CD INCLUIDO
APLICACIONES, HERRAMIENTAS...

ERA REVISTA DE PROGRAMACIÓN EN CASTELLANO

SUPROGRAMADORES

AÑO VII. 2.ª ÉPOCA • Nº 79 • UNA PUBLICACIÓN DE: REVISTAS PROFESIONALES S.L. • Precio: 995 Ptas. 5,98 € (España) (IVA incluido)

PRUEBA COMPARATIVA

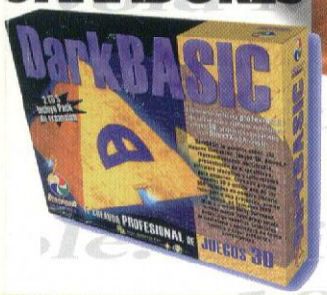
JAVA CONTRA C#

DESARROLLO:

COLABORACIÓN EN RED

WINDOWS: CREACIÓN DE CONTROLADORES

DARKBASIC: LOS 20 GANADORES



SERVIDOR WEB CON JAVA, CORREO ELECTRÓNICO: SMTP, CONSULTORIO...

Correo electrónico: SMPT

JOSÉ LUIS BLANCO CLARACO.

En este artículo se expondrá el funcionamiento del protocolo *SMTP*, usado para el envío de correo electrónico, describiendo además algunas de sus extensiones más interesantes, y su implementación en *Visual C++* con *sockets* asíncronos.

CORREO ELECTRÓNICO

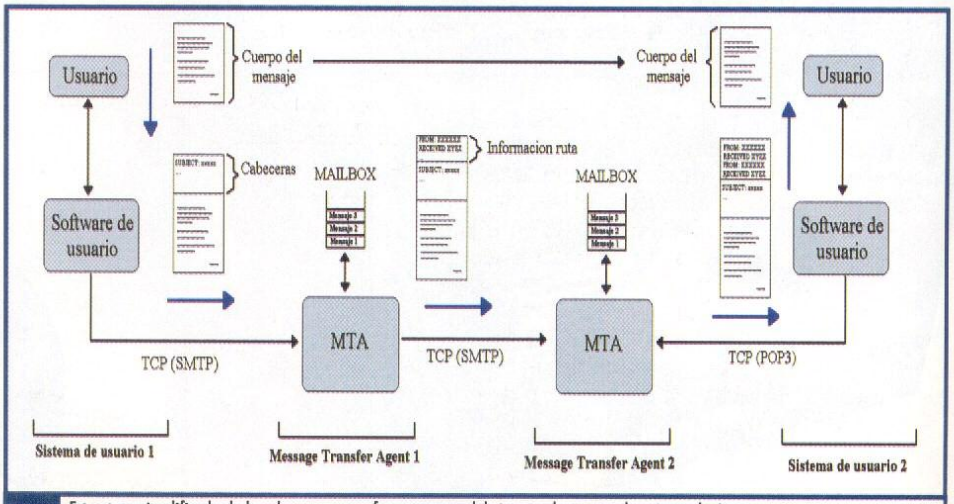
Sin duda, el correo electrónico es uno de los más usados y conocidos servicios de *Internet*, ya que permite el envío casi instantáneo de texto, imágenes o ficheros a un usuario en cualquier lugar del mundo. En este artículo veremos los elementos que hacen posible esto, analizaremos el protocolo *SMTP* y los conceptos de *Windows Sockets* necesarios para poder implementar en *Visual C++* un pequeño cliente de correo.

