Practicals on Computer Networks and Distributed Systems

Simple C/S application written in the C language (WIP)

Introduction to UDP 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 sequence of limited-size pieces before transmission. The resulting pieces are conventionally known as packets, that is, IP packets. 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 5 blocks each comprising 10 pages, for example. Now, each block of pages fits within the envelope size you seem to avail now. Each of the 5 10-page blocks will have to be addressed to your friend. You may recall from past courses that each message, potentially, might be transported over different routes than the others and they might arrive in an order other than the order in which they were originally transmitted. Also, it is worth mentioning now that any of the outgoing letters could be lost by the mail system or get damaged, thereby never being delivered at the end destination. The example just developed briefly represents various aspects of 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 concept 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 -as usual, we must consider network as short for internetwork. We build a packet and submit it to the *network* , where it will be eventually delivered at its destination host. The packet is the envelope in the mail analogy, then, where is the letter? The letter corresponds to the data that is generated by some application, one of those that we frequently run as users. That data, must be encapsulated into a UDP protocol's data unit (in this practical) which we call a *datagram* in this context. In other words, if you want that your application, whichever that might be, send data over the Internet, then it should use UDP, a transport protocol, so that programming the send

turns out to be straightforward. Other, more sophisticated transport protocols are available; we start studying UDP, though, due to its simplicity.

All in all: we have some -data- 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.

UDP: The simple, process multiplexer

The example we are building in this practice aims to imitate the **ping program** that we studied in the Computer Networks course. Recall that the real ping command is based on an Internet control-plane protocol named **ICMP**; here, we set out to emulate ping by doing our programming against the UDP protocol (An Internet layer-3 transport protocol) instead of against ICMP (Also a layer-3 protocol, though not quite 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 address identifies a single host in Internet and each IP packet directed toward it is ultimately delivered to it (If all is functioning correctly). IP addresses help solve the problem of delivering packets to their intended destination host.

> **Once a packet is delivered to its destination host, an deterministic decision should be made about how to deliver the packet's topmost payload to a specific process running in the host.**

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) provides a means of identifying a single process on the destination host that is to receive the topmost payload contained in the packet. UDP does so by providing a *name space for processes*, one that is standardized in the Internet: the port number.

> **By using port numbers, UDP's multiplexing key, UDP allows exposing a selected process in a host to the Internet.**

The User Datagram Protocol (UDP) data unit is known as Datagram. The header of a UDP datagram is composed of four fields, one of which constitutes the **multiplexing key**, the **Destination Port** field. The remaining three fields are the Source Port, The Checksum and the Length (Refer to the relevant lecture presentation for the semantics, the meaning of each). The destination port contained in a given UDP datagram marks the destination process that is to receive the payload encapsulated into the datagram at the destination host's stack.

When programming to UDP's service interface, your program will need acquire a UDP port by creating and configuring what is known as a UDP socket (Datagram socket is the name we'll actually use in this practicals). Once your program has taken hold of the UDP socket it can use it to transmit and receive datagrams over it. The following outline sketches the steps required:

1. Your program acquires a UDP socket on port 60000, for example.

2. Now, your program is bound to that UDP port, in other words, the datagram socket *knows* about your program and, conversely, your program *knows* about the datagram socket

3. When your host receives an IP packet that contains a UDP datagram whose destination port is 60000, the datagram's payload (Typically an application-level block of data) will be delivered to your C program.

All in all: An IP address identifies a host in the Internet, whereas a UDP destination port identifies a process running in that host. Ports are the multiplexing keys employed by the UDP protocol.

A C/S 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. Follow the steps below to have the C/S application running:

1. Download client: `$ wget http://paloalto.unileon.es/ds/lab/echoClientBase.c`
2. Download server: `$ wget http://paloalto.unileon.es/ds/lab/echoServerBase.c`
3. Compile both programs:

```
$ gcc –o cli echoClientBase.c

$ gcc –o srv echoServerBase.c
```

4. Request an additional terminal (Shell) from your system, where you will run the server

5. Run server in the new terminal window; specify port number 60000 for the server to listen to:

```
$ ./echoServer 60000
```

6. Run client in the old terminal window; assume that the server is running in host 192.168.1.100 in our Lab:

```
$ ./echoClient -n 5 192.168.1.100 60000
```

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.

