Servidor Gopher

Jesús Manuel Muñoz Mazuecos 6270974
Universidad de Castilla - La Mancha

Índice General

1. Arquitectura

Se trata de una aplicación para UNIX escrita en lenguaje C sobre un servidor concurrente mixto del servicio estándar Gopher [RFC 1436] que usa el interfaz de programación Socket BSD.
La aplicación se divide en tres módulos interrelacionados entre sí. El módulo principal se llama gophermxd.c y es el que realiza el cometido principal de la aplicación. Éste está preparado para aceptar conexiones con los clientes enviándoles la información correspondiente de acuerdo con el servicio gopher. Al mismo tiempo este módulo es dependiente del módulo SocketPasivoTCP.c cuyo cometido principal es vincular y asignar un socket TCP. Estos dos módulos dependen de otro módulo llamado notiferror.c el cual se utiliza como un manejador de excepciones que puedan generar los otros módulos.

1.1 Alternativas de diseño

En esta sección se van a describir alternativas de diseño que han ido surgiendo durante el desarrollo del problema.
La primera alternativa de diseño contemplada fue comprobar la existencia del path emitido por el cliente sobre el árbol de directorios del servidor, esta fue una decisión en vano ya que el servidor va a enviar un archivo sólo si este se puede localizar mediante el identificador de archivo recibido. Si ese archivo no se encuentra, simplemente es problema del cliente, porque el servidor siempre va a especificar de manera correcta en sus ficheros de contenido los identificadores para obtener el contenido de un archivo o directorio. Por lo que al final no contemplé esta alternativa también porque esta alternativa suponía una carga de trabajo innecesaria.

Otra de las alternativas de diseño pensada para la aplicación trata sobre los indetificadores de archivo o directorio que utiliza el cliente para explorar archivos o directorios. La forma de los identificadores de archivo en mi servidor, queda definida según el siguiente ejemplo:

0dir/subdir/file.txt

Si nos fijamos el primer carácter es el identificador del tipo de archivo, en este caso es cero porque se trata de un archivo de texto, posteriormente le sigue el path del archivo. En este caso se trata del archivo file.txt que está en el directorio *subdir*, que éste a su vez se encuentra dentro del directorio *dir* colgando este último del directorio raíz del servidor. La decisión de indicar el tipo de archivo antes del path fue aplicada mayormente para librar de una carga adicional de trabajo al servidor. Simplemente comprobando el primer carácter de la cadena emitida por el cliente ya sabemos qué tipo de archivo se va a abrir y como tratarlo para enviar su contenido. Una desventaja sobre esta alternativa podría ser la estética del identificador de archivo, ya que se puede decir que quedaría mucho más elegante y formal no incluir el carácter que identifica el tipo de archivo. Aun así se ha decidido aceptar esta alternativa

Y por último, una alternativa que se ha considerado útil ha sido la de definir una constante con el nombre del fichero de contenido de un directorio, ya que este va a tener el mismo nombre para cada directorio del servidor. En nuestro caso, el fichero de contenido se llama index. Por eso si el cliente solicita la información de un directorio, simplemente se enviaría el contenido de index en el directorio solicitado. Se definió una constante por si en algún momento de la vida de la aplicación se decide cambiar el nombre de los ficheros de contenido.

2. Especificación de módulos

En esta sección se va a proceder a la explicación detallada de cada uno de los módulos utilizados en la aplicación junto con las funciones contenidas en cada módulo. Primero se describirá el módulo principal de primer nivel, y luego los demás de segundo nivel.

El módulo gophermxd.c

El módulo *gophermxd.c* realiza el cometido principal de la aplicación y está compuesto por la función main, otra función llamada gophermxd y una función llamada listar que se utiliza para enviar el contenido de un archivo.

Función main

int main(int argc, char *argv[]);

