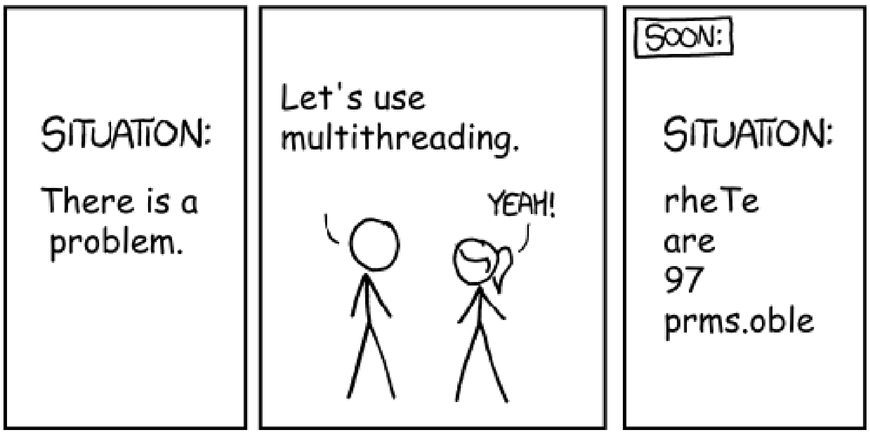
GROUP T LEUVEN CAMPUS

Operating Systems
Project labs: step 3

A multi-threaded server



Bachelor Electronics/ICT

Course coordinator: Bert Lagaisse

Lab coaches:

- Ludo Bruynseels
- Toon Dehaene

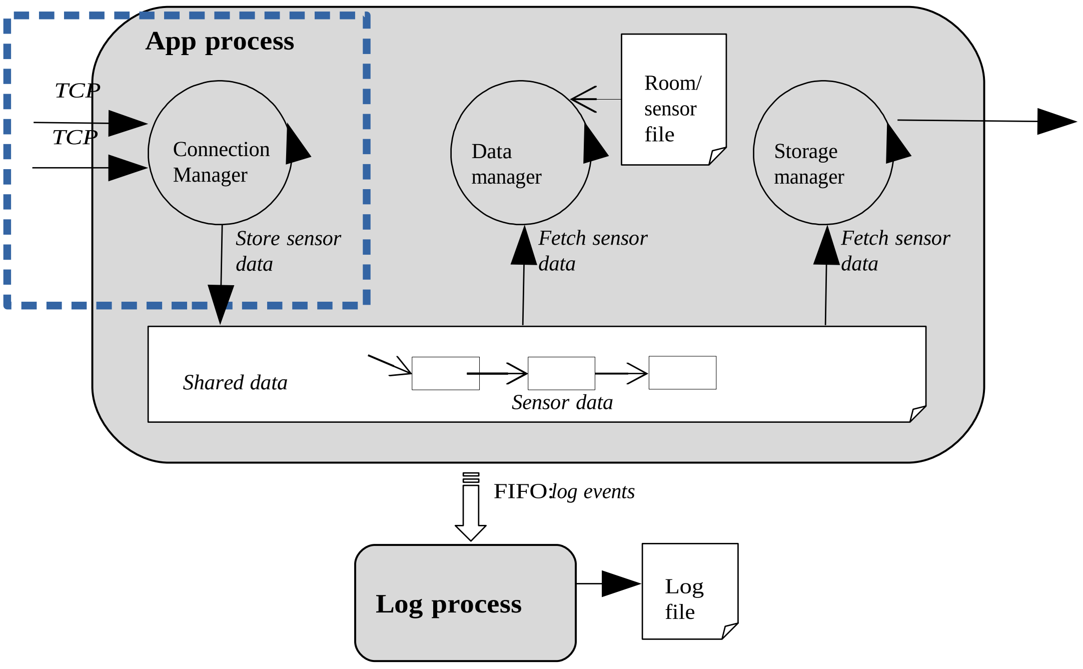
Last update: juli 12, 2024

Project labs: step 3 – A multi-threaded server

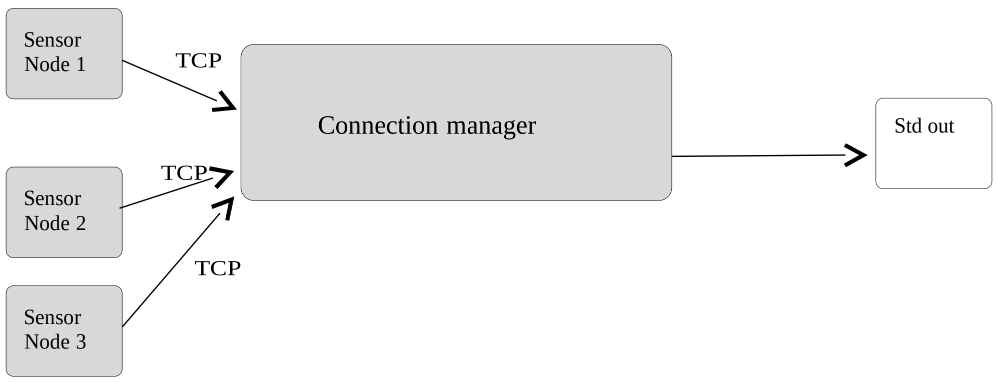
Lab target 1: Learn how to develop processes supporting TCP/IP communication.
Lab target 2: Apply the concepts of multi-threading to a server-side process to handle communication of different clients in parallel.

1. Project overview

The relationship of this lab to the final assignment is indicated by the blue line.



2. Exercise: Processes, threads and network communication



The scope of this lab is a server-side process with a connection manager that listens for incoming sensor data from client-side sensor nodes and writes all incoming sensor readings on the console (std out). TCP sockets are used for the communication between the connection manager and the sensor nodes. For testing purposes, the local loopback network 127.0.0.1 is used for all communication. A sensor node opens a TCP connection to the server and this TCP connection remains open during the entire lifetime of the sensor node process. All data (data format as defined in previous labs

and in the client code) is sent over this TCP connection.
The code for a client-side sensor node is given, together with example code of a simple server. In the given server code, however, only one client connection is supported at the same time. The main problem with the code is that it is single-threaded.
A sensor node is started with the sensor node ID, the sleep time (in seconds) between two measurements, the IP-address and port number of the server as command line arguments (e.g.: `./sensor_node 101 2 127.0.0.1 1234`). This will start a new sensor node that measures every 2 seconds the temperature and sends the result tagged with sensor ID 101 and a timestamp to the server listening at 127.0.0.1:1234.

Specifically, you have received the following code:

- sensor_node.c : a client-side program that sends sensor readings to a given server. You should not change this code. It is for testing purposes only.
- test_server.c : a server-side process that listens for incoming client connections and then continuously reads sensor readings from one client until the client is disconnected. This is the code you need to change and extend.
- lib/tcpsock.h and lib/tcpsock.c : We provide you this abstraction layer over the C socket API, and it is already used by the client and server above to communicate. This part of the code should not be changed.
- Makefile : a make file to build the client and server, to run a server and to run two clients. You should run the server and the clients on different terminals. If you use CLion, the test_server target is the target you will compile and debug. Make sure you configure the binary file and runtime arguments to use during debugging.

Implement the server's connection manager such that it is able to handle multiple TCP connections at the same time. The server should be started as a terminal application with a commandline argument to define the port number on which it has to listen for incoming connections, as well as the maximum number of clients it will handle before the server shuts down. E.g.: `./test_server 1234 3` starts a server with a connection manager listening on port 1234. The server will shut down when the 3rd client disconnects. All three clients must be served continuously until they disconnect.

Moreover, in the provided solution only one process with one thread is used to handle multiple TCP clients. Each time a new TCP connection request is received, the accept() socket call will return a new socket to handle this TCP connection. That means that the main server-side process has to listen to a set of sockets (one for each connected sensor node and one to listen for new connection requests) at the same time. Remember that socket operations (send, receive, accept, …) are by default working in blocking mode such that a problem arises when the server's connection manager is blocked on one socket while at the same time there is incoming data on another one.

Summary of the requested extensions:
1) Make sure you configure the maximum amount of client connections and the port that the server is listening on as a commandline argument when debugging with CLion.
2) Design and implement a solution using multi-threading to solve the problem that only one client is served at the same time. What are the different tasks and how many threads do you need for them? When do you create the thread ?
3) The server shuts down after it has served MAX_CONN client connections. Think about using a counter to keep track of the number of connections in a **thread-safe** manner. Also don't forget to link in the *pthread* library.