DSPro

Optional Practice on Distributed Systems

General guidelines for solving the exercises

- Each exercise is worth 0,25 points out of a total of 1,00 points which is the credit assigned to the present homework in the course. You have no obligation to deliver the full homework assignment; 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 optional.*

- According to the *Syllabus of DS* for the current academic year, the right of submitting DSPro is only granted to those students who are attending the practice lab sessions regularly.

- Program sources must be clearly commented.

- Apply a simple OO design strategy or structured programming strategy depending on whether you are programming in Java or in C, respectively.

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

- You can only submit your original work, programs and explanations. You can incorporate source code from open software projects, in which case you must cite the authors and their overall weight must be small.

- Submit the solution to **each exercise** in a **separate folder** which name must be "Exercise 1", "Exercise 2", etc. Each folder must contain one folder for the sources (**src**), another folder for the explanations (**docs**) and a last folder for the executable files (**bin**). If necessary, include a build **dir**, also.

- Include build files, written in make, gmake or in Ant.

- 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 hamper your grade).

    - Submit the zip-compressed archive to **the agora task titled DSPro**

1. Simple probabilistic time synchronization algorithm

Using ICMP timestamps, build a C program that synchronizes its host's clock (The client) with the clock of another host of your choosing (The server). The client program will fetch the server's time several times and each time it will calculate the achieved Rtt and synchronize the clock only considering the minimum Rtt of the different attempts (Review the Christian's algorithm).

The program must print out the following items:

i. The delta obtained at each timestamp request

ii. The mean delta; the std deviation; the minimum Rtt

iii. The local time before invoking adjtime()

iv. The local after invoking adjtime()

a. Stop your NTP client altogether. Explain what you do to stop ntp. Consult the practice that we did in the course practices where we provided you details about how to stop NTP.

b. Highlight the Linux commands involved in managing the local clock that you used to perform the tests.

c. How long does adjtime() take for reaching a target time that is 5 min forward? Devise an experiment to demonstrate that your results are reasonable.

d. Explain what tests you will perform to demonstrate that the program functions correctly.

e. Use paloalto.unileon.es for synchronizing your clock.

2. RFC 868 time protocol

Skim the RFC 868 time protocol and implement the following RFC 868 C/S pairs:

Implemented over	Client	Server
TCP	C	Java
UDP	Java	C
JRMP	Java RMI	Java RMI

a. Deliver each C/S pair in a separate folder which containing the corresponding sources, executable code and documentation

b. Provide extensive explanations of the considerations and problems that you found in this question

c. Explain what role *marshalization* plays in this exercise (Consult the presentation on *marshalization* in paloalto.unileon.es)

d. Explain the tests that you designed to check the code

3. Vector clocks

Develop a distributed application that simulates three processes each of which has a *vector clock*. Each process can communicate with every other process in which case the sending process piggybacks its own vector clock and the receiving process computes its new vector clock by applying the implementation guidelines given in the notes on *vector clocks.*

a. Provide an extensive discussion of your software design and the tests that demonstrate that it functions correctly. The implementation can be done in any programming language and computing infrastructure.

b. Highlight the core difficulties involved in this tiny distributed project

4. Correcting a run-time error in the server from the MT C/S practice

In the practice about C/S with Stream sockets we provided two implementations, the first was Single-Threaded and the second one was Multi-Threaded:

- Overall concepts about Stream sockets-based C/S and single-threaded implementation of server:

  http://paloalto.unileon.es/ds/lab/pract3.pdf

  http://paloalto.unileon.es/ds/lab/StreamSock-API.pdf

- Multithreaded implementation of server:

  http://paloalto.unileon.es/ds/lab/serverSolution.c

The multithreaded implementation *may* produce a run-time error when many connection requests are received in a relatively small time. Every time the server receives a new connection request, it creates a new thread and passes it *the memory address (A pointer to)* of the int variable that holds the delegate socket. When a *large* number of connections are received in a relatively short time, it is possible that the variable is *unexpectedly changed* by a *young thread* while one other thread is still using for implementing the application-layer protocol. This will create a number of undesirable effects not the least stale connections that will never be fully closed, for example. Follow the outline below here to profiling this problem better and providing a solution to it:

a. Explain the problem with your own words.

b. Create an environment for reproducing the problem.

c. Document the results that you have obtained.

d. Devise a solution to this problem, implement it and demonstrate that it works.