Practical Exercises in Computer Networks

Echo client/server written in the C language (WIP)

This practical aims to present the Datagram Sockets API by creating a Client/Server program pair written in C

Datagram Sockets

The Internet as a whole can be considered a packet-switching network where any information we want to send must be broken down into a series of small chunks before transmission. The resulting pieces are conventionally known as packets and each is transmitted and transported independently of the others. Packets are similar to letters in many respects, for example, suppose you need to send fifty A4-sized written sheets to a friend in New York City, you will have to break down the **50-page block** into smaller blocks such that each fits within the envelope size you own at the moment. Each of the 5 10-page blocks will have to be addressed to your friend. Potentially, some of them might be transported over different routes than the others and they might arrive in an order other than the original order. We must not forget an important fact now: any of the outgoing letters could be lost by the mail system or get damaged. The example just developed briefly represents the nature of Internet which is technically known as the **Internet Model of Service**.

The terms used in Internet parlance are not the same used to describe the physical world: the 50-page block would be named the message to send, each of the 5 10-page blocks would be called a packet and the network (The Internet) could deliver those packets out-of-order, duplicated, with errors or, even not deliver some of them at all. The Internet Model of Service is a ***best effort model,*** one which guarantees are very loose. There is an Internet model of service property that has no counterpart in the real world example: the network can, unexpectedly duplicate packets erroneously; this does not happen in the mail example - that figures. The key word here is *packet*, a sort of envelope that contains the Internet address of the host to which it is meant to be delivered. This is the basic nature of Internet: lack of deterministic guarantees, but this seems to leave us in a disadvantageous position as beginners, for, what is the rationale of a network that offers us no delivery guarantee? We will delve into the Internet Service Model in upcoming lectures, therefore, for the time being we will want to recover the lost trust, somehow: what the network can not guarantee us, we will compensate for those guarantees by increasing the responsibilities of the sending and the receiving hosts (An end-to-end responsibility, not the network's).

The key word is packet. The network switches packets, pretty much like the switchboard operator of the old circuit-switched telephone networks switched circuits, though you already know there are a lot of differences between both approaches. We build a packet and submit it to the network for transfer to its destination. The packet is the envelope in the mail analogy, then, where is the letter? The letter we name it *datagram* in the context of this practical exercise, that suffices for the time being.

All in all: we have some message that we want to deliver to some application running on a destination host, the

message gets broken up into separate datagrams, each of which subsequently gets encapsulated into an IP packet, each of these is eventually submitted to the network for transfer to its destination host. How can we illustrate this in a practical manner? The best we can do is make a program that uses the protocols installed in our machine for transmitting information, specifically, we are programming in C against an API that provides us access to the UDP protocol's Service Interface.

A brief introduction to UDP: the simple demultiplexer

The example we are building now is a **ping program**. Recall that the real ping command is based on an Internet control-plane protocol named ICMP; here, we set out to emulate ping by programming against UDP (An Internet layer-3 transport protocol) instead of against ICMP (Also a layer-3 protocol, though not a transport protocol). How come UDP, what protocol is that?

Let's assume that the host where we are programming these examples has a single NIC which IP address is 193.146.101.46 whose DNS name is paloalto.unileon.es. The correctly assigned IP identifies a single host in Internet and each IP packet directed toward it is ultimately delivered to it (If all is functioning correctly). With IP addresses we solve the problem of delivering packets to the correct destination host, but, we need to decide how to deliver the packet's payload to a specific process. We are assuming that the host is running a multitasking operating system like Linux where a multitude **of processes will be running concurrently**. UDP (User Datagram Protocol) is capable of identifying a single process on the destination host, it does so by providing a *name space for processes*.