ELEMENTOS DEL SISTEMA

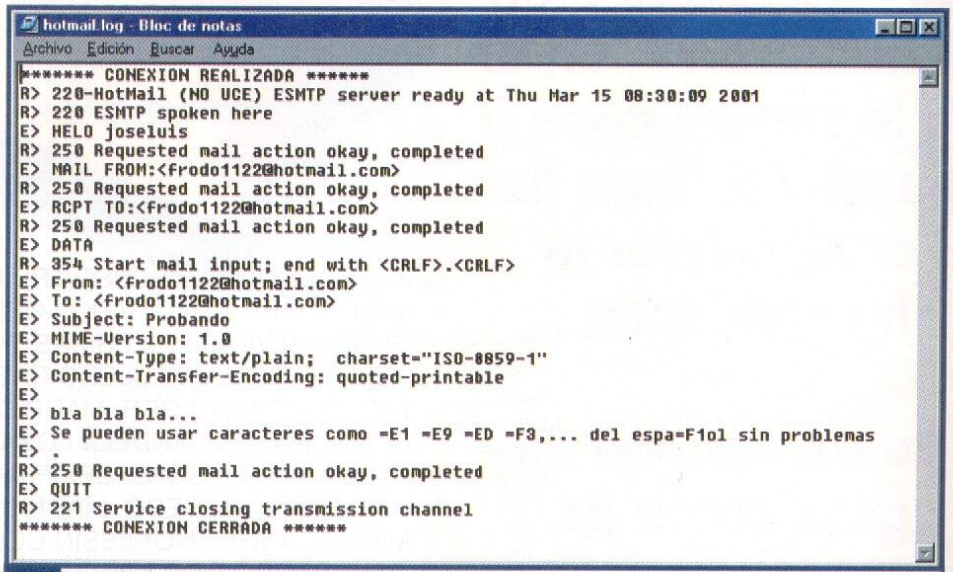
Un usuario que quiere enviar un mensaje no hace uso directo de los protocolos de envío de correo, sino que usa un *software de usuario* (*user agent software*). Estas aplicaciones (como *Outlook Express*) son las que conectan con los *MTA* (*Message Transfer Agent*, servidores de correo) que distribuyen los mensajes para que lleguen al destino. Para realizar la transmisión de mensajes desde el cliente del usuario hasta el *MTA*, el cliente del usuario realiza una conexión *TCP* al *MTA* al puerto número 25. Una vez que la conexión queda establecida, el servidor y el cliente usan el protocolo *SMTP* (descrito más abajo) para la transferencia del mensaje. Ahora, si el destinatario del correo no está en el mismo dominio de este *MTA*, este conectará con otro *MTA* (usando nuevamente *SMTP* en *TCP/25*) y así sucesivamente, hasta que se llegue al dominio final, donde el *MTA* guardará el mensaje en el *mailbox* del usuario. Los *MTA* no avisan a los usuarios de que tienen correo nuevo. Para que el usuario pueda recibir sus mensajes, el *software de usuario* debe pedir explícitamente que se le envíen éstos. La petición de correo se realiza según el protocolo *POP3*.

SMTP

El protocolo *SMTP* define las normas de la conversación entre el cliente y el servidor de correo. Aquí solo vamos a ver sus características más importantes. El diálogo entre cliente y servidor transcurre así: El cliente envía comandos, y el servidor responde con un código numérico junto con un texto explicativo. A continuación se describen los comandos más usuales:



1 Estructura simplificada de los elementos que forman parte del sistema de correo electrónico de *Internet*.



2 Log realizado por el programa cliente. Esta es una secuencia típica de comandos usada para enviar un mensaje.

- *HELO <dominio>*. Le indica al servidor que va a comenzar la transacción de correo, y además le informa de la identidad de la maquina cliente.
- *MAIL FROM:<direccion>* Indica la dirección de correo electrónico del usuario que envía el mensaje.
- *RCPT TO:<direccion>* Indica el destinatario del mensaje.
- *DATA* Con este comando se indica que se van a enviar los datos correspondientes al mensaje: cabeceras, cuerpo,... Para marcar el fin del mensaje se envía un *<CRLF>.<CRLF>* (*"\r\n.\r\n"*).
- *QUIT* Al enviar este comando, el servidor cierra el canal de comunicación y prepara el mensaje para enviarlo al destinatario.

En cuanto a las respuestas del servidor, comienzan con un código numérico de tres cifras. La cifra más importante es la primera, que indica el éxito o error

Código fuente del mensaje

```
From frodo1122@hotmail.com Thu Mar 15 08:31:47 2001
Received: from [62.82.123.109] by hotmail.com (3.2) with ESMTP id MHotMailBC7A39110076400
From: <frodo1122@hotmail.com>
To: <frodo1122@hotmail.com>
Subject: Probando
MIME-Version: 1.0
Content-Type: text/plain;  charset="ISO-8859-1"
Content-Transfer-Encoding: quoted-printable

bla bla bla...
Se pueden usar caracteres como =E1 =E9 =ED =F3,... del espa=F1ol sin problemas
```

