CODINOA Consejo de Decanos de Ingeniería del NOA
XII JORNADAS DE CIENCIA Y TECNOLOGÍA DE FACULTADES DE INGENIERÍA DEL NOA
San Fernando del Valle de Catamarca, 10 y 11 de agosto del 2017.

Programación distribuida con interfaz de socket en IPv6

Albaca Paraván, Carlos[1], Saade, Sergio D. [2] y Lutz, Federico H.[3]

(1) Facultad de Ciencias Exactas y Tecnología, Universidad Nacional de Tucumán. calbaca@herrera.unt.edu.ar

(2) Facultad de Ciencias Exactas y Tecnología, Universidad Nacional de Tucumán. ssaade@herrera.unt.edu.ar

(3) Facultad de Ciencias Exactas y Tecnología, Universidad Nacional de Tucumán. fhlutz@gmail.com

RESUMEN: El presente trabajo aborda el desarrollo de aplicaciones distribuidas utilizando la interfaz de sockets con el protocolo IPv6 en lugar del tradicional protocolo IPv4. El trabajo comienza haciendo un breve análisis de las rutinas utilizadas para el protocolo IPv4. Luego las compara con las modificaciones introducidas para adaptar al protocolo IPv6. En base a la nueva API definida se construyó una aplicación simple que funcione totalmente sobre IPv6. Se sacaron importantes conclusiones como por ejemplo la forma de codificar en el caso común de ambientes con dual stack IPv4/IPv6, codificación independiente de la familia de direcciones y otras.

ABSTRACT: This paper addresses the development of distributed applications using the sockets interface with the IPv6 protocol instead of the traditional IPv4 protocol. The work begins by making a brief analysis of the routines used for the IPv4 protocol. Then compare them with the modifications introduced to adapt to the IPv6 protocol. Based on the new API defined, it was built a simple application that works totally on IPv6. Important conclusions were drawn such as how to codify in the common case of environments with dual stack IPv4/IPv6, coding independent of the address family and others.

Palabras claves: Interfaz de Socket, IPv6, Programación Distribuida
Keywords: Socket Interface, IPv6, Distributed programming

1 INTRODUCCIÓN

Hay dos arquitecturas que han sido determinantes y básicas en el desarrollo de los estándares de comunicación: el conjunto de protocolos TCP/IP y el modelo de referencia OSI. TCP/IP es la arquitectura más adoptada para la interconexión de sistemas, mientras que OSI se ha convertido en el modelo estándar para clasificar las funciones de comunicación.

TCP/IP es el resultado de la investigación y desarrollo llevados a cabo en la red experimental de conmutación de paquetes ARPANET y se denomina globalmente como la familia de protocolos TCP/IP. Consiste en una extensa colección de protocolos que se han adoptado como estándares en Internet.

Al contrario que en OSI, no hay un modelo oficial de referencia TCP/IP. No obstante, como menciona Stallings (2000), basándose en los protocolos estándares que se han desarrollado a través de los años, todas las tareas involucradas en la comunicación se pueden organizar en cinco capas relativamente independientes:

- Capa de aplicación: contiene la lógica necesaria para posibilitar las distintas aplicaciones de usuario.
- Capa de transporte: contiene los procedimientos que garantizan una transmisión segura. Siendo el protocolo TCP el más usado para proporcionar esta funcionalidad.
- Capa de internet: contiene los procedimientos para que los datos se encaminen a través de diferentes redes interconectadas. El protocolo IP se utiliza en esta capa para ofrecer este servicio.

- Capa de acceso a la red: es la responsable del intercambio de datos entre el sistema final y la red a la cual está conectado.
- Capa física: define la interfaz física entre el dispositivo de transmisión de datos y el medio de transmisión o red.

La Fig.1 muestra la arquitectura de capas de los protocolos TCP/IP entre dos sistemas finales y una red que los conecta. Nótese que la capa física y la de acceso a la red proporcionan interacción entre el sistema final y la red, mientras que la capa de aplicación y la de transporte albergan los llamados protocolos extremo a extremo ya que facilitan la interacción entre los sistemas finales. La capa de Internet tiene algo de las dos aproximaciones anteriores. En esta capa, los sistemas origen y destino proporcionan a la red la información necesaria para realizar el encaminamiento, pero a la vez, deben proporcionar algunas funciones adicionales de intercambio entre los sistemas finales.