En la función *main* se toma como argumento de la línea de órdenes el puerto del servidor en el cual va a estar a la escucha nuestra aplicación.
Nada más entrar en la función *main* se comprueba si el numero de argumentos es el correcto, si no es así el programa lanzará una excepción. Posteriormente se crea un socket maestro, se obtiene el número máximo de ficheros que un proceso puede tener abiertos, se asocia el socket maestro al conjunto de descriptores activos, permaneciendo este socket a la escucha.
Después el programa entra en un bucle infinito que va a estar en espera hasta que ocurra algún cambio de estado en un descriptor. Lo primero que hace el bucle es copiar nuestro conjunto de descriptores activos a un conjunto de descriptotes de lectura, posteriormente se comprueba si alguno de estos descriptores de lectura ha sufrido un cambio de estado. Si se ha producido un cambio de estado y ha sido en el descriptor del socket maestro, significa que hay que aceptar conexión entrante creando un socket esclavo y asignándolo al conjunto de descriptores activos (En la siguiente iteración del bucle el descriptor del socket esclavo se copiará al conjunto de descriptores de lectura y se esperará hasta que se produzca algún cambio de estado en él o en los demás descriptores), si no, probablemente se trata de un cambio de estado en un socket esclavo que seguramente será utilizado para dar servicio a un cliente gopher. Ocurrido esto, se comprueba si el descriptor de ese socket esclavo está dentro del conjunto de descriptores de lectura y además que sea distinto del socket maestro. Si estas condiciones se cumplen se llama a la función *gophermxd ( )* que es la que da el servicio. Una vez que se da el servicio se cierra la conexión con el cliente. Y sucesivamente se van aceptando clientes de manera concurrente que no bloquean a los que entraron después de ellos si estos todavía no han cerrado la conexión ya que la función *select* del lenguaje C sólo devuelve los descriptores en los que se ha producido algún evento.

Función gophermxd

int gophermxd(int s);

Parámetros:
int s Descriptor del socket.

Valor devuelto: 0 en caso de éxito y -1 en caso de error
Cometido: Interpretar el identificador de archivo enviado por el cliente

Una vez que se ha llamado a la función gophermxd ( ), lo primero que se hace es guardar en una cadena de caracteres llamada raíz el directorio raíz del servidor. En nuestro caso el directorio raíz del servidor es un directorio que se encuentra en el mismo directorio que el archivo objeto de nuestro programa. Así con la función de biblioteca getcwd ( ) obtenemos el directorio actual de trabajo. A este directorio raíz le tengo que concatenar el nombre del directorio que figura como directorio raíz del servidor, para ello se ha definido una constante simbólica. Después de preparar el directorio raíz del servidor esperamos a que el cliente nos envíe un identificador de archivo o directorio. Posteriormente extraemos el primer carácter del identificador para saber que tipo de archivo desea el cliente (Ver sección alternativas de diseño). Si se trata de un directorio simplemente concatenamos el nombre del archivo contenido al identificador enviado por el cliente. Finalmente recogemos el tamaño en bytes del archivo que solicita en cliente y si este supera un tamaño especificado por una constante simbólica, el contenido del fichero será enviado por un proceso hijo, sino, el contenido del archivo será enviado por el proceso padre. Si todo ha transcurrido de manera correcta se devuelve el valor cero a la función invocadora.

Función listar

void listar(char *path, int s, char opc);

Parámetros:
char path: Descriptor del socket.
int s: Descriptor del soket
char opc: carácter que inidica el tipo de archivo que se va a examinar

Cometido: enviar el contenido del archivo especificado en path al cliente.

Para enviar el contenido de un archivo o directorio al cliente se ha definido la función listar, que según el tipo de archivo que se va a tratar, abre el archivo de una manera u otra, realiza un tratamiento especializado para el tipo de archivo que se va a enviar y envia linea por linea el contenido del archivo hasta que llega a fin de archivo. También si se produce algún error durante el envío de una línea de un archivo se cerrará la conexión con el cliente. Notar que los archivos binarios se van a abrir en tipo lectura binaria.

El módulo SocketPasivo.c

Este módulo consta de una función llamada *SocketPasivoTCP,* la cual crea un socket lo vincula y posteriormente permanece a la escucha. La definición de la función es la siguiente:

Función SocketPasivoTCP

int SocketPasivoTCP(const char *servicio, int longcola);

Parámetros:

const char *servicio: Es el puerto o nombre del servicio.
int longcola: Un entero que indica la longitud de la cola de espera.

Valor devuelto: en caso de éxito devuelve el descriptor del socket asociado y en caso de error devuelve un entero menor que cero.

Cometido: Asigna y vincula un socket usando TCP.

El módulo notiferror.c

Este es un módulo que es utilizado para notificar los errores que puedan lanzar funciones contenidas en los módulos anteriores.

void NotifyError(const char *message, int error);

Parámetros:

const char *message: mensaje de error
int error: número del error

Cometido: Notificar un error.

