Course on Distributed Systems at Universidad de Leon

School of Industrial, Computer and Aerospace Engineering

DSPro

Independent Study Exercises on Distributed Systems

General guidelines for solving the exercises

- Each exercise is worth 0,20 points out of a total of 1,00 points which is the credit assigned to this homework this academic year (2024-2025). You have no obligation to submit the solutions to all of the exercises; any part thereof will count if properly explained and developed. *The mentioned credit does not contribute to the passing mark of the course, that is the sense in which this practice is named optional.*

- Some exercises require *remote* access to host paloalto.unileon.es.

- The right to submit DSPro and that it be assessed is only granted to those students who have attended the practical lab sessions regularly.

- Program sources must be clearly commented.

- Apply a simple OO design strategy, or structured programming strategy depending on the specific language used in each case in this homework (Java or C, respectively).

- Include rich explanations of your design decisions and the unit tests that check the correction of your programs.

- Only your original work, produced personally by you should be submitted. You can incorporate source code from open software projects, in which case you must cite the authors and their overall weight in your project must be small.

- Submit the solution to **each exercise** in a **separate folder** which name must be "Exercise 1", "Exercise 2", etc. Each directory may contain C source files, Java source files, zip-compressed files, .mk (make build utility) files, .ant (Java ant build tool), or .pdf documentation files.

- Be sure to include the build files that you have used in your project, written either in make, gmake or in Ant.

- Compress the solution to all of the exercises (Their whole file hierarchy) in a single .zip file (***Only .zip is acceptable!!!***)

- Compress the complete folder structure mentioned above in a .zip file (Please, use **.zip exclusively**, otherwise, I might not be able to decompress the archive which might compromise my assessing your work, altogether).

- Submit the zip-compressed archive to the agora task titled ***Homework #4: DS Pro 2024:***

    - ***Publication date: 6th-November-2024 at 17:00***
    - *Submission deadline: 20th-November-2024 21:00*

Homework #4: DS Pro 2024

Apertura: jueves, 14 de noviembre de 2024, 21:30 **Cierre:** miércoles, 20 de noviembre de 2024, 21:35

Figure 1. *Agora task for DSPro 2024. The homework statement will be available at 17:00 on that date*

1. New ping utility for Linux

i. Develop a *ping* utility for the Linux operating system in C based on ICMP echo messages. The utility should at least offer the following functionality:

   a. Send a request every 10 seconds for a period of 1 hr

   b. Printout the Rtt to each request/response cycle

   c. A brief Rtt statistics, including the number of lost packets, the min Rtt, the max Rtt, the avg Rtt and the Rtt variance

   d. Exclusively use host 193.146.101.46 for your unit tests

ii. Explain the greatest difficulties that you have got into in developing this program

iii. Apply a modular and structured programming methodology and have your code appropriately commented.

iv. Draw the protocol stack used by your application-layer tool. Explain the *detailed* semantics to the box representing your protocol.

2. Some routers filter ICMP Timestamp Requests

The ICMP Timestamp message, when sent from clients over Internet, may not eventually be delivered to their destination host (A server) because some of the routers comprising the path to it, would drop ICMP Timestamp Requests. This filtering is configured by network administrators as a cyber security measure, which context is beyond this course scope.

In this exercise, we ask you to suggest *reasonable, sensible and feasible* strategies for circumventing this security restriction.

You might start with observing the similitude between the ICMP headers to the echo messages (Those used by the ping utility) and those to the ICMP timestamp messages, and by carefully studying the RFC to the ICMP protocol.

i. What would be necessary for passing timestamp requests/replies as simple ICMP echo and echo replies?

ii. Discuss how you would develop a *server* program that would treat *our* ICMP echo messages as timestamp requests and then respond to them accordingly.

iii. Explain your strategy for programing this in C, notice however that no actual programming is expected as the solution to this exercise.

iv. Is solving the preceding question possible at all? Discuss whether or not it would be possible to have a *server* program receive IP Protocol 1 (ICMP) messages. Recall that those messages are handed to Linux's ICMP protocol module.

v. Now, write a program that creates a PF_INET/RAW_SOCK socket for receiving ICMP echo messages and have them printed out. Your program should send back no ICMP responses. Check your program in your own Linux PC by sending it ICMP echo requests with the Linux ping utility and using localhost (Usually 127.0.0.1) as the destination IP address.