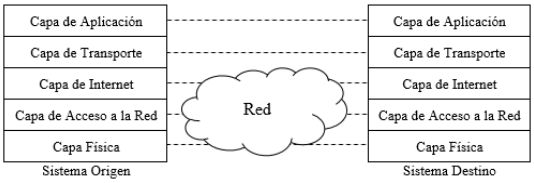

Figura 1. Implementación de los protocolos TCP/IP en sistemas finales.

La versión del protocolo IP utilizada desde la implementación de la suite de protocolos TCP/IP hasta la actualidad es la versión 4, pero a la fecha, las direcciones IPv4 utilizadas originalmente en Internet se encuentran en una franca etapa de agotamiento a nivel mundial. Por ejemplo, LACNIC: Centro regional de otorgamiento de direcciones IP para Latinoamérica y el Caribe- (2017), ya se encuentra en la cuarta y última fase de entrega de direcciones. Finalizada esta última fase se dejarán de brindar direcciones IPv4 a los diferentes países de la región.

Durante el transcurso de los años, se introdujeron modificaciones a IPv4 (NAT, CIDR, etc.) para tratar de subsanar este inconveniente, pero la solución definitiva a este problema es la implementación de IPv6, sin embargo en nuestro país el grado de implementación es prácticamente nulo, tal como lo menciona Nader (2015).

Saade et all (2014, 2015, 2016), miembros del Laboratorio de Redes de Computadoras de la Facultad de Ciencias Exactas y Tecnología de la Universidad Nacional de Tucumán, vienen trabajando desde hace varios años en la investigación y desarrollo de IPv6. El presente trabajo en particular aborda la temática del uso de la interfaz de socket en la programación de aplicaciones distribuidas utilizando el protocolo IPv6 como protocolo de Internet.

2 PUERTOS Y SOCKETS

La forma en que una aplicación se identifica con el protocolo de transporte (y viceversa) es a través de un número denominado puerto (rango de 0 a 65535). Un puerto identifica en forma unívoca a la aplicación, es decir que dentro de un computador, no pueden existir más de una aplicación con el mismo número de puerto. La Fig.2 muestra algunas aplicaciones típicas (lado servidor), con los números de puertos que utiliza y el correspondiente protocolo de transporte.

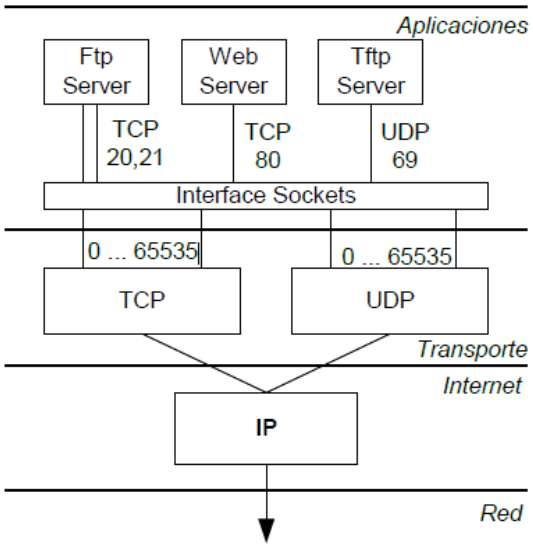

Figura 2. Número de puertos en aplicaciones (Servidoras).

