ANEXO A. FUNDAMENTOS TEÓRICOS DE LLAMADAS AL SISTEMA: RESUMEN API C DE RED.

En este anexo se realiza una descripción teórica de las llamadas al sistema que componen la API C de red y que hemos utilizado para la programación de las aplicaciones CLIENTE y SERVIDOR. No se trata de un manual de referencia de la API C de red ya que no se incluyen todas las funciones y llamadas al sistema de dicha API. Para una información más detallada se recomienda acudir a la bibliografía consultada referente a la API C de red [1][2][3].

1. Resumen API C de Red.

A continuación explicamos los elementos de la API C que hemos empleado.

SOCKET

socket - crea un extremo de una comunicación.

SINOPSIS

```
#include <sys/types.h>

#include <sys/socket.h>

int socket(int dominio, int tipo, int protocolo);
```

DESCRIPCIÓN

Socket crea un extremo de una comunicación y devuelve un descriptor. El parámetro dominio especifica un dominio de comunicaciones. Esto selecciona la familia de protocolo que se usará para la comunicación (PF_INET Protocolos de Internet IPv4).El conector tiene el tipo indicado, que especifica la semántica de la comunicación (SOCK_STREAM proporciona flujos de bytes basados en una conexión bidireccional secuenciada, confiable. Se puede admitir un mecanismo de transmisión de datos fuera de banda. El protocolo especifica un protocolo particular para ser usado

con el conector.

VALOR DEVUELTO

Se devuelve un -1 si ocurre un error; en otro caso el valor devuelto es un descriptor para referenciar el conector.

BIND

bind - enlaza un nombre a un conector (socket)

SINOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *my_addr, socklen_t
addrlen);
```

DESCRIPCIÓN

bind da al conector sockfd la dirección local my_addr. my_addr tiene una longitud de addrlen bytes. Tradicionalmente, esto se conoce como "asignar un nombre a un conector." Cuando un conector se crea con socket, existe en un espacio de nombres (familia de direcciones) pero carece de nombre. Normalmente, es necesario asignar una dirección local usando bind a un conector SOCK_STREAM antes de que éste pueda recibir conexiones.

VALOR DEVUELTO

Se devuelve 0 en caso de éxito. En caso de error, se devuelve -1 y a errno se le asigna un valor apropiado.

LISTEN

listen - espera conexiones en un conector (socket)

SINOPSIS

```
#include <sys/socket.h>
int listen(int s, int backlog);
```

DESCRIPCIÓN

Para aceptar conexiones, primero se crea un conector con socket, luego se especifica con listen el deseo de aceptar conexiones entrantes y un límite de la cola para dichas conexiones, y por último las conexiones son aceptadas mediante accept. La llamada listen se aplica solamente a conectores de tipo SOCK_STREAM o SOCK_SEQPACKET. El parámetro backlog define la longitud máxima a la que puede llegar la cola de conexiones pendientes. Si una petición de conexión llega estando la cola llena, el cliente puede recibir un error con una indicación de ONNREFUSED o, si el protocolo subyacente acepta retransmisiones, la petición puede no ser tenida en cuenta, de forma que un reintento tenga éxito.

VALOR DEVUELTO

En caso de éxito, se devuelve cero. En caso de error, se devuelve -1 y se pone en errno un valor apropiado.

ACCEPT

accept - acepta una conexión sobre un conector (socket).

SINOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int s, struct sockaddr *addr, socklen_t
          *addrlen);
```

DESCRIPCIÓN

La función accept se usa con conectores orientados a conexión (SOCK_STREAM, SOCK_SEQPACKET y SOCK_RDM). Extrae la primera petición de conexión de la cola de conexiones pendientes, le asocia un nuevo conector con, prácticamente, las misma propiedades que s y reserva un nuevo descriptor de fichero para el conector, el cuál es el valor devuelto por la llamada. El conector original s no se ve afectado por esta llamada. Dese cuenta que cualquier opción por descriptor de fichero (cualquiera que se pueda establecer con F_SETFL de fcntl, como no bloqueante o estado asíncrono) no se hereda en un accept. El argumento s es un conector que ha sido creado con socket, ligado a una dirección local con bind y que se encuentra a la escucha tras un listen. El argumento addr es un puntero a una estructura sockaddr. Esta estructura se rellena con la dirección de la entidad con la que se conecta, tal y como la conoce la capa de comunicaciones. El formato exacto de la dirección pasada en el parámetro addr viene determinado por la familia del conector. El argumento addrlen es un parámetro de entrada/salida: al efectuar la llamada debe contener el tamaño de la estructura apuntada por addr; a la salida, contendrá la longitud real (en bytes) de la dirección devuelta. Cuando addr es NULL nada se rellena. Si no hay conexiones

pendientes en la cola y el conector no está marcado como "no bloqueante", accept bloqueará al invocador hasta que se presente una conexión. Si el conector está marcado como no bloqueante y no hay conexiones pendientes en la cola, accept devolverá EAGAIN.