3. Simple probabilistic time synchronization algorithm

Using ICMP timestamp request and response messages, build a C program that synchronizes its host's clock (The client) with the clock of host 193.146.101.46 (The server).

Initially, the client program will measure the Rtt to the server host for a period of 5 min at every 10 seconds. Along this loop, the program must compute the minimum Rtt observed so far and keep this minimum Rtt.

After computing the minimum Rtt, the program will start a loop of 100 ICMP Timestamp requests to the server, each separated 5 seconds to the next one. In each, the server must measure the Rtt and whenever it is less than the minimum, the program should synchronize the local clock and printout the maximum synchronization error achieved. This synchronization algorithm is due to the late Flaviu Christian. Using his own words, this algorithm is *probabilistic.*

i. **Why is it the case that Christian used the term *probabilistic* in his** original research paper? The paper can be downloaded from within the Unileon virtual campus, and not from paloalto.unileon.es, due to copyright-related regulations. You can fetch the paper from the folder associated with the DS-Pro agora task (SpringerChristianAlgorithm.pdf, which was downloaded from its publisher website, Springer, by using the institutional *Universidad de Leon* subscription.

ii. **Procedure for allowing your time sync client to hand ICMP** Timestamp requests over to host 193.146.101.46.

    The networks that comprise the paths to host paloalto.unileon.es (193.146.101.46) filter ICMP message types 13 and 14 (ICMP Timestamp Request and Response), thereby hindering any form of clock sync based on that protocol. A number of strategies are available to circumvent that limitation of which I have chosen the creation of an end-to-end ssh-based tunnel. These ssh tunnels allow a client application to communicate with the server on the other end by *transparently* encrypting the messages generated by the client, and ultimately having those ssh-encrypted, client-generated messages decrypted by the receiving ssh server and finally handed up to the server application (The kernel-implemented ICMP server). Ultimately, the protocol stack, supplemented at each side with the SSH protocol, enables transparent communication between client and server as though no network stood between the two.

Overall, the low-level detail about ssh tunnels is beyond the scope of our course, consequently, we'll simply deploy the tunnel and use it as what it turns out to be: a virtual interface to a *point-to-point network* that transparently communicates your local host and 193.146.101.46, the target host. Thus, the ICMP packets generated by your clock sync client (By way of an IP RAW_SOCK) will travel encapsulated into SSH-encrypted messages which will not be filtered –*hopefully*.

Each student will be assigned a number from 1 to 75. That number is exclusively assigned to each student and is the only number that they should use in the resolution of the present homework.

1. Find your *personal* number in the table contained in following file by looking up the last 5 digits of your National ID (DNI):

   ```
   $ wget paloalto.unileon.es/ds/tunnels/stnumbers.pdf
   ```

   Notice, henceforth, your personal number will be denoted as **N** throughout this document.

2. The *passphrase* used in the creation of all of the private keys is:

   "`tun-ssh-ds2024`"

   You'll be asked for this *passphrase* later at some point in the progress of this procedure.

3. Switch to **super-user**, either by issuing **su** or **sudo**, depending on your specific configuration. Download your exclusive *private key file*, which name is id_tun**<N>**_rsa. You must replace the mandatory field **<N>** for your own personal number. For example: *assuming that your personal number is **35***, then you would download your personal private key file by executing this command:

   ```
   # wget paloalto.unileon.es/ds/pvkeys/id_tun35_rsa
   ```

4. Continuing by assuming that your student number is **35**, now, set the correct file permissions for the private key file:

   ```
   # chmod 600 id_tun35_rsa
   ```

5. Download the shell (Bash) shell script that creates the virtual interface making up this side of the point-to-point network to 193.146.101.46:

```
# wget http://paloalto.unileon.es/tunnels/student_tun.sh
```

6. Download the bash script that configures the newly created tun, virtual interface:

```
# wget http://paloalto.unileon.es/tunnels/student_link.sh
```