Las aplicaciones de red bajo TCP/IP, siguen normalmente el modelo Cliente/Servidor, en donde se tiene por una parte una aplicación servidora que atiende y resuelve las solicitudes de (hipotéticamente) diversas aplicaciones clientes. Por ejemplo, se puede tener una aplicación para la

transferencia de archivos, como FTP. Esta aplicación consta de una parte servidora – el servidor FTP – y una parte cliente – el cliente FTP. Este último solicita al primero el servicio de transferencia de un cierto archivo, para lo cual el servidor contesta con el envío del mismo.
Las aplicaciones servidoras, siempre atienden o escuchan en puertos que son universalmente conocidos (*well known ports*), mientras que las aplicaciones clientes, lo hacen sobre puertos que son creados dinámicamente (*dinamically allocated ports*).
En la Fig.2, las aplicaciones que se muestran son servidoras, por lo tanto, los puertos correspondientes son puertos universalmente conocidos.
Con un puerto se identifica una aplicación dentro de un computador en forma precisa. Con un *socket*, es posible identificar a una aplicación dentro de toda la red. Eso se consigue especificando además de la aplicación, la dirección IP del computador donde reside (que es única). Se tiene así una terna: (Dirección IP, Protocolo de Transporte, Puerto) que define en forma unívoca a una aplicación dentro de toda la Internet. Esta terna se denomina un *socket*.
En particular, como enuncia Saade (2004), si tenemos una comunicación entre dos aplicaciones cooperativas, se puede representar la asociación entre ambas aplicaciones a través de una quíntupla: (Protocolo, Dirección IP Local, Puerto Local, Dirección IP Remota, Puerto Remoto).

3 INTERFAZ DE SOCKET

La interfaz de *socket* es una capa ubicada entre la de aplicación y la de transporte dentro de un *host*, tal como puede apreciarse en la Fig.3. Esta interfaz también es conocida como API (*Application Programmers Interface*) entre la aplicación y la red, ya que justamente la interfaz de *socket* es la interfaz de programación con las que las aplicaciones distribuidas son desarrolladas en Internet. El desarrollador de aplicaciones tiene el control total de lo correspondiente al lado de la capa de aplicación del *socket*, pero muy poco control del lado de la capa de transporte del mismo. Es más, como lo indica Kurosa (2002), el único control que tiene el desarrollador sobre la capa de transporte es la selección del protocolo de transporte y quizás la capacidad de fijar algunos parámetros como el tamaño máximo de los *buffers* o el *maximum segment size*.

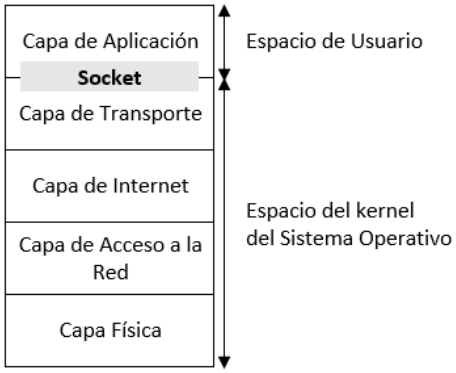

Figura 3. Interfaz de Socket en el modelo TCP/IP.

La interfaz de *sockets* tiene como atractivo particular, el hecho de que realiza una analogía entre la interfaz de acceso al sistema de archivos de Unix y la comunicación entre aplicaciones distribuidas en red.
En efecto, en Unix, un archivo está identificado en cada proceso a través de un número que se denomina descriptor de archivo. La API del sistema de archivo, define funciones como open(), close(), read(), write(), etc. que utilizan a tal descriptor para acceder al archivo y realizar las operaciones correspondientes sobre él.
Es posible extrapolar este concepto a un *socket*. De esta forma se asociaría un descriptor de *socket* con una aplicación específica en la red. Las funciones serían básicamente las mismas: open(): para abrir un *socket* o asociar una aplicación específica a un número (descriptor de *socket*), read(): para recibir o leer un mensaje desde la aplicación (remota), write(): para escribir un mensaje en la aplicación (remota). En las implementaciones actuales de *sockets*, si bien el concepto del manejo de aplicaciones distribuidas con una interfaz análoga a la del sistema de archivo sigue vigente, Stevens (1990) y Gray (1998) remarcan que las funciones que se utilizan son diferentes. Eso se debe a que en realidad los parámetros que se deben especificar para una comunicación entre aplicaciones de red son mayores, como por ejemplo, si la aplicación es cliente o servidor, si el protocolo de transporte que se usará es orientado a conexión (TCP) o no

orientado a conexión (UDP), direcciones de red (IP), número de puertos, etc.