3. Conclusiones y propuestas

En esencia, la práctica nos ilustra el funcionamiento de un servidor de Internet y como envía el contenido de los ficheros que posee en su directorio raíz a los determinados clientes. Pudiendo también ese mismo servidor tener referencias a archivos situados en otros servidores conformando así un sistema distribuido sobre el que se ejecuta el servicio gopher. En cuanto a la aplicación descrita en este documento, una mejora significativa hubiera sido crear un mecanismo mucho más especializado para tratar de una manera más personalizada, archivos que no sean de texto.
Una prueba que quedaría por hacer es instalar la aplicación en servidores pertenecientes a redes distintas y observar su funcionamiento.

Apéndice A. Organización del código

El código está organizado de manera que hay un directorio para guardar los ficheros objeto generados para luego enlazarlos, que cuelga del directorio gophermxd en el que se encuentran todos los módulos implementados y archivos cabecera. Para compilar y enlazar los distintos módulos se necesita abrir una terminal, situarse en el directorio gophermxd y escribir la orden make, debido a la decisión de definir un archivo makefile que se describirá en el Apéndice B.
El ejecutable se creará en el directorio gophermxd y se ejecuta de la forma.

$./gophermxd <numero de puerto>

Organización de los archivos de código fuente y objeto en el directorio

gophermxd/

bin
gophermxdEjecutable)
gophermxd.c
makefile
notiferror.c
notiferror.h
SocketPasivoTCP.c
SocketPasivoTCP.h

gophermxd/bin:

gophermxd.o
notiferror.o
SocketPasivoTCP.o

Apéndice B. Código fuente

En este apéndice se van a describir con detalle cada uno de los módulos que componen la aplicación. Primero que se va a detallar el contenido del fichero makefile y posteriormente el código fuente de cada una de las funciones en los distintos módulos.

Módulos de la aplicación

notiferror.c
SocketPasivo.c
gophermxd.c

Archivos auxiliares para la compilación, enlace y ejecución

notiferror.h
SocketPasivo.h

Contenido del fichero makefile

```
CFLAGS= -c -Wall
LNKFLAGS = -Wall
CC = gcc

DIREXEC = ./
DIRBIN = bin/
DIRLIB = ./
DIRSRC = ./

EXEC=gophermxd

all: $(EXEC)

$(EXEC) : gophermxd.o SocketPasivoTCP.o notiferror.o
      $(CC) $(LNKFLAGS) -o $(DIREXEC)$(EXEC) $(DIRBIN)gophermxd.o
$(DIRBIN)SocketPasivoTCP.o $(DIRBIN)notiferror.o

gophermxd.o : $(DIRSRC)gophermxd.c $(DIRLIB)notiferror.h
$(DIRLIB)SocketPasivoTCP.h
      $(CC) $(CFLAGS) -I $(DIRLIB) -o $(DIRBIN)gophermxd.o $(DIRSRC)gophermxd.c

SocketPasivoTCP.o : $(DIRSRC)SocketPasivoTCP.c $(DIRLIB)SocketPasivoTCP.h
      $(CC) $(CFLAGS) -I $(DIRLIB) -o $(DIRBIN)SocketPasivoTCP.o
$(DIRSRC)SocketPasivoTCP.c

notiferror.o : $(DIRSRC)notiferror.c $(DIRLIB)notiferror.h
      $(CC) $(CFLAGS) -I $(DIRLIB) -o $(DIRBIN)notiferror.o
$(DIRSRC)notiferror.c
```

