V 1.0 All rights reserved © 2024 by José María Foces Morán and José María Foces Vivancos

Universidad de León
School of Industrial, Computer and Aerospace Engineering
Course on Distributed Systems and Networks

Homework #1: Introduction to Distributed Systems

Submit a single .zip file containing your solutions to the HW exercises. Only .pdf and .c files are accepted. Submit via agora by 21:00 on Friday, 27th-September-2024.

Exercises

1. Introduction to Distributed Systems

1. What is a distributed system made of?

Distributed Systems are made of processes that communicate with each other by exchanging messages. Those processes run on operating systems like Linux and the communication is made possible by way of Internet stacks implemented in them.

2. Interpret the meaning of this phrase: *"The TCP protocol makes omission faults transparent to applications"*

An omission fault happens whenever a system will not provide a response to a given request that it received earlier. Consider a client host that sends some information to a server host, if this information is lost amid its trip to the server, then, Internet has committed an omission fault, because that fault will hinder the server from ever sending back to the client the response. The TCP protocol compensates for these faults by using a number of mechanisms known as ARQ, or Automatic Repeat Request. These mechanisms, in TCP, are based on the use of positive and cumulative acknowledgements. Ultimately, TCP affords that application programmers be oblivious about the happening of omission faults. In other words, TCP makes those faults transparent to application programmers.

2. UDP

1. Download the script to the practical that we did on week #39, titled "Practical on CS with datagram sockets in C" (The link that points to it is http://paloalto.unileon.es/ds/lab/udpcsscript.pdf) and redo exercise 1 in your own Linux host, at your home and having no Internet connectivity. Explain each step you take alongside the final tcpdump trace.

Turn off network interfaces (Ethernet, Wifi, Bluetooth). Execute the server at TCP port 60000, for example. Then, run the client and enter the server IP

address as 127.0.0.1 (Usually, this is your localhost) and port 60000. If you wish, you can capture the generated IP traffic with tcpdump -i 127.00.1, etc. Note that, in this case, the traffic will remain confined in the host's stack, in which, IP will forward the received packets back to the same stack by using the localhost IP interface.

2. **Extend the echoServerBase.c and the echoClientBase.c from the practical mentioned in the previous exercise such that the server returns the actual loop's *iteration number* (An int, which in C, under a mainstream platform is represented with 32 bits) along with the data that should be echoed back to the client.**

   The client should, as well, access that integer when received and have it properly printed out. The loop's source code is located at line number 62, and the counter is represented in an int variable which name is counter. To simplify your work, the following *empty* function has been included in the source code:

```
void appendCounter(char *buffer, unsigned int counter, unsigned int *nbytes){
}
```

   Function appendCounter(), receives all of the parameters necessary for appending the value of counter to the buffer that is to be returned to the client. Consequently, you must code your solution within function appendCounter(), exclusively. In Computer Networks we reviewed how to send an integer over a socket (Functions htonl() and ntohl()).

   Extend the client's code so that it prints the iteration number out every time it sends a new request to the server.

   The server program executes a basic loop where it receives a message and computes and sends back the response. In this exercise, you must modify the server so that it sends back the iteration number in which the response will be sent back. The integer that represents the iteration counter must be translated from the internal byte-ordering used by the specific processor architecture used by the server (Little Endian or Big Endian) to Network Byte order, the ordering used by networks, which is exactly like the Big Endian. Functions htonl() and ntohl() automate these marshalization processes in a consistent way.

   The client, when it receives the new response format, which contains the iteration counter, must print it out on the screen, but, consistently with the above explanation, the client should translate the integer to the internal ordering in use in its processor architecture.