4 TIPOS DE SOCKETS

Al especificar el protocolo de transporte en el momento de la creación de un *socket*, se está determinando uno de los dos tipos básicos de sockets que existen. A saber, según Saade (2016):

4.1 *Sockets de flujo*

Cuando se utilizan estos *sockets* se tiene una comunicación confiable y libre de errores, en donde los datos se transfieren de un extremo al otro conformando un flujo continuo de bytes (*stream*). Los *sockets* de este tipo son orientados a conexión, es decir, existe una conexión lógica entre los procesos o aplicaciones. Debido a estas características, estos tipos de *sockets* utilizan como protocolo de transporte a TCP.

4.2 *Sockets de datagrama*

En este caso, la comunicación es potencialmente no-confiable y no es orientada a conexión. Es decir, cada datagrama es independiente del previo y del posterior, por lo que hipotéticamente los datos extremo a extremo, pueden ser recibidos fuera del orden en que fueron enviados (o incluso no recibidos). El protocolo de transporte asociado a estos tipos de *sockets* es UDP.

5 TIPOS DE APLICACIONES

Como lo remarca Saade (2016), de acuerdo al tipo de socket que se seleccione para la comunicación, se tendrá entonces aplicaciones cliente/servidor orientadas a conexión o no-orientadas a conexión, cuya naturaleza es radicalmente diferente.

5.1 *Aplicaciones orientadas a conexión*

La secuencia que siguen un par de aplicaciones cliente/servidor, mostrada en la Fig.4, es básicamente la siguiente:

- Se crea un *socket* (socket()). Este *socket* no es más que una asociación entre la aplicación y el protocolo de transporte, ya que en esta instancia, no se asocia a dicho *socket* con una dirección IP, ni con un número de puerto.
- El *socket* es asociado a una dirección de red y a un número de puerto, que en este caso será un puerto bien conocido, puesto que es una aplicación servidora (bind()).
- Se crea una cola para las futuras solicitudes de conexiones remotas (listen()).
- A partir de ese momento, la aplicación servidora está preparada para aceptar conexiones de aplicaciones clientes (accept()). Por lo tanto se auto bloquea esperando tales conexiones remotas.
- El cliente por su parte, crea su *socket* y se conecta al servidor (connect()).
- A partir de ese momento comienza el intercambio de datos, con sucesivas lecturas (read()) y escrituras (write()).

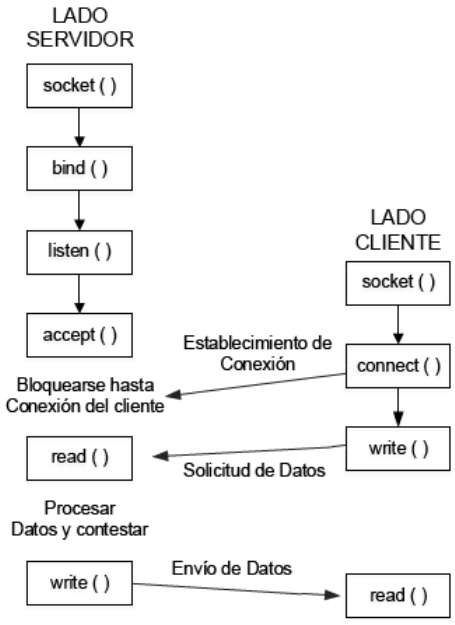

Figura 4. Diagrama de flujo de interacción cliente/servidor con sockets de flujo (Aplicaciones orientadas a conexión).

5.2 *Aplicaciones no orientadas a conexión*

La secuencia de eventos entre una aplicación cliente/servidor no-orientada a conexión, como muestra la Fig.5, tiene algunos elementos en común, como por ejemplo, la creación de *sockets*.

Sin embargo, tanto el servidor como el cliente, asocian a dicho *socket* con una dirección (a través de bind()). Se observa que el cliente no establece una conexión con el servidor. En su lugar, tanto el servidor como el cliente reciben y envían datagramas desde y hacia direcciones específicas (recvfrom() y sendto()).