3 El código recibido en *Outlook* correspondiente al mensaje enviado que se ve en la Figura 2.

Bibliografía:

RFC 821 Simple Mail Transfer Protocol (1982)

RFC 1225 Post Office Protocol-Version 3 (1991)

RFC 822 Standard for the format of ARPA Internet Text Messages (1982)

RFC 1521 MIME, Multipurpose Internet Mail Extensions (1993)

en el comando anterior. Las demás cifras permiten especificar más exactamente el resultado de los comandos. Además del código numérico, el servidor añade una cadena *user friendly*, que puede variar de una implementación del servidor a otra. Se prevé el uso de respuestas multilínea: si el cuarto carácter de una línea es un espacio, la línea es la última, y si es un guión le siguen más líneas correspondientes a la misma respuesta. Como ejemplo, en la Figura 2 se muestra un *log* realizado por el cliente. Las líneas que comienzan con *"E>* " son las enviadas por el cliente, y por *"R>* " las recibidas del servidor.

MEJORAS A SMTP: CABECERAS

El protocolo *SMTP* es muy antiguo (1982), y solo especifica las normas para transferencia de texto plano *ASCII*(7 *bits*). Como todos sabemos, actualmente el contenido de los mensajes no se limita a mensajes de texto sin formato, sino que se usa texto de diferentes formatos, documentos *HTML*, ficheros adjuntos, imágenes, vídeo,... Esto se puede conseguir con el uso de cabeceras en el mensaje. *MIME* representa la extensión más importante a las capacidades multimedia del correo electrónico, mediante la adición de cabeceras extra. Además de campos de información estándar sobre el mensaje (p.ej. el *Asunto*), *MIME* describe el formato del mensaje, la codificación de caracteres usada...

Las cabeceras deben estar al principio del mensaje, y separadas del texto del mensaje por una línea en blanco. Las cabeceras más usuales, las que implementaremos en nuestro cliente son:

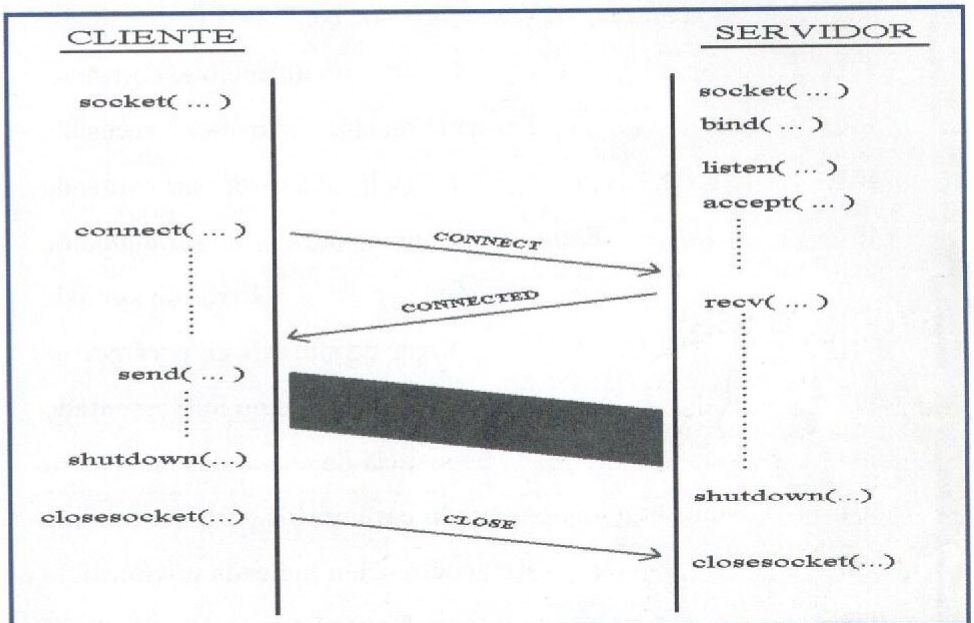
- *From: <texto>*
- *To: <texto>*

Los nombres del emisor y receptor del mensaje.

- *Subject: <texto>*

El asunto del mensaje. Facilita la búsqueda de mensajes importantes entre todos los mensajes recibidos.