7. Add the execution attribute to the two files that you just downloaded:

```
# chmod +x student*.sh
```

8. Before continuing with this procedure, check that you have end-to-end connectivity with 193.146.101.46 (paloalto.unileon.es) by using the ping utility:

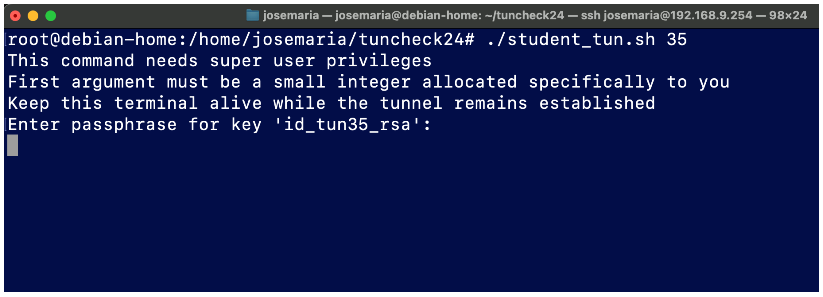
```
# ping 193.146.101.46
```

Check that you are receiving echo backs

9. Create the ssh tunnel to 193.146.101.46; ascertain that you are super-user. We keep using 35 as our *example* student number (Replace it for that which is your actual number!):

```
# ./student_tun.sh 35
```

If no error is issued by this command, the terminal will look idle, and that is the normal behavior at this point (See fig. 2). Leave that terminal in that state, now. Here, I show what happened in my own Linux, at home:

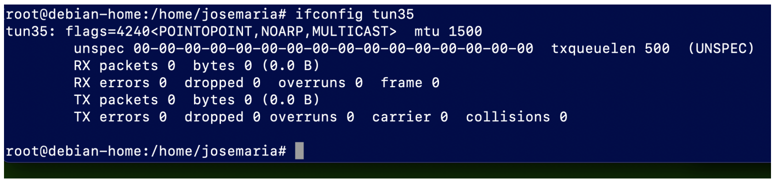

josemaria — josemaria@debian-home: ~/tuncheck24 — ssh josemaria@192.168.9.254 — 98×24

```
root@debian-home:/home/josemaria/tuncheck24# ./student_tun.sh 35
This command needs super user privileges
First argument must be a small integer allocated specifically to you
Keep this terminal alive while the tunnel remains established
Enter passphrase for key 'id_tun35_rsa':
```

Figure 2. Idle terminal after creation of the ssh tunnel

10. Request a new terminal and switch to super-user on it (su or maybe sudo), then, execute ifconfig tun35 to check the state of the created interface.

```
# ifconfig tun35
```



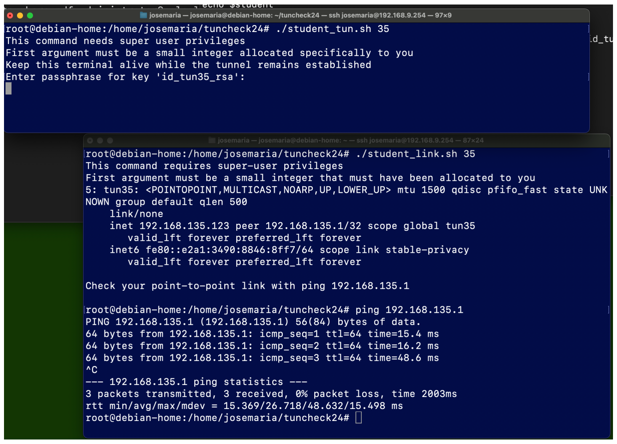
```
root@debian-home:/home/josemaria# ifconfig tun35
tun35: flags=4240<POINTOPOINT,NOARP,MULTICAST>  mtu 1500
        unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00  txqueuelen 500  (UNSPEC)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@debian-home:/home/josemaria# 
```

Figure 3. A new super-user terminal printing out the state of interface tun**35**

Observe that interface tun35 is not UP. We bring it up in the next step.