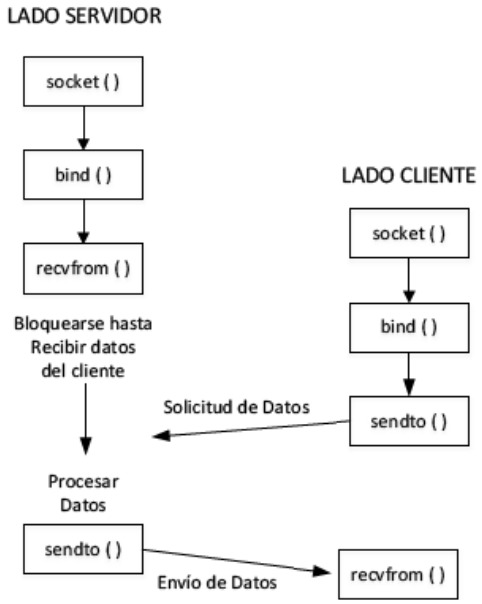

Figura 5. Diagrama de flujo de interacción cliente/servidor con sockets de datagrama (Aplicaciones no-orientadas a conexión).

6 CONSIDERACIONES

Hagino (2004) sugiere que hay que tener en cuenta las siguientes cuestiones de acuerdo a si se programará con IPv4 o IPv6 como protocolo de Internet al momento de programar usando la interfaz de *socket*:

6.1 *Familia de direcciones*

Al momento de crear un *socket* se debe especificar el tipo de direcciones con el que se realizará la comunicación. A los fines de este trabajo cabe mencionar las constantes AF_INET que identifica un *socket* IPv4 y AF_INET6 que identifica uno IPv6.

6.2 *Estructuras de datos*

Para identificar IPv4 en la API de *socket*, se utiliza (en C) una estructura llamada sockaddr_in, mostrada a continuación:

```
struct sockaddr_in {
u_int8_t        sin_len; /* Longitud de sockaddr */
u_int8_t        sin_family; /* Familia de direcciones */
u_int16_t       sin_port; /* Nº de puerto TCP/UDP */
struct in_addr  sin_addr; /* Dirección IPv4 */
int8_t          sin_zero[8]; /* Relleno */
};
```

Mientras que para identificar IPv6 se usa una llamada sockaddr_in6:

```
struct sockaddr_in6 {
u_int8_t        sin6_len; /* Tamaño de esta estructura */
u_int8_t        sin6_family; /* AF_INET6 */
u_int16_t       sin6_port; /* Nº de puerto TCP/UDP */
u_int32_t       sin6_flowinfo; /* Información de flujo */
struct in6_addr sin6_addr; /* Dirección IPv6 */
u_int32_t       sin6_scope_id; /* Id de zona de alcance*/
};
```

6.3 *Independencia de la familia de direcciones*

Hay numerosas razones para tomar esta decisión al momento de programar las aplicaciones, a saber:

- Soportar ambientes *dual stack* IPv4/v6, ya que los programas deben manejar ambos protocolos apropiadamente.
- Evitar la reescritura de código en caso de la aparición de un nuevo protocolo de capa de Transporte o de Internet, o en caso de querer soportar otra familia de direcciones como por ejemplo AppleTalk.
- Si se fija la familia de direcciones en el programa, éste puede no llegar a funcionar si el *kernel* del sistema operativo no la soporta.

6.4 *APIs que no deberían usarse*

Como se recomienda programar en forma independiente de la familia de direcciones, no es recomendable usar APIs que utilicen las estructuras in_addr o in6_addr, por ejemplo: getservbyname, gethostbyname, inet_lnaof, getservbyport, inet_makeaddr, gethostbyname2, entre otras.