- *MIME-Version: <version>*



5 Proceso de conexión usando *sockets*.

Debe ser 1.0. Indica que se usan cabeceras MIME. Nosotros usaremos estas:

Content-Type: text/plain; charset="ISO-8859-1"

Content-Transfer-Encoding: quoted-printable

Para poder usar en el mensaje los caracteres especiales del español (tildes, la ñ...) que no se incluyen en el *ascii* estándar de 7 *bits*. Para eso, se sustituyen estos caracteres antes de enviarlos por *=XX*, donde *XX* es el código *ascii* en hexadecimal. Para más detalles ver el código fuente en el CD.

WINDOWS SOCKETS

Los *sockets* nacieron como una interfaz para programación de aplicaciones que estén conectadas a una red *TCP/IP*, como *Internet*, que originalmente fue desarrollada bajo *UNIX* en la década de los 80. La versión para sistemas *Windows* se llama *Windows Sockets* o *Winsock*, y tiene un gran parecido con su antecesor. En nuestra aplicación, vamos a usar la versión 2.0 de *Winsock*.

SOCKETS

Un *socket* es la terminación de un enlace entre dos procesos en máquinas remotas. El uso más común de los *sockets* se da sobre redes *TCP/IP*, y por lo tanto, podemos usar dos tipos de *sockets*, orientados a conexión, usando *TCP* o sin conexión, usando *UDP*. A lo largo de este artículo sólo nos vamos a centrar en los orientados a conexión, por sus conocidas ventajas en la mayoría de las aplicaciones. Además, el protocolo que nos interesa, *SMTP*, está definido para *TCP*.

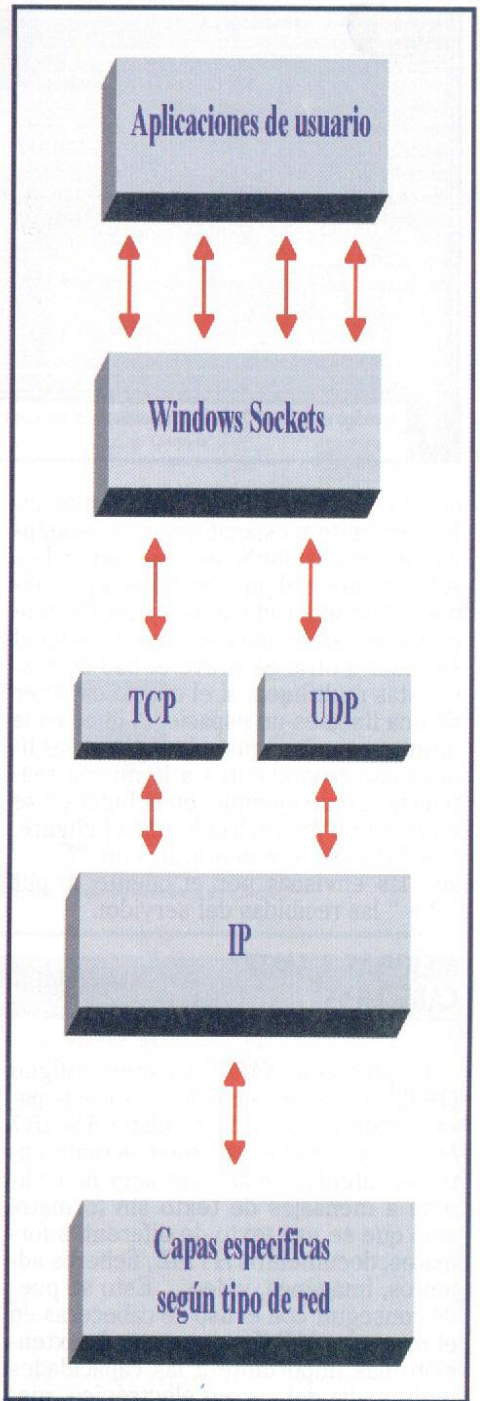
En principio, los *sockets* se diseñaron desde su primera versión bajo *UNIX* siguiendo una analogía con el acceso a ficheros. Un *socket* se tiene que abrir antes de usarse. Después se realizan operaciones de lectura y escritura, y finalmente se cierra. Sin embargo, surgen varios problemas que no aparecen en el manejo de ficheros: no se pueden leer datos de un *socket* hasta que no lleguen desde el otro extremo, el *socket* lo puede cerrar la máquina remota,... Más