VALOR DEVUELTO

La llamada devuelve -1 ante un error. Si tiene éxito, devuelve un entero no negativo que es el descriptor del conector aceptado.

CONNECT

connect - inicia una conexión en un conector (socket)

SINOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *serv_addr,
socklen_t addrlen);
```

DESCRIPCIÓN

El descriptor de fichero sockfd debe referenciar a un conector. Si el conector es del tipo SOCK_DGRAM entonces la dirección serv_addr es la dirección a la que por defecto se envían los datagramas y la única dirección de la que se reciben datagramas. Si el conector es del tipo SOCK_STREAM o SOCK_SEQPACKET, esta llamada intenta hacer una conexión a otro conector. El otro conector está especificado por serv_addr, la cual es una dirección (de longitud addrlen) en el espacio de comunicaciones del conector. Cada espacio de comunicaciones interpreta el parámetro

serv_addr a generalmente, los conectores de protocolos orientados a conexión pueden conectarse con éxito mediante connect una vez solamente.

VALOR DEVUELTO

Si la conexión o enlace tiene éxito, se devuelve 0. En caso de error,se devuelve -1, y se asigna a la variable errno un valor apropiado.

SEND

send, sendto, sendmsg - envía un mensaje de un conector (socket)

SINTAXIS

```
#include <sys/types.h>
#include <sys/socket.h>
int send(int s, const void *msg, size_t len, int flags);
int sendto(int s, const void *msg, size_t len, int flags,
const struct sockaddr *to, socklen_t tolen);
int sendmsg(int s, const struct msghdr *msg, int flags);
```

DESCRIPCIÓN

Send, sendto y sendmsg son utilizados para transmitir un mensaje a otro conector. Send solo puede ser usado cuando un conector está en un estado connected mientras sendto y sendmsg pueden ser utilizados en cualquier momento. La dirección de destino viene dada por to con tolen especificando su tamaño. La longitud del mensaje viene dada por len. Si el mensaje es demasiado largo para pasar automáticamente a través del protocolo inferior, se devuelve el error EMSGSIZE y el mensaje no es transmitido.

VALOR DEVUELTO

Las llamadas devuelven el numero de caracteres enviados, o -1 si ha ocurrido un error.

RECV

recv, recvfrom, recvmsg - reciben un mensaje desde un conector

SINOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
int recv(int s, void *buf, size_t lon, int flags);
int recvfrom(int s, void *buf, size_t lon, int flags,
struct sockaddr*desde, socklen_t *londesde);
int recvmsg(int s, struct msghdr *msg, int flags);
```

DESCRIPCIÓN

Las llamadas recvfrom y recvmsg se emplean para recibir mensajes desde un conector (‘‘socket"), y pueden utilizarse para recibir datos de un conector sea orientado a conexión o no. Si desde no es NULL y el conector no es orientado a conexión, la dirección fuente del mensaje se llena. El argumento londesde es un parámetro por referencia, inicializado al tamaño del buffer asociado con desde, y modificado cuando la función regresa para indicar el tamaño real de la dirección guardada ahí. La llamada a recv se utiliza normalmente sólo en un conector conectado (vea connect) y es idéntica a recvfrom con un parámetro desde con valor NULL.

VALOR DEVUELTO

Estas llamadas devuelven el número de bytes recibidos, o bien -1 en caso de que ocurriera un error.

CLOSE

close - cierra un descriptor de fichero

SINOPSIS

```
#include <unistd.h>
int close(int fd);
```

DESCRIPCIÓN

close cierra un descriptor de fichero de forma que ya no se refiera a fichero alguno y pueda ser reutilizado. Cualesquiera bloqueos mantenidos sobre el fichero con el que estaba asociado, y propiedad del proceso, son eliminados (sin importar qué descriptor de fichero fue utilizado para obtener el bloqueo). Si fd es la última copia de cierto descriptor de fichero, los recursos asociados con dicho descriptor son liberados.