11. Execute script student_link.sh to configure the brand new, virtual network interface (tun35 in this example). Again, I show you what happened in my own home system:

    `# ./student_link.sh 35`



```
root@debian-home:/home/josemaria/tuncheck24# ./student_tun.sh 35
This command needs super user privileges
First argument must be a small integer allocated specifically to you
Keep this terminal alive while the tunnel remains established
Enter passphrase for key 'id_tun35_rsa':
```

```
root@debian-home:/home/josemaria/tuncheck24# ./student_link.sh 35
This command requires super-user privileges
First argument must be a small integer that must have been allocated to you
5: tun35: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNK
NOWN group default qlen 500
    link/none
    inet 192.168.135.123 peer 192.168.135.1/32 scope global tun35
       valid_lft forever preferred_lft forever
    inet6 fe80::e2a1:3490:8846:8ff7/64 scope link stable-privacy
       valid_lft forever preferred_lft forever

Check your point-to-point link with ping 192.168.135.1

root@debian-home:/home/josemaria/tuncheck24# ping 192.168.135.1
PING 192.168.135.1 (192.168.135.1) 56(84) bytes of data.
64 bytes from 192.168.135.1: icmp_seq=1 ttl=64 time=15.4 ms
64 bytes from 192.168.135.1: icmp_seq=2 ttl=64 time=16.2 ms
64 bytes from 192.168.135.1: icmp_seq=3 ttl=64 time=48.6 ms
^C
--- 192.168.135.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 15.369/26.718/48.632/15.498 ms
root@debian-home:/home/josemaria/tuncheck24# 
```

Figure 4. The lower terminal configures tun35 and checks the tunnel

12. After successfully setting up the tunnel, your Linux stack has a new IP interface tun**35** that allows your host to communicate with 193.146.101.46, transparently by using these two IP addresses:

    - Your local IP address is 192.168.1**35**.123
    - Your remote IP address is 192.168.1**35**.1

    Notice that if your **N=35**, your local IP address is 192.168.1**35**.123 and the remote is 192.168.1**35**.1. Be careful to use your correct and exclusive IP addresses, as much for the client as for the server, as explained above.

iii. Procedure for finally checking your time sync client program:

a. Download the base .c source file included in the time sync practical.

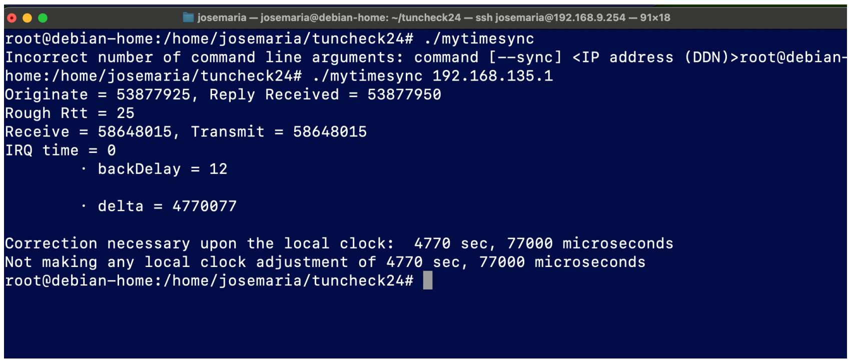
That program, as I demonstrated in the practical lab session in Lab B6, computes the correction necessary for the local clock to be in sync with the specified host's clock and prints it out. Optionally, the program can enforce the correction upon the local clock (Option -w).

This .c source file is your basis code for implementing the Christian's algorithm.

b. Compile the program and run it in a new terminal. Switch to super-user, before running it. Also, make sure that the tunnel, for student 35 that we created above is up and running.

```
# wget http://paloalto.unileon.es/ds/lab/icmptimestamp.c

# gcc -o mytimesync icmptimestamp.c

# ./mytimesync 192.168.135.1
```

