DnsClient** recibirá el nombre *Cliente* en nuestro formulario, y la propiedad *Async* del cliente **DNS** se debe situar con el valor *FALSE*.

Estos controles especiales (DnsClient, Socket, etc.) no son visibles en tiempo de ejecución, ni son realmente controles, pero se añaden al formulario como controles normales con el fin de facilitar la labor al crear programas gráficos. El objeto DnsClient, en nuestro caso llamado *Cliente*, se crea al desarrollar el formulario y existe hasta que se destruye al cerrarlo o al aplicar el método *Delete* en dicho formulario.

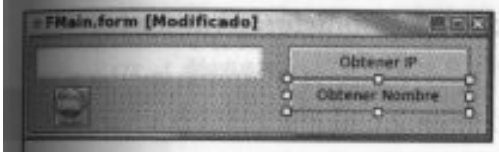

Figura 8. Formulario *FMain*.

El aspecto del diseño será similar al de la Figura 8.

El código será el siguiente:

```
PUBLIC SUB BtnToIp_Click()

  Cliente.HostName = TxtHost.Text
  Cliente.GetHostIP()
  Message.Info("IP: " & Cliente.HostIP)

END

PUBLIC SUB BtnToName_Click()

  Cliente.HostIp = TxtHost.Text
  Cliente.GetHostName()
  Message.Info("Nombre: " & Cliente.HostName)

END
```

En el primer caso, se rellena la propiedad **HostName** con el nombre del host que deseamos resolver; se llama al método **GetHostIP()** y, después, leemos la IP de dicho host recién obtenida con el valor de la propiedad **HostIP**.

El segundo caso es exactamente el contrario: rellenamos la propiedad **HostIp** con la IP que ha escrito el usuario en la caja de texto; llamamos al método **GetHostName()** y leemos y mostramos el valor de la propiedad **HostName** que hemos resuelto.

En ambos casos, si la resolución falla, la propiedad a leer (**HostIP** en el primer caso y **HostName** en el segundo) quedará en blanco. También podemos controlar el estado de error con la propiedad *Status*. Si vale cero, la resolución terminó correctamente; si tiene un valor menor que cero, hubo un error, no se encontró la IP o nombre de host.

DnsClient también permite el trabajo en modo asíncrono. En ocasiones, una resolución puede tardar bastante tiempo, en el que la interfaz de usuario queda bloqueada trabajando en modo síncrono.

Para activar el modo asíncrono, se debe poner la propiedad **Async** del cliente DNS a **TRUE**. Esto también requiere modificar el código. Ya no podemos llamar al método **GetHostIP()** o **GetHostName()** e inmediatamente leer el valor de las propiedades **HostName** o **HostIp** respectivamente, ya que mientras el código del programa principal se está ejecutando, el cliente DNS está trabajando internamente. Tenemos dos opciones: la primera es esperar el evento *Finished*, que se dispara al acabar el proceso; y la segunda es comprobar el estado de la propiedad **Status**, que valdrá un número mayor que cero mientras el proceso se haya en curso, cero cuando finaliza con éxito o un valor menor que cero si hubo un error.

Éste es un pequeño ejemplo modificando el código inicial para trabajar en modo asíncrono. Se comprueba en un bucle el estado de la propiedad **Status**, repitiéndose mientras es mayor que cero. En ese bucle se podría, por ejemplo, controlar la pulsación de un botón **Cancelar** para abortar el proceso.

```
PUBLIC SUB BtnToIp_Click()

  Cliente.Async = TRUE
  Cliente.HostName = TxtHost.Text
  Cliente.GetHostIP()

  DO WHILE Cliente.Status > 0

    WAIT 0.01

  LOOP

  Message.Info("IP: " & Cliente.HostIP)

END

PUBLIC SUB BtnToName_Click()

  Cliente.Async = TRUE
  Cliente.HostIp = TxtHost.Text
  Cliente.GetHostName()

  DO WHILE Cliente.Status > 0

    WAIT 0.01

  LOOP

  Message.Info("Nombre: " & Cliente.HostName)

END
```

6. 7 Protocolo HTTP

En un nivel por encima de todo lo explicado hasta ahora se encuentra el protocolo HTTP, en el que ya no se trata sólo de conectar con un servidor, sino de establecer un formato para la comunicación entre clientes y servidores. El protocolo HTTP es uno de los más extendidos a lo largo de Internet, dado que, entre otras cosas, se emplea para transmitir páginas web.