TABLA 1. Códigos de respuesta del servidor

Código	Significado
211	Se muestra el estado del sistema
214	Mensaje de ayuda como respuesta a HELP
220	Servidor preparado (primera cadena transmitida)
221	Cerrando canal de transmisión: Respuesta a QUIT
250	*Se completó el último comando*
251	El usuario no es local. Se reenviará
354	Empieza la entrada de texto. Termina con <CRLF>.<CRLF>
421	Servidor no disponible, cerrando.
450	Error: Dirección indicada no accesible
451	Error local al procesar comando
452	Error: Sin recursos para procesar comando
500	Error de sintaxis
501	Error de sintaxis en parámetros
502	Comando no implementado
503	Secuencia de comandos incorrecta
504	No se han implementado parámetros para el comando
550	Error: dirección indicada no accesible
551	Error: El usuario no es local
552	Se sobrepasó el tamaño máximo
553	Error con la dirección de correo
554	La transacción falló

TABLA 2. Funciones del API Winsock

Función	¿Bloquea?	Descripción
WSAStartup		Inicia la DLL de WinSock
WSACleanup		Libera los recursos de WinSock
socket		Crea un socket
setsockopt		Configura socket
bind		Asigna al socket una dirección local
listen		Pone al socket en modo esperar conexión
accept	X	Acepta conexión entrante
connect	X	Conecta con un socket remoto
closesocket	X	Libera el socket
htonl		Byte order del host al de la red. P.ej. para dar el número de puerto TCP
shutdown		Cierra en parte una conexión
gethostbyname	X	Permite resolución DNS
WSAAsyncSelect		Ver texto del artículo



4 Situación de *Windows Sockets* en la torre de protocolos.

abajo veremos cómo se pueden manejar estas situaciones.

Una característica esencial de los *sockets* es que están claramente orientados a la estructura cliente-servidor: uno de los dos extremos es el que inicia la conexión, conectándose al otro extremo, que previamente estaba "escuchando" en un determinado puerto. Por lo tanto, existen dos formas de usar un *socket*: como cliente, o como servidor. Un proceso típico se muestra en la figura 5, junto con los nombres de las funciones usadas. En el Listado 1 se incluye la función de nuestro cliente de ejemplo que se encarga de conectar con el servidor asignado.

SOCKETS NO BLOQUEANTES ASÍNCRONOS

Consideremos el siguiente caso típico: el programa en uno de los extremos de un *socket* envía una petición al otro extremo, y espera su respuesta. Un fragmento de código que puede hacer esto es:

```
send( hsocket, lpbuffer, nbytes , 0);
nBytes=recv(                    hSocket,
lpRecibido,nMaxBytes, 0);
```

Con la primera línea se envía la petición, y con la segunda se recoge la respuesta del otro extremo. Pero por defecto los *sockets* creados con *WinSock* son *bloqueantes*, lo que quiere decir que una llamada a la función *recv* no devolverá el control hasta que no lleguen datos. Además si se tiene una aplicación con multitud de *sockets* conectados a multitud de destinos distintos, esto hace que se pierda tiempo que se puede usar con otros *sockets*. Además, el servidor podría no responder por cualquier motivo. Esto es un serio problema, que se puede solucionar con las funciones *Select* o con *WSAAsynSelect*. *select* es una función más antigua que funciona pasándole tres listas de *sockets* a los cuales se les comprueba si les han llegado datos, si han terminado sus transmisiones y si han sufrido errores. Sin embargo, la función *WSAAsyncSelect* introducida con *WinSock 1.1* introduce el uso del sistema de mensajes de *Windows* para notificación de los eventos que nos interesen. En la Tabla 2 se muestran las funciones más usuales y se indica si son o no *bloqueantes*.

USO DE WSAAsynSelect

La declaración de la función y sus parámetros son:

int *WSAAsyncSelect* (*SOCKET s, HWND hWnd,unsigned int wMsg,long lEvent);*

- *s*: El *socket* que se quiere hacer *no bloqueante* asíncrono.
- *hWnd*: La ventana a la cuál se van a enviar los mensajes de notificación.
- *wMsg*: El valor numérico del mensaje que se enviará.
- *lEvent*: Indica qué eventos se monitorizan. Los posibles valores están en la Tabla 3.