7 EJEMPLO

A continuación se brinda un ejemplo simple de una aplicación cliente/servidor. Para el mismo se utilizó TCP como protocolo de capa de transporte con el uso de sockets de flujo e IPv6 como protocolo de Internet. Por cuestiones de espacio, se realizó una aplicación que corre exclusivamente sobre IPv6 (no soportando *dual stack*), se omitieron mensajes de bienvenida e informativos, y control de errores de las llamadas a sistemas y funciones utilizadas. El programa se realizó utilizando lenguaje C sobre sistema operativo Linux e implementa el diagrama mostrado en la Fig.4.

La aplicación servidora, mostrada en la Fig.6, se comporta de la siguiente manera:

- Se declaran e inicializan las estructuras de datos requeridas para realizar la comunicación con las aplicaciones cliente.
- Se invocan las llamadas a sistemas necesarias para crear un *socket*, asociar la dirección IP y número de puerto al mismo, y especificar el número de conexiones que el servidor podrá mantener en espera.
- La aplicación entra en un bucle infinito que la mantendrá en ejecución hasta que el usuario decida finalizarla de forma manual.
- La aplicación se bloquea esperando una conexión cliente, y al momento que se realiza la misma, el servidor envía un mensaje al cliente solicitando el ingreso de una cadena de caracteres.
- La cadena enviada por el cliente es recibida por el servidor, transformada a mayúscula y devuelta al cliente.
- La aplicación cierra el *socket* y se bloquea esperando nuevamente otra conexión.

La aplicación cliente, mostrada en la Fig.7, se comporta de la siguiente manera:

- Se declaran e inicializan las estructuras de datos requeridas para realizar la comunicación con la aplicación servidora.
- Se invocan las llamadas a sistemas necesarias para crear un *socket* y realizar la conexión con el servidor.
- Se recibe un mensaje indicando que se debe ingresar una cadena de caracteres, y luego de haberla ingresado, la misma es enviada al servidor.
- La cadena convertida a mayúscula es recibida y mostrada por pantalla, finalizando la aplicación cliente.

8 CONCLUSIONES

La programación distribuida utilizando la interfaz de *socket* con IPv6 constituye una herramienta simple de usar y muy poderosa si, por un lado, se poseen conocimientos básicos de programación, y por el otro, se tienen en cuenta las consideraciones presentadas en este trabajo.

La complejidad de la utilización de un protocolo de Internet u otro no varía de forma considerable, la principal dificultad radica específicamente en la posibilidad de programar para aplicaciones que puedan utilizar ambas versiones del protocolo IP y que además puedan ser escalables o adaptables con poca necesidad de reprogramación.

Por último, cabe destacar que este trabajo surgió como resultado de la investigación llevada a cabo para ver la posibilidad de actualizar un trabajo práctico de laboratorio de programación distribuida sobre IPv4 que se dicta en la asignatura Protocolos de Comunicación TCP/IP (Ingeniería en Computación – FACET – UNT).

9 AGRADECIMIENTOS

El presente trabajo fue realizado en el marco del Programa de Investigación del Consejo de Investigaciones de la UNT (CIUNT) 26/E543 “Métodos de Diseño en Tecnología de la Información”, específicamente en el proyecto 26/E575 “Protocolos de Comunicación: de los sistemas embebidos a Internet – Segunda Parte”.

```
#include <stdio.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <string.h>
int main(){
    int socket_original, socket_nuevo, socket_size, i, tamanio=101;
    char ip_string[INET6_ADDRSTRLEN], mensaje[tamanio], opcion[1];
    char bienvenida[]= “Ingrese una cadena de hasta 100 caracteres:”;
    struct sockaddr_in6 servidor_addr, cliente_addr;
    socket_original = socket(AF_INET6, SOCK_STREAM, 0);
    servidor_addr.sin6_family = AF_INET6;
    servidor_addr.sin6_port = htons(5000);
    servidor_addr.sin6_addr = in6addr_any;
    bind(socket_original, (const void*)&servidor_addr, sizeof(servidor_addr));
    listen(socket_original,5);
    while(1){
        socket_size = sizeof(cliente_addr);
        socket_nuevo = accept(socket_original, (struct sockaddr*) &cliente_addr, &socket_size);
        inet_ntop(AF_INET6, &(cliente_addr.sin6_addr), ip_string, INET6_ADDRSTRLEN);
        printf("\nConexión desde IP: [%s] y Puerto: %i\n", ip_string, cliente_addr.sin6_port);
        write(socket_nuevo, bienvenida, strlen(bienvenida));
        read(socket_nuevo, mensaje, tamanio);
        printf("\nLa cadena recibida es: %s\n", mensaje);
        for(i = 0; i<tamanio; i++)
            mensaje[i] = toupper(mensaje[i]);
        write(socket_nuevo, mensaje, tamanio);
        close(socket_nuevo);
    }
    close(socket_original);
    return 0;
}
```