The User Datagram Protocol (UDP) data unit is named Datagram, every datagram has a source port field and a destination port field, these ports somehow map the originating process to the destination process in the destination host. Your C program runs in a process space in the host operating system and it may acquire a UDP port by creating what is known as a UDP socket. Once your Java program has hold of the UDP socket it can use it to transmit and receive datagrams over it, observe the following sketch of the steps required:

1. Your program acquires a UDP socket on port 50001
2. Now, your program is bound to that port: the UDP socket knows your program and your program knows the UDP socket
3. When your host receives an IP packet that contains a UDP datagram whose destination port is 50001, the datagram's payload (Typically an application-level block of information) will be delivered to your C program.

All in all: An IP's identifies a host and a UDP port identifies a single process running in that host

A ping application based on UDP

We are starting with two base programs written in C, one for the client and the other one for the server. Please, follow the steps below to have the C/S application running:

1. Download client: `$ wget paloalto.unileon.es/cn/labs/datalink/echoClient.c`
2. Download server: `$ wget paloalto.unileon.es/cn/labs/datalink/echoServer.c`
3. Compile both programs:

```
$ gcc –o echoClient echoClient.c

$ gcc –o echoClient echoServer.c
```

4. Request an additional terminal (Shell) from your system
5. Run server in the new terminal window
   ```
   $ ./echoServer
   ```

6. Run client in the old terminal window
   ```
   $ ./echoClient
   ```

If all has been successful so far, you will have to see the client sending 5 messages to the server over the Datagram socket. Observe the server reaction to each received message and how it responds to the client. Finally, confirm that the client receives the response produced by the server.

Exercise 1. Modify the server program so that it really behaves as an echo server, i.e., the response it sends back is the information received from the client.

Exercise 2. Check the C/S protocol by starting a Wireshark trace. Specify a display filter "UDP" which will restrict the displayed frames to those containing UDP datagrams. Investigate how to compose a filter that restricts displayed frames to those that contain UDP datagrams whose source or destination port is the one used in the programs: UDP port 50001.

Exercise 3. Modify the client program so that it requests the server IP and port from the command line. Make sure you properly structure and comment your program.

Exercise 4. Now run your server at a host other than the one on which the client is running. Discuss the results obtained from using Wireshark, more concretely; we want you to observe the differences with the previous trace where both programs were running on the same host (IP addresses must be different). Please, notice how the C/S application runs correctly when both programs run on the same host and when they run on different hosts.

Exercise 5. Observe the trace obtained at the Client and check it against the trace obtained at the Server. You will observe that the data exchanged is exactly the same (Except in the rare case an error took place in our LAN) and that the IP's and Port's involved are interchanged.

Exercise 6. Observe the trace obtained at the client when it sends one of its messages to the server and depict a protocol graph of the protocols activated as the client message is sent. The protocol graph must be coherent with the obtained results. Check that the results make sense according to the contents of the lecture on the UDP protocol.

echoServer.c

```
/*
 * (C) José María Foces Morán
 * Computer Networks 2017
 * Basic, non-reentrant, UDP-based Echo Server
 *
 */

#include <stdio.h>
#include <stdlib.h>

#include <fcntl.h>
#include <memory.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>

#include <signal.h>
#include <errno.h>
#include <sys/time.h>

#define OURPORT 50001
#define REQLENGTH 127

char DATEREQUEST[REQLENGTH + 1];

int MAXITERATIONS = 5;


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

    /*
     * Crear la direcci√≥n de la socket UDP
     */
    struct sockaddr_in server;
    // Socket del tipo 'internet'
    server.sin_family = AF_INET;
    // Elegimos un puerto no reservado
    server.sin_port = htons(OURPORT);
    /* Esta socket (En este server) va a a escuchar en
     * cualquiera de las IPs, reales o virtuales
     */
    server.sin_addr.s_addr = INADDR_ANY;

    /*
     * Se crea una socket internet, del tipo datagrama
     */
    int s;
    s = socket(AF_INET, SOCK_DGRAM, 0); //√öltimo arg, siempre 0


    /*
     * Unimos a la socket s la direcci√≥n que hemos creado en los
     * p√°rrafos anteriores:
     */
    bind(s, (struct sockaddr *) &server, sizeof (server));


    /*
     * Direcci√≥n del cliente que nos va a contactar
     */
    struct sockaddr_in client;
    unsigned int addr_length;
    //addr_length = sizeof (client);

    printf("Servidor esperando a recibir una solicitud\n");
    fflush(stdout);
```

```
    /*
     * Usar la socket s para recibir un mensaje que se almacenar√° en DATEREQUEST y
     * que no ser√° de longitud efectiva superior a REQLENGTH (20 bytes),
     * la direcci√≥n y el puerto del cliente quedar√°n registrados en client
     */
    addr_length = sizeof (client);
    int i = 0;

    while (i < MAXITERATIONS) {
        int nbytes = recvfrom(s, DATEREQUEST, REQLENGTH, 0, (struct sockaddr *) &client,
&addr_length);

        printf("Solicitud recibida con un tama√±o de %u bytes, procedentes de %s@ puerto %u:\n",
                nbytes, inet_ntoa(client.sin_addr), client.sin_port);

        printf("'%s'\n", DATEREQUEST);

        char *response = "Respuesta de prueba";

        /*
         * Usar la socket s para enviar un DATAGRAMA de respuesta al cliente desde
         * el que hemos recibido la solicitud
         */
        nbytes = sendto(s, response, strlen(response) + 1, 0, (struct sockaddr *) &client, sizeof
(client));

        printf("Enviados %u bytes:\n\n", nbytes);

        i++;
    }
}
```