Para poder procesar los mensajes, hay que modificar la función *WndProc* correspondiente a la ventana indicada. Cuando llegue un mensaje, el parámetro *lParam* contendrá el valor *FD_X* correspondiente al evento ocurrido.

IMPLEMENTACIÓN DEL CLIENTE

El listado completo se encuentra en el *CD*, así que aquí sólo vamos a comentar las partes más importantes.

TABLA 3. Eventos que se pueden monitorizar con WSAAsynSelect

Valor	Avisar cuando...
FD_READ	Lleguen datos al socket
FD_WRITE	El socket queda libre para enviar datos
FD_OOB	Lleguen datos fuera de banda (p.ej.en X.25)
FD_ACCEPT	Preparado para aceptar conexión entrante
FD_CONNECT	Ha finalizado el proceso de conexión
FD_CLOSE	La conexión se cierra
FD_QOS	Cambie el Quality of service del socket
FD_GROUP_QOS	Cambie el QOS de un grupo de sockets

LISTADO 1. La función Conecta Servidor resumida . (Se ha eliminado la parte de detección de errores)

```
SOCKET  CClienteDlg::ConectarServidor(char *lpDireccion,int nPuertoTCP)
{
        SOCKET                  hSock=INVALID_SOCKET;
        LPHOSTENT                       lpHE;
        SOCKADDR_IN             Dir;
        int                             opcion=1;
        unsigned long           IP;

        // ¿ Es una direccion IP?
        if (INADDR_NONE== (IP=inet_addr( lpDireccion )) )
        {
                // No es IP: usar DNS:
                if (!( lpHE=gethostbyname(lpDireccion) ) )    {
                        MessageBox(lpDireccion,"No se encuentra el servidor de
correo:",MB_OK);
                        return hSock;
                }
                else
                {       // Nombre DNS resuelto
                        Dir.sin_addr = *((LPIN_ADDR)*lpHE->h_addr_list);
                }
        }
        else
                // Usar direccion IP devuelta
                Dir.sin_addr = *((LPIN_ADDR)&IP);

        // Crear socket:
        hSock= socket( AF_INET,SOCK_STREAM,IPPROTO_TCP );

        // No esperar a envios al cerrar el socket
        res=setsockopt ( hSock,SOL_SOCKET,SO_DONTLINGER,
                (char*)&opcion,sizeof(int) );

        // Preparar señalizacion de eventos:
        // Conexion , cierre y llegada de datos
        WSAAsyncSelect  ( hSock,m_hWnd,WM_AVISO,  FD_CLOSE|FD_CONNECT|  FD_READ
);

        // Rellenar la estructura de la direccion:
        Dir.sin_family = AF_INET;
        Dir.sin_port = htons( nPuertoTCP );

        connect ( hSock,(PSOCKADDR)&Dir, sizeof(SOCKADDR_IN) );

        return hSock;
}
```

Una forma de enviar los datos necesarios al servidor de correo podría consistir en algo así:

```
i=0; error = false;
while ( error==false && i<nComandos
)       {
```