VALOR DEVUELTO

close devuelve 0 en caso de éxito y -1 si ocurre algún error. 3.2.2. Estructura sockaddr y sockaddr_in. Cuando se utilizan los sockets como API, las direcciones de red se almacenan en una estructura sockaddr definida en el fichero linux/include/linux/in.h. A continuación se muestra la definición de la estructura directamente copiada del fichero de cabecera /include/linux/in.h.

```
/* Structure describing an Internet (IP) socket address. */ #define
__SOCK_SIZE__ 16 /* sizeof(struct sockaddr) */
struct sockaddr_in {
sa_family_t sin_family; /* Address family */
```

```
unsigned short intsin_port; /* Port number */
struct in_addr sin_addr; /* Internet address */
/* Pad to size of `struct sockaddr'. */
unsigned char __pad[__SOCK_SIZE__ - sizeof(short int) -
sizeof(unsigned short int) - sizeof(struct in_addr)];
};
#define sin_zero __pad /* for BSD UNIX comp. -FvK */
```

Como se puede observar dicha estructura encapsula una dirección de transporte definida por la pareja dirección IP y número de puerto. Sin embargo, la estructura anterior solo define la interfaz empleada para el protocolo de Internet (IP), mientras que es posible trabajar con otras familias de protocolos. Por ello, la API de sockets de C trabaja realmente con una estructura genérica llamada sockaddr. Esta estructura emplea abstracción de datos, de forma que mientras que los tipos de datos (en este caso familias de protocolos) pueden cambiar, la forma de trabajar de las funciones permanece igual. A continuación se muestra la definición de la estructura sockaddr, directamente extraída del fichero de cabecera /include/sys/socket.h.

```
/* This is the type we use for generic socket address arguments.
With GCC 2.7 and later, the funky union causes redeclarations or
uses with any of the listed types to be allowed without complaint.
G++ 2.7 does not support transparent unions so there we want the
old-style declaration, too. */
#if defined __cplusplus || !__GNUC_PREREQ (2, 7) || !defined __USE_GNU
# define __SOCKADDR_ARG struct sockaddr *__restrict
# define __CONST_SOCKADDR_ARG __const struct sockaddr *
#else
/* Add more `struct sockaddr_AF' types here as necessary.
These are all the ones I found on NetBSD and Linux. */
# define __SOCKADDR_ALLTYPES \
__SOCKADDR_ONETYPE (sockaddr) \
__SOCKADDR_ONETYPE (sockaddr_at) \
__SOCKADDR_ONETYPE (sockaddr_ax25) \
__SOCKADDR_ONETYPE (sockaddr_dl) \
__SOCKADDR_ONETYPE (sockaddr_eon) \
__SOCKADDR_ONETYPE (sockaddr_in) \
__SOCKADDR_ONETYPE (sockaddr_in6) \
__SOCKADDR_ONETYPE (sockaddr_inarp) \
__SOCKADDR_ONETYPE (sockaddr_ipx) \
```

```
__SOCKADDR_ONETYPE (sockaddr_iso) \
__SOCKADDR_ONETYPE (sockaddr_ns) \
__SOCKADDR_ONETYPE (sockaddr_un) \
__SOCKADDR_ONETYPE (sockaddr_x25)
# define __SOCKADDR_ONETYPE(type) struct type *__restrict __##type##__;
typedef union { __SOCKADDR_ALLTYPES
} __SOCKADDR_ARG __attribute__ ((__transparent_union__));
# undef __SOCKADDR_ONETYPE
# define __SOCKADDR_ONETYPE(type) __const struct type *__restrict
__##type##__;
typedef union { __SOCKADDR_ALLTYPES
} __CONST_SOCKADDR_ARG __attribute__ ((__transparent_union__));
# undef __SOCKADDR_ONETYPE
#endif
```

2. Ejemplo: Cliente y Servidor de Eco.

El código fuente necesario para compilar el cliente y el servidor de eco se distribuye en varios ficheros:

1) misfunciones.h: incluye los prototipos de las funciones y constantes definidas por el programador.
2) misfunciones.cpp: contiene el código fuente de las funciones diseñadas por el programador.
3) ecoclient.cpp: código fuente del cliente de eco. El formato es: ecoclient.exe dirIP_servidor Port_servicio.
4) ecoserver.cpp: código fuente del servidor de eco. El formato es: ecoserver.exe Port_escucha.
5) makefile: contiene la secuencia necesaria de ordenes para la compilación tanto del cliente como del servidor de eco. Para compilar el cliente: make ecoclient.exe. Para compilar el servidor: make ecoserver.exe.

```
// MISFUNCIONES.H
// Constantes de error definidas
#define ErrorPort "No se ha asignado numero de puerto"
#define ErrorServ "No se ha indicado el nombre del servicio"
#define ErrorHost "Nombre de Host no valido"
```

```
#define ErrorDirIp "Direccion IP no valida"
#define ErrorSocket "No se ha asignado un descriptor de socket valido"
#define ErrorBind "Fallo al llamar a Bind"
#define ErrorSend "Fallo al llamar a Send"
#define ErrorSendto "Fallo al llamar a Sendto"
#define ErrorRecv "Fallo al llamar a Recv"
#define ErrorRecvfrom "Fallo al llamar a Recvfrom"
#define ErrorConnect "Fallo al llamar a Connect"
#define ErrorAccept "Fallo al llamar a Accept"
#define ErrorListen "Fallo al llamar a Listen"
#define ErrorEcoClient "Comando incorrecto.\nEl formato es: ecoclient.exe
dirIP_servidor Port_servicio mensaje"
#define ErrorEcoServer "Comando incorrecto.\nEl formato es: ecoserver.exe
Port_escucha"
#define ErrorHostAddress "Puntero sockaddr NULL"
#define ErrorMemDinamica "Insuficiente espacio de memoria\n"
// Tiempo de espera de respuesta en segundos
#define Plazo 30
int recibido_en_plazo(int sd, int segundos);
void error(char *mensaje);
void error_cierre(char *mensaje, int sd);
int accion_cliente_eco(int cliente, char *mensaje);
int accion_servidor_eco(int servidor);
struct sockaddr* hostaddress_tcp_ip(char* dir_IP, int puerto);
int crear_conector_servidor_tcp_ip(int cola, struct sockaddr* addr);
int conexion(int sd, struct sockaddr* addr);
int crear_conector_cliente_tcp_ip(struct sockaddr* addr);

// MISFUNCIONES.C
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/time.h>
#include "misfunciones.h"
// Permite esperar a que llegue algo por el descriptor del socket con un
plazo
determinado en segundos
int recibido_en_plazo(int sd, int segundos)
```

```
{
struct timeval plazo;
fd_set readfds;
plazo.tv_sec = segundos;
plazo.tv_usec = 0;
FD_ZERO(&readfds);
FD_SET(sd, &readfds);
return
select(sd+1,
&readfds,
(fd_set *)NULL,
(fd_set *)NULL,
&plazo) !=0;
}
// Imprime un mensaje de error
void error(char *mensaje)
{
fprintf(stderr,"Error: %s\n",mensaje);
_exit(1);
}
// Imprime el error y cierra la conexion
void error_cierre(char *mensaje, int sd)
{
close(sd);
error(mensaje);
}
// Accion que lleva a cabo el cliente de eco y devuelve 0 si no hay error
int accion_cliente_eco(int cliente, char *mensaje)
{
int flagerr = 0;
char buffer[1024];
int bytesrx;
if ( (send(cliente, mensaje, sizeof(mensaje), 0)) < 0 )
{
error_cierre(ErrorSend, cliente);
flagerr = 1;
}
if( recibido_en_plazo(cliente,Plazo) )
{
bytesrx = recv(cliente, buffer, sizeof(buffer), 0);
if( bytesrx > 0 )
{
printf("El servidor responde:\n%s\n",buffer);
close(cliente);
}
```

```
else
{
flagerr = 1;
error_cierre(ErrorRecv, cliente);
}
} else
{
printf("El servidor no responde\n");
}
return flagerr;
}
// Accion que lleva a cabo el servidor de eco
int accion_servidor_eco(int cliente)
{
int flagerr = 0;
char buffer[1024];
int bytesrx;
int bytestx;
bytesrx = recv(cliente, buffer, sizeof(buffer), 0);
if( bytesrx > 0 )
{
bytestx = send(cliente, buffer, sizeof(buffer), 0);
if( bytestx == -1 )
{
flagerr = 1;
error_cierre(ErrorSend, cliente);
}
} else
{
flagerr = 1;
error_cierre(ErrorRecv, cliente);
}
return flagerr;
}
// Inicializa la estructura sockaddr y devuelve un puntero a dicha
estructura
para TCP/IP
struct sockaddr* hostaddress_tcp_ip(char* dir_ip, int puerto)
{
struct sockaddr *punt = NULL;
if ( dir_ip != NULL )
{
struct sockaddr_in *addr = new struct sockaddr_in();
addr->sin_family = PF_INET; // red IP
addr->sin_port = htons( puerto ); // puerto especificado
```

```
inet_aton( dir_ip, &(addr->sin_addr) );
punt = reinterpret_cast<struct sockaddr*>(addr);
} else
{
error(ErrorDirIp);
}
return punt;
}
// Crea el conector del servidor, le da nombre(bind) y lo pone a la
escucha
para TCP/IP
// devuelve el descriptor del socket
int crear_conector_servidor_tcp_ip(int cola, struct sockaddr* addr)
{
int s;
if ( ( s = socket(PF_INET, SOCK_STREAM, 0) ) < 0 )
{
error(ErrorSocket);
}
if ( bind(s, addr,sizeof(struct sockaddr_in)) < 0 )
{
error_cierre(ErrorBind, s);
}
if ( listen(s, cola) != 0 )
{
error_cierre(ErrorListen, s);
}
return s;
}
int conexion(int sd, struct sockaddr* addr)
{
int s_client;
socklen_t len = sizeof(struct sockaddr_in);
if ( ( s_client = accept(sd, addr, &len) ) < 0 )
{
error(ErrorAccept);
}
printf("\nCONECTADO\n");
return s_client;
}
// Crea el conector del cliente y llama a connect para TCP/IP
// devuelve el descriptor del socket
int crear_conector_cliente_tcp_ip(struct sockaddr* addr)
{
int s;
```

```
if ( ( s = socket(PF_INET, SOCK_STREAM, 0) ) < 0 )
{
error(ErrorSocket);
}
if ( connect(s, addr,sizeof(struct sockaddr_in)) < 0 )
{
error_cierre(ErrorConnect, s);
}
return s;
}

// CLIENTE.CPP
// Cliente que envía un mensaje y recibe e imprime una copia del mismo
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include "misfunciones.h"
int main (int argc, char *argv[])
{
int cliente;
struct sockaddr *serv;
if( argc != 4 )
{
error(ErrorEcoClient);
} else
{
if ( (serv = hostaddress_tcp_ip(argv[1], atoi(argv[2]))) != NULL)
{
cliente = crear_conector_cliente_tcp_ip(serv);
accion_cliente_eco(cliente, argv[3]);
} else
{
error(ErrorHostAddress);
}
}
}

//ECOSERVER.C
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include "misfunciones.h"
int main (int argc, char *argv[])
{
int cola = 20;
```

```
int servidor;
int cliente;
struct sockaddr *serv;
if( argc != 2 )
{
error(ErrorEcoServer);
} else
{
if ( (serv = hostaddress_tcp_ip("0.0.0.0", atoi(argv[1]))) != NULL)
{
servidor = crear_conector_servidor_tcp_ip(cola, serv);
// Bucle infinito del servidor
for(;;)
{
if ( recibido_en_plazo(servidor, Plazo) )
{
cliente = conexion(servidor, serv);
} else
{
close(servidor);
printf("****NO OIGO NADA*****\n");
_exit(0);
}
if ( cliente > 0 ) // El cliente respondio en plazo
{
accion_servidor_eco(cliente);
close(cliente);
}
}
} else
{
error(ErrorHostAddress);
}
}
}
```