El módulo gophermxd.c

Este módulo está compuesto por tres funciones: gophermxd, listar, main.

```
/*****************************************************************************
*
Función gophermxd: Interpretar el identificador de archivo enviado por el
cliente
Parámetros:
      int s Descriptor del socket.

Valor devuelto: 0 en caso de éxito y -1 en caso de error
*****************************************************************************
*/
int gophermxd(int s){

    char buf[MAXLINEA+1];
    char raiz[MAXLINEA+1]; /* = "/home/jesus/AplDistr/gophermxd/serv1"; */
    int cliStringLen;
    struct stat *bufStat;
    char tipo;
    int hijo = 0;
    int pid, estado;

    if(getcwd(raiz, sizeof(raiz)) == NULL ) { /* Obtengo la raiz del mi
                                                 directorio de tarbajo */
        NotifyError("Error al obtener el direcotorio raiz", -1);
        return -1;
    }

    strncat(raiz, servidor, strlen(servidor)); /* Concatena raiz con el nombre
                                                  de la carpeta del servidor que
                                                  va a ser descendiente del
                                                  directorio de trabajo */

    if(strlen(raiz) > MAXLINEA) {
        NotifyError("Path del directorio raiz demasiado largo", -1);
        return -1;
    }

    cliStringLen = read(s, buf, MAXLINEA); /* recibir mensaje del cliente */

    if(cliStringLen < 0) {
        NotifyError("Error al recibir el mensaje", -1);
        return -1;
    }

    if(cliStringLen > MAXLINEA) {
        NotifyError("Línea demasiado larga", -1);
        return -1;
    }

    if(strncmp(buf, "\r\n", 2) == 0) { /* Si se recibe \n \r envio por defecto
el fichero index del dir raiz */
        tipo = DIRECTORIO;
    }
    else {
        buf[cliStringLen-2] = '\0'; /* \r = \0 */
        tipo = buf[0]; /* buf[0] indentifica el tipo de archivo según los
                          index.txt*/
        buf[0] = '/';  /* la posicion cero del buffer la sustituyo por el
                          caracter / para poder concatenar el contenido del
                          buffer con el directorio raiz */
```

```
        strncat(raiz, buf, cliStringLen-1); /* concateno hasta el caracter \0
                                               inclusive */
    }

    if(tipo == DIRECTORIO) { /* si se trata de un diretorio se le concatena el
                                nombre del fichero de contenido */
        strncat(raiz, indice, strlen(indice));
    }

    bufStat = (struct stat *)malloc(sizeof(struct stat));

    if(bufStat == NULL) {
        NotifyError("Error al asignar memoria\n", -1);
        return -1;
    }

    if(stat(raiz, bufStat) == -1) {
        NotifyError("Error en stat\n", -1);
        return -1;
    }

    if(bufStat -> st_mode > N) { /* Si el tamaño fichero es mayor que N el
                                    contenido será enviado por el hijo */
        hijo = 1;
    }

    switch(fork()) {

        case -1:

            NotifyError("Error en el fork\n", -1);
            return -1;
            break;

        case 0:

            if(hijo) {
                listar(raiz, s, tipo);
            }
            exit(0);
            break;

        default:

            pid = wait(&estado); /* Esperar a que finalize el hijo */

            if(!hijo) {
                listar(raiz, s, tipo);
            }
            break;
    }
    return 0;
}
```

```
/*****************************************************************************
*
Función listar:  enviar el contenido del archivo especificado en path al
cliente.
Parámetros:
char path: Descriptor del socket.
int s: Descriptor del soket
char opc: carácter que inidica el tipo de archivo que se va a examinar
*****************************************************************************
*/
void listar(char *path, int s, char opc){
    FILE *fp;
    char linea[MAXLINEA+1];

    switch(opc){

        case ARCHIVO:

            if( (fp = fopen(path, "r")) == NULL) {
                NotifyError("Error al examinar el archivo o directorio o no
                             existe", -1);
            }

            while( fgets(linea, MAXLINEA, fp) != NULL ) {

                if(strncmp(linea, ".\r\n", 3) == 0){
                    strncat(".", linea, strlen(linea));
                }

                if(write(s, linea, strlen(linea)) < 0) {
                    close(s);
                    NotifyError("Error al enviar", -1);
                }
            }

            if(write(s, ".\r\n", 3) < 0) {
                NotifyError("Error al enviar", -1);
            }
            break;

        case DIRECTORIO:

            if( (fp = fopen(path, "r")) == NULL) {
                NotifyError("Error al examinar el archivo o directorio o no
                             existe", -1);
            }

            while( fgets(linea, MAXLINEA, fp) != NULL ) {

                if(write(s, linea, strlen(linea)) < 0) {
                    close(s);
                    NotifyError("Error al enviar", -1);
                }
            }

            if(write(s, ".\r\n", 3) < 0) {
                NotifyError("Error al enviar", -1);
            }
            break;

        case BINARIO:
            if( (fp = fopen(path, "rb")) == NULL) {
                NotifyError("Error al examinar el archivo o directorio o no
                             existe", -1);
```

```
            }

            while( fgets(linea, MAXLINEA, fp) != NULL ) {

                if(write(s, linea, strlen(linea)) < 0) {
                    close(s);
                    NotifyError("Error al enviar", -1);
                }
            }
            break;
    }
}


/*****************************************************************************
Función main:  crea el socket TCP acepta clientes concurrentemente y llama al
               servicio gopher.
Parámetros:
argv[1]: Número de puerto.
*****************************************************************************/

int main(int argc, char *argv[]){

    char    *servicio;       /* servicio */

    struct sockaddr_in fsin; /* direccion del cliente */
    int msock;               /* socket maestro*/
    fd_set  rfds;            /* conjunto de descriptores de lectura */
    fd_set  afds;            /* conjunto de descriptores activos */
    unsigned int    alen;    /* longitud de la direccion */
    int fd, nfds;

    switch (argc) {

        case 2:

            servicio = argv[1];
            break;

        default:

            NotifyError("Uso: <puerto> \n", 0);
            return 1;
    }

    msock = SocketPasivoTCP(servicio, QLEN);

    nfds = getdtablesize(); /* Número máximo de ficheros que un proceso puede
                               tener abiertos */
    FD_ZERO(&afds);         /* vacia el conjunto de descriptores activos */
    FD_SET(msock, &afds);   /* añade el descriptor msock al conjunto de
                               descriptores archivos */

    while (1) {
        memcpy(&rfds, &afds, sizeof(rfds)); /* volcar el conjunto de
                                               descriptores activos en el conjunto de
                                               descriptores de lectura */

        if (select(nfds, &rfds, NULL, NULL, NULL) < 0){ /* Comprueba si se ha
                                                           producido algun descriptor de
                                                       lectura ha cambiado de estado */
            NotifyError("gophermxd: Error en select\n", -1);
            return -1;
```

```
        }

        if (FD_ISSET(msock, &rfds)) { /* si ha cambiado de estado el descriptor
                                         de lectura msock */
            int ssock;

            alen = sizeof(fsin);
            ssock = accept(msock, (struct sockaddr *)&fsin, &alen); /* aceptar
                                                                       conexion */

            if (ssock < 0) {
                NotifyError("gophermxd: Error en accept\n", -1);
                return -1;
            }

            FD_SET(ssock, &afds); /* el socket esclavo se añade al conjunto de
                                     descriptores activos */
        }

        for (fd=0; fd<nfds; ++fd)
            if (fd != msock && FD_ISSET(fd, &rfds)) /* si uno de los
                                                       descriptores cambió de
                                                       estado y no es msock */
                if (gophermxd(fd) == 0) {  /* llamada al servicio */
                    close(fd);             /* cierra la conexion */
                    FD_CLR(fd, &afds);     /* el descriptor se elimina del
                                              conjunto de descriptores activos */
                }
    }
}
```