LISTADO 2. La función CClienteDLG: :WindowProc

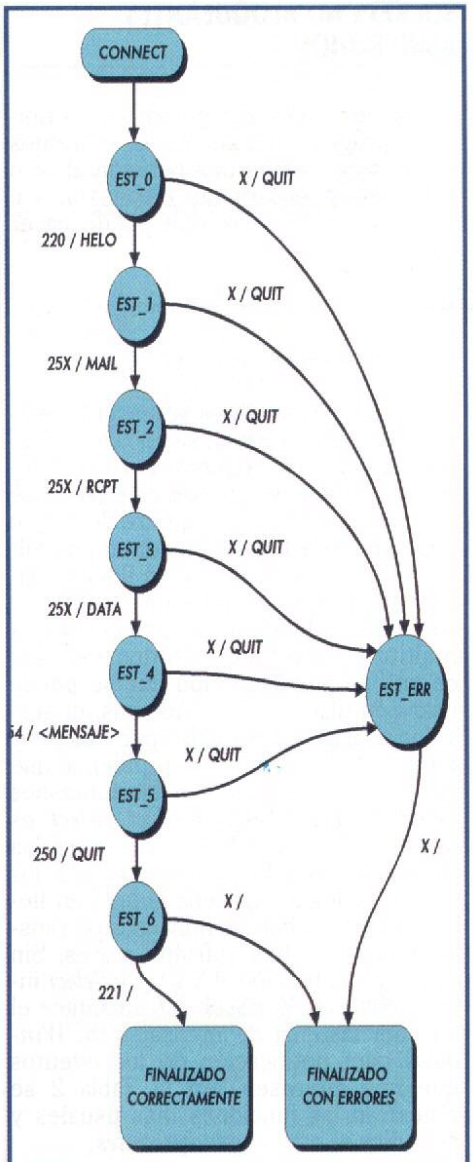
```
LRESULT CClienteDlg::WindowProc(UINT message, WPARAM wParam, LPARAM lParam)
{
        char            bufRecep[MAX_LON];
        int             nBytesRec;
        char            auxStr[MAX_LON];
        int             pos;
        if (message==WM_AVISO)
        {
                switch (lParam)
                {
                case FD_CONNECT:
                 Log("******* CONEXION REALIZADA *******",0);
                 break;

                case FD_CLOSE:
 closesocket(hSocket);  // Liberar socket
 m_btnEnvio.EnableWindow ( true );
                 m_btnAbortar.EnableWindow ( false );
                 Log("******* CONEXION CERRADA *******",0);
                 Estado= EST_DESCONECTADO;
                 hSocket= INVALID_SOCKET;
                 break;

                case FD_READ:
                 nBytesRec=recv ( hSocket,bufRecep,MAX_LON,0);
                 bufRecep[nBytesRec]=0; // Es una cadena
                 // Pueden recibirse datos de antes de cerrar:
                 if (Estado==EST_DESCONECTADO) break;
                 // Procesar cada una de las lineas por separado:
                 pos=0;
                 while ( SiguienteLinea(bufRecep,&pos,auxStr ) )
                 {
                        // Mostrar la cadena en el log:
                        Log( auxStr ,2);
                        // Procesarla
                        MaquinaEstados( auxStr );
                 }
                 break;

                }       // Fin del switch
                return 0;
        }
        return CDialog::WindowProc(message, wParam, lParam);
}
```

```
                send(   ...   Comando[i]
... );
                recv( ... respuesta ...
);
                if ( ... ) error= true
                i++;
  }
```

Un código muy sencillo, pero con el problema de que la función *recv* es *bloqueante* por defecto, cosa no deseable. Nosotros usaremos *sockets* asíncronos no *bloqueantes*, y llamaremos a una función cada vez que el servidor responda, procesando la respuesta y enviando el siguiente comando.

MÁQUINA DE ESTADOS

Los programas que se comunican usando protocolos se pueden ver como máquinas de estados. Nosotros vamos a usar la máquina de estados que se muestra en la Figura 6. En la imagen se representan con círculos los diferentes estados, con el nombre de cada estado. Las flechas indican transiciones, y van acompañadas de un texto con el formato *A/B*. Significa que si se recibe del servidor el código *A*, se sigue la flecha y además se responde al servidor con *B*. Una *X* en un código significa que esa cifra no es significativa. Cuando salgan dos flechas de un estado, la marcada con *X* se sigue si no se cumple la otra condición. El lector puede comparar la máquina de estados con el *log* de la Figura 2 para comprobar la secuencia de los comandos. La implementación de la máquina está en *CClienteDlg::MaquinaEstados(char *lpRec)* en el listado del CD.



6 La máquina de estados que usaremos para nuestro cliente de correo.

PROCESADO DE MENSAJES

La función *WindowProc* de cada clase *CWnd* del *MFC* sirve para procesar los mensajes de interés antes de que lleguen al procesado estándar de la ventana. En nuestro cliente, esta función procesa los mensajes generados de forma asíncrona: los eventos de apertura, cierre y llegada de datos. Así, se encarga de llamar a la máquina de estados y de avisar de la apertura y cierre del *socket*. Además, se encarga de separar en varias líneas las respuestas del servidor que sean multilínea, como se puede ver en el Listado 2.

CONCLUSIÓN

Hemos expuesto herramientas como máquinas de estados y los *sockets* que son útiles para implementar, no sólo este protocolo, sino cualquier otro protocolo de comunicaciones sobre *Internet*. Si el lector desea más información sobre el protocolo *SMTP* o cualquier otro, puede visitar la página *http://www.rfc-editor.org/*