echoClient.c

```
/*
 * (C) José María Foces Morán, Universidad de León
 *
 * Prácticas de Telecomunicaciones en la Industria 2017
 *
 * UDP Echo client: Sends a message 5 times and checks response from server
 *
 */

#include <stdio.h>
#include <stdlib.h>

#include <fcntl.h>
#include <memory.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>

#include <signal.h>
#include <errno.h>
#include <sys/time.h>

int createDatagramSocket() {

    /*
     * Create a sockaddr_in struct that represents the full
     * UDP socket addressing (IP and UDP port number)
     */
    struct sockaddr_in client;
    client.sin_family = AF_INET;
    client.sin_port = INADDR_ANY;
    client.sin_addr.s_addr = INADDR_ANY;

    /*
     * Create a new UDP socket in the AF_INET domain and of
     * type SOCK_DGRAM (UDP)
     */
    int s = socket(AF_INET, SOCK_DGRAM, 0); //Always 0

    /*
     * Bind the socket s to address client
     */
    bind(s, (struct sockaddr *) &client, sizeof (client));

    return s;
}

struct sockaddr_in createServerAddress() {

    /*
     * Create a sockaddr_in struct that represents the full
     * UDP socket addressing for the server
     */
    struct sockaddr_in server;
    server.sin_family = AF_INET;
    server.sin_port = htons(50001);
    //Type the server IP right below
    server.sin_addr.s_addr = inet_addr("192.168.2.107");

    return server;
}

void sendLoop(int s, struct sockaddr_in server) {

    int i = 0;

    while (i < 5) {

        printf("Send iteration %u\n", ++i);

        /*
         * Send mensaje through socket s to UDP server socket
         * whose address is server
         *
         * flags is 0
         */
```

```
        char *message = "Hello world! aeiou abcdefeghijklmnopqrstuvxyz";

        int nbytes = sendto(s, message, strlen(message), 0, (struct sockaddr *) &server, sizeof
(server));

        printf("%u actually sent\n", nbytes);
        printf("%s\n", message);

        fflush(stdout);

        char response[1025];

        struct sockaddr_in addr;
        unsigned int addr_length;

        addr_length = sizeof (addr);
        nbytes = recvfrom(s, response, 1024, 0, (struct sockaddr *) &addr, &addr_length);

        printf("%u bytes received:\n", nbytes);
        response[nbytes] = '\0';
        printf("%s\n---------------\n", response);
        sleep(3);
    }

}

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


    int s = createDatagramSocket();

    struct sockaddr_in server = createServerAddress();

    sendLoop(s, server);

    printf("Client exiting\n");

}
```