El módulo SocketPasivoTCP.c

Este módulo está compuesto por la función SocketPasivoTCP

```
/*****************************************************************************
Función SocketPasivoTCP: Asigna y vincula un socket usando TCP.
Parámetros:
      const char *servicio: Es el puerto o nombre del servicio.
      int longcola: Un entero que indica la longitud de la cola de espera.

Valor devuelto: en caso de éxito devuelve el descriptor del socket
            asociado y en caso de error devuelve un entero menor que cero.
*****************************************************************************/
int SocketPasivoTCP(const char *servicio, int longcola)
{
      struct servent    *pse; /* Puntero a descripcion de un servicio */
      struct sockaddr_in sin; /* Direccion IP*/
      int   s;    /* Descriptor del socket*/

      memset(&sin, 0, sizeof(sin));
      sin.sin_family = AF_INET;

    /* Obtiene puerto asociado al servicio */
      if ((pse = getservbyname(servicio, "tcp")) != NULL )
            sin.sin_port = pse->s_port;
      else if ( (sin.sin_port = htons(atoi(servicio))) == 0 ) {
            NotifyError("SocketPasivoTCP: Servicio incorrecto", 0);
            return -1;
      }

    /* direccion IP del servidor */
      sin.sin_addr.s_addr = INADDR_ANY;

    /* Crea el socket */
      if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
            NotifyError("SocketPasivoTCP: Imposible crear el socket", 1);
            return -2;
      }

    /* Vincular el socket local */
      if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) != 0) {
            NotifyError("SocketPasivoTCP: Imposible vincular el socket", 1);
            close(s);
            return -3;
      }
    /* Activar el modo pasivo del socket */
    if (listen(s, longcola) < 0) {
            NotifyError("SocketPasivoTCP: Imposible poner a la escucha el
                        socket", 1);
            return -4;
      }

  return s;
}
```

El modulo notiferror.c

El módulo notiferror.c consta solamente de una función.

```
/**********************************************************************
 Funcion NotifyError: función usada para la notificacion de errores
 Parámetros:
      const char *message: mensaje de error
      int error: número del error
**********************************************************************/
void NotifyError(const char *message, int error)
{
      fprintf(stderr, "%s\n", message);
      if(error) perror("Error");
}
```

Apéndice C. Ejemplos de test y salidas típicas.