En el protocolo HTTP, el cliente solicita al servidor un documento en una ubicación dada, y éste lo devuelve siguiendo una serie de convenciones acerca de la codificación de caracteres, control de errores, nivel de compresión de datos, formato de los datos binarios, etc.

El protocolo HTTP establece dos métodos principales para solicitar datos al servidor. El primero, y más común por ser utilizado para recibir páginas web en el navegador cuando escribimos una dirección, por ejemplo, es el llamado método *GET*, en el cual el cliente tan sólo solicita una dirección URL, devolviendo el servidor el resultado. En el segundo método, llamado *POST*, el cliente no sólo solicita la URL, si no que envía una serie de informaciones adicionales que el servidor procesará antes de enviar el resultado. Es el caso habitual cuando rellenamos un formulario con datos en una página web y pulsamos el botón enviar.

Gambas aporta en el componente *gb.net.curl*, un cliente llamado *HttpClient*, que provee acceso a servidores HTTP. El trabajo de negociación entre cliente y servidor, así como la gestión de formatos, es tarea interna del cliente HTTP, la aplicación que desarrollemos sólo habrá de preocuparse en pedir un documento y recibirlo.

El cliente HTTP puede funcionar de dos modos: el más simple es el modo síncrono, en el que recibiremos directamente el documento tras la llamada a los métodos *GET* o *POST*; en el segundo, asíncrono, el modo de trabajo se parece más al descrito para los sockets: con la aplicación en funcionamiento iremos recibiendo fragmentos del documento en cada evento *READ* que uniremos en una cadena.

Crearemos un proyecto de consola para recibir la página web **http://gambas.gnulinex.org/gtk/**. Tendrá referencias a *gb.net.curl* y *gb.net*, y un módulo *ModMain*, con este código:

```
PUBLIC SUB Main()

  DIM Http AS HttpClient
  DIM sCad AS String

  Http = NEW HttpClient

  Http.Async = FALSE
  Http.TimeOut = 10
  Http.URL = "http://gambas.gnulinex.org/gtk/"

  Http.Get()

  IF Http.Status < 0 THEN
    PRINT "Error al recibir la página"
  ELSE
    READ #Http, sCad, Lof(Http)
    PRINT sCad
  END IF

  CLOSE #Http

END
```

En primer lugar definimos y creamos un objeto de la clase **HttpClient** llamado **Http**. Situamos su propiedad **Async** a **FALSE** para que el proceso sea síncrono, es decir, que el cliente HTTP quede bloqueado mientras se recibe la página. Puesto que el bloqueo podría durar un tiempo excesivo, definimos también con la propiedad **TimeOut** un tiempo máximo en segundos antes de dar por fracasada la conexión y recepción de datos.

Indicamos la URL que contiene el nombre del servidor y el documento dentro de éste.

Llamamos al método Get con el fin de recibir la página, con lo que el progran interrumpido hasta su recepción hasta un error o hasta que pasen 10 segund máximo.

Leemos el valor de la propiedad Status que, como en el caso de los sockets o te DNS, tendrá valor mayor que cero cuando está activo, cero en caso de éxito que cero en caso de error. Si hay un error, lo indicamos por la consola. Si bien, leemos, como en el caso de los sockets o cualquier otro flujo, empleanc trucción READ y mostramos el contenido de la página web recibida por co

Finalmente, cerramos el cliente Http. Esto es necesario, ya que el cliente mantener viva la conexión con el servidor mientras le es posible, para ace esta manera la recepción de múltiples documentos de un mismo servidor.

El protocolo HTTP establece un tiempo en el que el servidor mantiene el socket to con el cliente. Si necesitamos recibir varias páginas de un servidor o realizar peticiones POST consecutivas, deberíamos emplear la instrucción CLOSE sólo a de todo el proceso, con lo cual se ganará en velocidad y se utilizarán menos sos del sistema.

En cuanto al método asíncrono, el modo de proceder es exactamente el mis con los sockets: esperar al evento *Read* para ir leyendo fragmentos del doc en proceso de recepción, o al evento *Error* para atender posibles problemas de nicación con el servidor.