MAKEFILE.

```
CC= g++
FLAGG= -g
FLAGO= -o
FLAGC= -c
FLAGW= -Wall
NAMES= ecoserver
```

```
NAMEC= ecoclient
NAMEF= misfunciones
OBJS= $(NAMES).o
OBJC= $(NAMEC).o
OBJF= $(NAMEF).o
$(NAMEC).exe: $(OBJC) $(OBJF)
$(CC) $(FLAGG) $(FLAGW) $(FLAGO) $(NAMEC).exe $(OBJC) $(OBJF) >&
errecoclient
$(OBJC): $(NAMEC).cpp $(OBJC)
$(CC) $(FLAGG) $(FLAGW) $(FLAGC) $(NAMEC).cpp >& errecoclientobj
$(NAMES).exe: $(OBJS) $(OBJF)
$(CC) $(FLAGG) $(FLAGW) $(FLAGO) $(NAMES).exe $(OBJS) $(OBJF) >&
errecoserver
$(OBJS): $(NAMES).cpp $(OBJF)
$(CC) $(FLAGG) $(FLAGW) $(FLAGC) $(NAMES).cpp >& errecoserverobj
$(OBJF): $(NAMEF).cpp
$(CC)    $(FLAGG)    $(FLAGW)    $(FLAGC)    $(NAMEF).cpp    >&
errmisfunciones
```