A continuación se van a listar una serie de ejemplos de salidas típicas de la aplicación.

En este primer ejemplo se muestra la salida cuando el servidor recibe la cadena <CR><LF>. Que por defecto le mostrará al cliente el contenido del directorio raiz.

```
jesus@linuxR40:~/Escritorio/gophermxd$ telnet localhost 7000
Trying 127.0.0.1...
Connected to linuxR40.
Escape character is '^]'.

0file2.txt      0file2.txt      localhost 7000+
1dir    1dir    localhost 7000+
0file1.txt      0file1.txt      localhost 7000+

Connection closed by foreign host.
```

El siguiente ejemplo consiste en que el servidor muestre el contenido de un subdirectorio inidcado.

```
jesus@linuxR40:~/Escritorio/gophermxd$ telnet localhost 7000
Trying 127.0.0.1...
Connected to linuxR40.
Escape character is '^]'.
1dir
1subdir                 1dir/subdir     localhost 7000      +
0file11.txt             0dir/file11.txt     localhost 7000      +
0file12.txt             0dir/file12.txt localhost 7000+
9bin    9dir/bin    localhost 7000  +

Connection closed by foreign host.
```

Y por último se va a mostrar la salida cuando el cliente pide que se le envie el contenido de un archivo que se encuentra en un directorio especificado.

```
jesus@linuxR40:~/Escritorio/gophermxd$ telnet localhost 7000
Trying 127.0.0.1...
Connected to linuxR40.
Escape character is '^]'.
0dir/file11.txt
esto es file 1 en dir
.
Connection closed by foreign host.
```

Documentación de errores

A continuación se van a listar los errores que puede generar el servicio. Siempre que la aplicación genera una excepción se invoca la función *NotifyError* del módulo notiferror.c. Para notificar al sistema la ocurrencia de una excepción y abortar la ejecución en curso. Esta función escribe en el stderr el mensaje generado por la excepción, posteriormente el sistema operativo puede lanzar un mensaje de error de acuerdo con esa excepción.

Una de las excepciones más comunes surge cuando se trabaja con cadenas de caracteres que tienen una longitud superior a 255 caracteres. Por ejemplo a la hora de obtener el directorio raiz del servidor, si la funcion del Lenguaje c *getcwd* devuelve un path que tiene una longitud superior a 255 caracteres, la aplicación lanzará un mensaje de error notificando que el path es demasiado largo. En este caso se aborta la ejecución en curso del programa.

En cuanto a la recepción del mensaje proveniente del cliente, si recibimos un mensaje cuya longitud es menor que cero, quiere decir que se ha producido un error a la hora de recibir el mensaje. Entonces generamos un mensaje de excepción que indica que ha ocurrido un error al recibir el mensaje, notificamos al sistema la ocurrencia de esta excepción y abortamos la ejecución en curso de la aplicación.

A la hora de abrir archivos, si el cliente especifica un identificador de archivo que no se corresponde con ninguno perteneciente a nuestro servidor, lanzaremos un mensaje de excepción notificando que el archivo no existe o ha habido un error al leerlo, por supuesto se da al sistema constancia de ello y se aborta la ejecución.

Un aspecto importante que se ha considerado es que, si ocurre un error mientras se le está enviando datos al cliente antes se debe cerrar la conexión con el cliente. Posteriormente se notifica al sistema de la ocurrencia de esa excepción y se aborta la ejecución en curso.

Otra excepción que lanza la aplicación ocurre cuando se ejecuta la aplicación con un número de argumentos procedentes de la línea de órdenes incorrecto. Cuando ocurre esto se notifica al sistema de la excepción y se termina la ejecución del programa.

La aplicación también controla todas aquellas excepciones que puedan generar las funciones que asignan memoria dinámica, consultan los atributos de un archivo, aceptan conexiones entrantes y que crean procesos hijos. Notificando al sistema la ocurrencia de esa excepción, adjuntando el mensaje de error correspondiente y por supuesto abortando la ejecución en curso.