Al margen de los problemas físicos de comunicación, que se detectan con el *Error*, el propio protocolo HTTP puede especificar códigos de error en los que la nicación/recepción de datos ha sido perfecta a nivel físico, pero se han pro

ógicos. El caso más común es solicitar un documento que no existe en el en cuyo caso se genera el error *404*. Otros errores comunes pueden ser el so prohibido) o el *500* (error interno del servidor). Para detectar y tratar blemas, el cliente HTTP proporciona dos propiedades que se pueden con- *ode*, que es numérica y devuelve el código de error (el valor *200* significa fue bien), y *Reason*, que es una cadena de texto explicativa del error, pro- da por el servidor.

r devuelto en *Reason* depende del programa utilizado como servidor web y no gatorio. Por ello, a efectos de control de errores, debe emplearse sólo el valor co *Code* y usarse *Reason* sólo a efectos de información del usuario.

mentos HTTP se reciben empaquetados, de forma que existe un encabeza- ontiene *meta información* del servidor y un cuerpo donde se encuentra alo- ocumento en sí. La lectura mediante **READ** u otros métodos para flujos sólo cceso a los datos del cuerpo. Para obtener las cabeceras, que pueden ser úti- comprobar datos tales como la fecha y hora del servidor, el tipo de servidor nformaciones específicas, podemos recurrir a la propiedad **Headers**. Cada e *meta información* es una línea, por lo que **Headers** devuelve una matriz de cada una de las cuales es una información procedente del servidor. En este derivado del anterior se muestra al final los encabezados del servidor.

```
PUBLIC SUB Main()

    DIM Http AS HttpClient
    DIM sCad AS String

    Http = NEW HttpClient

    Http.Async = FALSE
    Http.TimeOut = 10
```

```
Http.URL = "http://gambas.gnulinex.org/gtk/"

Http.Get()

IF Http.Status < 0 THEN
  PRINT "Error al recibir la página"
ELSE
  READ #Http, sCad, Lof(Http)
  PRINT sCad


  PRINT "\n-----------Metadatos-------\n"
  FOR EACH sCad IN Http.Headers
    PRINT sCad
  NEXT

END IF

CLOSE #Http

END
```


El protocolo HTTP permite al cliente autenticarse para recibir determinadas páginas no accesibles al público en general. El usuario ha de disponer de un nombre de usuario y contraseña. Para introducir estos valores, se ha de especificar el nombre de usuario en la propiedad *User* y la contraseña en la propiedad *Password*, antes de llamar a los métodos *Get* o *Post*.

Por otra parte existen diversos métodos de autenticación, desde los menos seguros (envío de usuario/clave sin codificar) hasta algunos algoritmos más seguros. Además del usuario y clave, se ha de especificar el método de autenticación rellenando el valor de la propiedad *Auth* con alguna de las constantes que representan los métodos soportados:

Net.AuthBasic
Net.AuthDigest
Net.AuthGSSNegotiate
Net.AuthNTLM

El desarrollador de la aplicación ha de conocer de antemano el método que necesita para acceder al servidor.

El protocolo HTTP también contempla el uso de cookies, que es información que el servidor guarda en el cliente y que es consultada de nuevo por el servidor en ocasiones posteriores, antes de enviar un documento al cliente. Puede servir, por ejemplo, para saber si la página ya ha sido visitada con anterioridad por dicho cliente. Por defecto, el cliente HTTP no acepta las cookies provenientes del servidor. Si se especifica una ruta a un archivo con la propiedad *CookiesFile*, éstas se activarán y el cliente HTTP se nutrirá de las cookies existentes en dicho archivo para devolver información al servidor. Si la propiedad *UpdateCookies* se sitúa a TRUE, se permitirá el acceso de escritura al fichero de cookies, con lo cual las nuevas cookies se guardarán entre ejecución y ejecución del programa.

Este componente se encuentra en desarrollo y futuras versiones contemplarán el uso de SSL y certificados.

6. 8 Protocolo FTP

Al igual que el protocolo HTTP, FTP se encuentra muy extendido a lo largo de Internet y está diseñado específicamente para la transmisión y recepción de ficheros. La estructura lógica de un servidor FTP es muy próxima a la de un sistema de archivos locales: existen una serie de carpetas con estructura arbórea y dentro de cada carpeta existen archivos.

El soporte de FTP en Gambas es actualmente muy sencillo y también bastante limitado. Permite, esencialmente, *subir* y *bajar* archivos del servidor. Con una sintaxis

muy similar a la del cliente HTTP, tendremos que indicar la URL deseada y llamar a los métodos *Get()* o *Put()*.

Al igual que en el cliente HTTP, se permite la gestión de nombre de usuario y acceso al servidor.

Futuras versiones de este cliente darán acceso a listados de carpetas y comandos personalizados. No obstante, como se ha indicado, ahora es una clase muy limitada y puede plantearse como alternativa el uso directo de utilidades como *ftp*, *curl* o *wget* para aplicaciones complejas que necesiten acceso a servidores FTP, dado que la gestión de procesos externos es muy sencilla con Gambas.