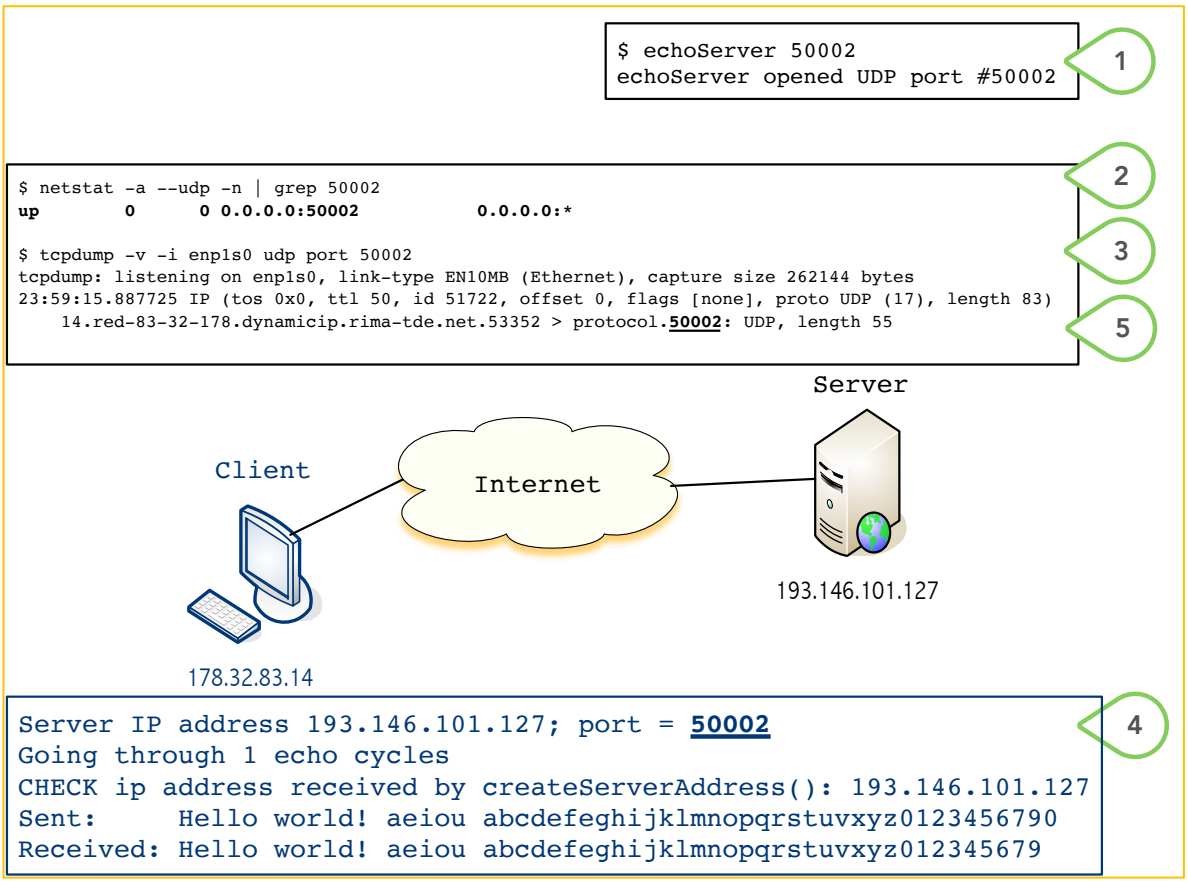

Figure 1. Overall setup of practical and checks using UDP port 50002

Exercises

1. Obtain the full protocol stack that applies to the following communication between a UDP client and a UDP server. The server and the client are implemented by the echoServerBase and the echoClientBase programs given above, respectively.

```
18:06:14.924116 e0:d5:5e:d8:86:a1 > e0:d5:5e:dd:ed:0d, ethertype IPv4
(0x0800), length 97: (tos 0x0, ttl 64, id 59786, offset 0, flags [DF],
proto UDP (17), length 83)

192.168.1.89.42139 > 192.168.1.210.60000: [bad udp cksum 0x84cc ->
0xd03d!] UDP, length 55
      0x0000:  4500 0053 e98a 4000 4011 cc93 c0a8 0159  E..S..@.@......Y
      0x0010:  c0a8 01d2 a49b ea60 003f 84cc 4865 6c6c  .......`.?..Hell
      0x0020:  6f20 776f 726c 6421 2061 6569 6f75 2061  o.world!.aeiou.a
      0x0030:  6263 6465 6665 6768 696a 6b6c 6d6e 6f70  bcdefeghijklmnop
      0x0040:  7172 7374 7576 7879 7a30 3132 3334 3536  qrstuvxyz0123456
      0x0050:  3739 30                                  790
```



```
18:06:14.924469 e0:d5:5e:dd:ed:0d > e0:d5:5e:d8:86:a1, ethertype IPv4
(0x0800), length 97: (tos 0x0, ttl 64, id 10697, offset 0, flags [DF],
proto UDP (17), length 83)

192.168.1.210.60000 > 192.168.1.89.42139: [udp sum ok] UDP, length 55
      0x0000:  4500 0053 29c9 4000 4011 8c55 c0a8 01d2  E..S).@.@..U....
      0x0010:  c0a8 0159 ea60 a49b 003f d03d 4865 6c6c  ...Y.`...?.=Hell
      0x0020:  6f20 776f 726c 6421 2061 6569 6f75 2061  o.world!.aeiou.a
      0x0030:  6263 6465 6665 6768 696a 6b6c 6d6e 6f70  bcdefeghijklmnop
      0x0040:  7172 7374 7576 7879 7a30 3132 3334 3536  qrstuvxyz0123456
      0x0050:  3739 30                                  790
```

2. Compile and run the client and server programs within a single computer. You will have to discover your localhost IP address in order for you to specify it in the client's command line. Observe and document the **C/S protocol** implemented by the client and server programs from scratch (No modification is to be done here). Start an appropriate tcpdump trace, one that includes a filter like "udp port 60000". As usual, deduce the protocol stack resulting from the trace. Depict that **protocol stack and annotate** it with the multiplexing key that applies to each of the protocols that appear on it.