See the illustration in fig. 5

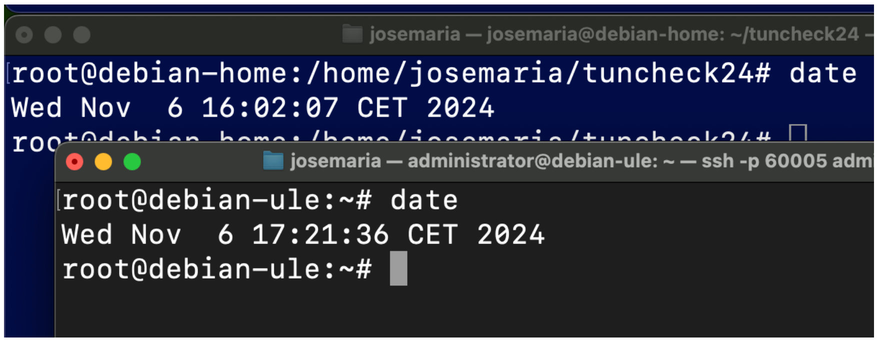

josemaria — josemaria@debian-home: ~/tuncheck24 — ssh josemaria@192.168.9.254 — 91×18

```
root@debian-home:/home/josemaria/tuncheck24# ./mytimesync
Incorrect number of command line arguments: command [--sync] <IP address (DDN)>root@debian-
home:/home/josemaria/tuncheck24# ./mytimesync 192.168.135.1
Originate = 53877925, Reply Received = 53877950
Rough Rtt = 25
Receive = 58648015, Transmit = 58648015
IRQ time = 0
        · backDelay = 12

        · delta = 4770077

Correction necessary upon the local clock:  4770 sec, 77000 microseconds
Not making any local clock adjustment of 4770 sec, 77000 microseconds
root@debian-home:/home/josemaria/tuncheck24# 
```

Figure 5. Illustration of results of running the basic time sync program

Observe that the offset between the local and the remote clocks is quite large: 4770,770 seconds. The reason is that I artificially set a time at the client quite different than the time set at the server, for the illustration purposes. See fig. 6.

In summary, the provided .c code allows you to send an ICMP TimeStamp Request, receive the ICMP TS response, calculate the Rtt and printout the offset between the two clocks. This is the code that you'll extend for implementing the Christian's algorithm. Note that before the program can adjust the local time, you will have to stop the NTP client that must be running in your host. If you don't stop NTP, then your local clock will be roughly in sync with the timeserver, and consequently, you'll not be able to see the effect of your implementation of Christian's algorithm. Continue to the next step, then.


josemaria — josemaria@debian-home: ~/tuncheck24
josemaria — administrator@debian-ule: ~ — ssh -p 60005 adm

```
root@debian-home:/home/josemaria/tuncheck24# date
Wed Nov  6 16:02:07 CET 2024
```

```
root@debian-ule:~# date
Wed Nov  6 17:21:36 CET 2024
root@debian-ule:~#
```

Figure 6. Artificially large offset between client (Blue background) and the server (Black background).

c. Stop your NTP client altogether. This operation and changing the clock entail your super-user credentials, thus you must switch to super-user, now. Explain what you do to stop ntp by consulting the time sync practical that we did in Lab B6 where we provided you details about how to stop NTP.

d. Artificially set your local clock to 5 minutes ahead the current real time. Highlight the Linux commands involved in setting that local clock time.

e. Now, run the base .c (mytimesync) program so that it slows down your clock. Soon, the local time will be in sync with the server's. Check this after 15 min, for example.

f. Now, repeat steps d and e above, but set your clock in sync by running your implementation of Christian's algorithm. Explain what tests you will perform to prove that the program functions correctly.

4. RMI C/S application

Write a Java RMI client that checks access to two remote methods at the host at your own Linux PC (IP address 127.0.0.1). The basic signatures to the methods exported by the remote object are **longStringHash**(String) and **factorial**(int). The server and the remote rmiregistry are deployed in your own server host, also. Download the zip file containing the source tree and, afterwards unzip the zip archive:

1. You must have Java JRE and JDK installed.

2. Download zip from:

   ```
   wget http://paloalto.unileon.es/ds/lab/ant.zip
   ```

3. Unzip the archive:

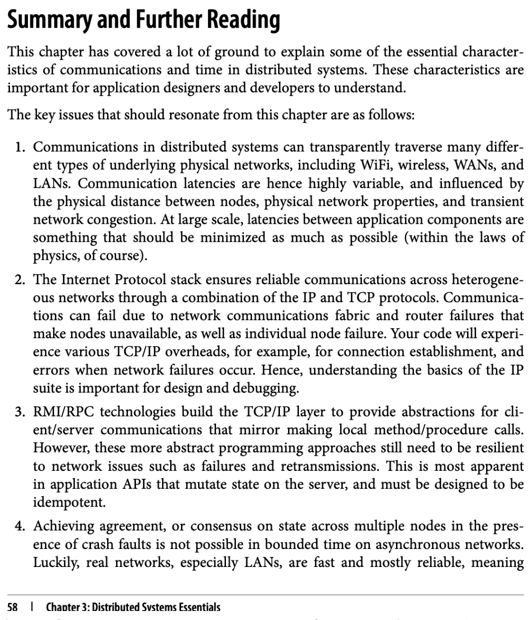
   ```
   $ unzip ant; cd ant; ./install.sh
   ```

4. Skim the ant script file and use the clauses that run rmiregistry, deploy the server and, finally check the client against the server.

5. Document the results that you have obtained.

5. Brief summary about transparencies, clocks, states and faults in distributed systems.

Study the following summary of chapter no. 3 from the book "Foundations of Distributed Systems, by Ian Gorton and published by O'Reilly (Copyright © 2022 Ian Gorton. All rights reserved). **Do a summary of that summary** by remarking what you think is of paramount importance and what you think that might inaccurate or somehow incomplete.



Summary and Further Reading

This chapter has covered a lot of ground to explain some of the essential characteristics of communications and time in distributed systems. These characteristics are important for application designers and developers to understand.

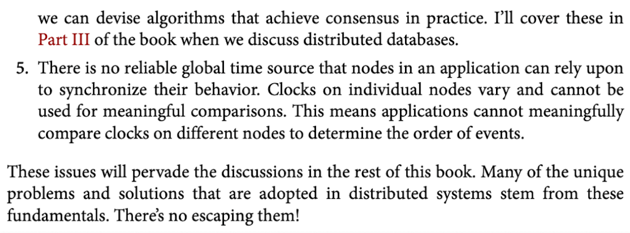
The key issues that should resonate from this chapter are as follows:

1. Communications in distributed systems can transparently traverse many different types of underlying physical networks, including WiFi, wireless, WANs, and LANs. Communication latencies are hence highly variable, and influenced by the physical distance between nodes, physical network properties, and transient network congestion. At large scale, latencies between application components are something that should be minimized as much as possible (within the laws of physics, of course).
2. The Internet Protocol stack ensures reliable communications across heterogeneous networks through a combination of the IP and TCP protocols. Communications can fail due to network communications fabric and router failures that make nodes unavailable, as well as individual node failure. Your code will experience various TCP/IP overheads, for example, for connection establishment, and errors when network failures occur. Hence, understanding the basics of the IP suite is important for design and debugging.
3. RMI/RPC technologies build the TCP/IP layer to provide abstractions for client/server communications that mirror making local method/procedure calls. However, these more abstract programming approaches still need to be resilient to network issues such as failures and retransmissions. This is most apparent in application APIs that mutate state on the server, and must be designed to be idempotent.
4. Achieving agreement, or consensus on state across multiple nodes in the presence of crash faults is not possible in bounded time on asynchronous networks. Luckily, real networks, especially LANs, are fast and mostly reliable, meaning

58 | Chapter 3: Distributed Systems Essentials

Figure 7. Summary of ch.3 from *Foundations of Distributed Systems, by Ian Gorton and published by O'Reilly (Copyright © 2022 Ian Gorton. All rights reserved)* , *part 1/2.*

(Continued on next page)



we can devise algorithms that achieve consensus in practice. I'll cover these in Part III of the book when we discuss distributed databases.

5. There is no reliable global time source that nodes in an application can rely upon to synchronize their behavior. Clocks on individual nodes vary and cannot be used for meaningful comparisons. This means applications cannot meaningfully compare clocks on different nodes to determine the order of events.

These issues will pervade the discussions in the rest of this book. Many of the unique problems and solutions that are adopted in distributed systems stem from these fundamentals. There's no escaping them!

Figure 8. Summary of ch.3 from *Foundations of Distributed Systems, by Ian Gorton and published by O'Reilly (Copyright © 2022 Ian Gorton. All rights reserved), part 2/2.*

(End of document)