Figura 6. Código en C de la Aplicación Servidora.

```
#include <stdio.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <string.h>
int main(int argc, char *argv[]){
    int socket_original, tamanio=101;
    char mensaje[tamanio];
    struct sockaddr_in6 servidor_addr;
    socket_original = socket(AF_INET6, SOCK_STREAM, 0);
    servidor_addr.sin6_family = AF_INET6;
    inet_pton(AF_INET6, argv[1], &(servidor_addr.sin6_addr));
    servidor_addr.sin6_port = htons(5000);
    connect(socket_original, (struct sockaddr*) &servidor_addr, sizeof(struct sockaddr_in6));
    read(socket_original, mensaje, tamanio);
```

```
    printf("\n%s", mensaje);
    fflush(stdin);
    fgets(mensaje, tamanio, stdin);
    write(socket_original, mensaje, tamanio);
    read(socket_original, mensaje, tamanio);
    printf("\nRespuesta del servidor: %s\n", mensaje);
    close(socket_original);
    return 0;
}
```

Figura 7. Código en C de la Aplicación Cliente.

10 REFERENCIAS

Gray, J. S, *Interprocess Comunications in UNIX*, Prentice-Hall, Second Edition, 1998.

Hagino, J., *IPv6 Network Programming*, Elsevier Digital Press, 2004.

Kurosa, J. and K. Ross, *Computer Networking: A Top-Down Approach Featuring the Internet*, Addison-Wasley, Second Edition, 2002.

Lacnic. Fases de Agotamiento de IPv4. [Online]. <http://www.lacnic.net/web/lacnic/agotamiento-ipv4> [Accedido 23/02/2017].

Nader, F., S. Saade and J. Bilbao, Estado de Implementación de IPv6 en Argentina, *Anales de X Jornadas de Ciencia y Tecnología de las Facultades de Ingeniería del NOA*, 2015.

Saade, S., C. Albaca Paraván, F. Lutz and A. Gallardo, Protocolo IPv6: Estudio e Implementación en Laboratorio, *Anales de Jornadas 2014 en Ingeniería Eléctrica, Electrónica y Computación*, 2014.

Saade, S., C. Albaca Paraván, F. Lutz and A. Gallardo, DHCPv6: Una comparativa con DHCPv4, *Anales de X Jornadas de Ciencia y Tecnología de las Facultades de Ingeniería del NOA*, 2015.

Saade, S., C. Albaca Paraván, F. Lutz and A. Wolfenson, Análisis Comparativo de Direcciones IPv6 Autoconfiguradas en Plataformas MS Windows y Linux, *Anales de Jornadas 2014 en Ingeniería Eléctrica, Electrónica y Computación*, 2015.

Saade, S., C. Albaca Paraván and F. Lutz, Incorporación de IPv6 en la Currícula de Ingeniería en Computación, *Anales de IEEE Argencon 2016 – El Congreso Bienal de IEEE Argentina*, 2016.

Saade, S. and J. Borelli, Programación en Redes TCP/IP Utilizando Interfase de Sockets, *Cet: Revista de Ciencias Exactas e Ingeniería*, 25, pp. 20-28, 2004.

Saade, S. *Protocolos de Comunicación en Internet*, Edunt, 2016.

Stallings, W., *Comunicaciones y Redes de Computadoras*, 6ta edición. Pearson Education, 2000.

Stevens, W. R., *Unix Network Programming*, Prentice-Hall Software Series, 1990.