3. Explain how the client program is capable of accepting the server IP address and port as **command line parameters**. The IP address that is supplied to your program is supposed to be formatted in DDN (Dot Decimal Notation; for example, 192.168.1.123). The binary form to a DDN IP address is obtained by calling function inet_aton() – check its man page for details. Likewise, the port is provided as an ASCII string which has to be translated into its expected unsigned integer representation by calling function atoi(). Finally, the bytes from the integer returned by atoi() have to be ordered in the Network Byte Order by calling htons().

4. Observe how the server program **echoes back** the same block of characters that it received from the client

5. Can **multiple clients** obtain echo service from your server? Devise a way to check this practically.

6. Extend the server program by having it send back to the client the iteration number where the servicing of its request happened. The server must send back all of the data that it received from the client plus the value of the iteration counter appended to it. Since the counter is an int, which on the typical platform will have a width of 32 bits (sizeof(int) returns 4, that is, 4 x 8 = 32 bits), it is ordered by following one convention, whichever it is, you'll have to reorder it into the *network byte order* before transmission, and then, when it is received it should be reordered to the receiving host's internal ordering). The client program must be able to access this counter and print it out on the screen.

7. [Preparation for upcoming homework] A UDP server is running in host 193.146.101.46 at port 60007. This server sends back a single short integer as the response to a legitimate client request. The only legitimate client request honored by the server is composed of the following string:

   `"Distributed_Systems_23-24"`

Whenever the server receives that string from a client, it will react by sending back a 16-bit unsigned integer (unsigned short int). If the server receives but the above string, it will send back nothing.

Write a client that complies with the provided specifications. Use tcpdump to observe the sent and the received messages.

