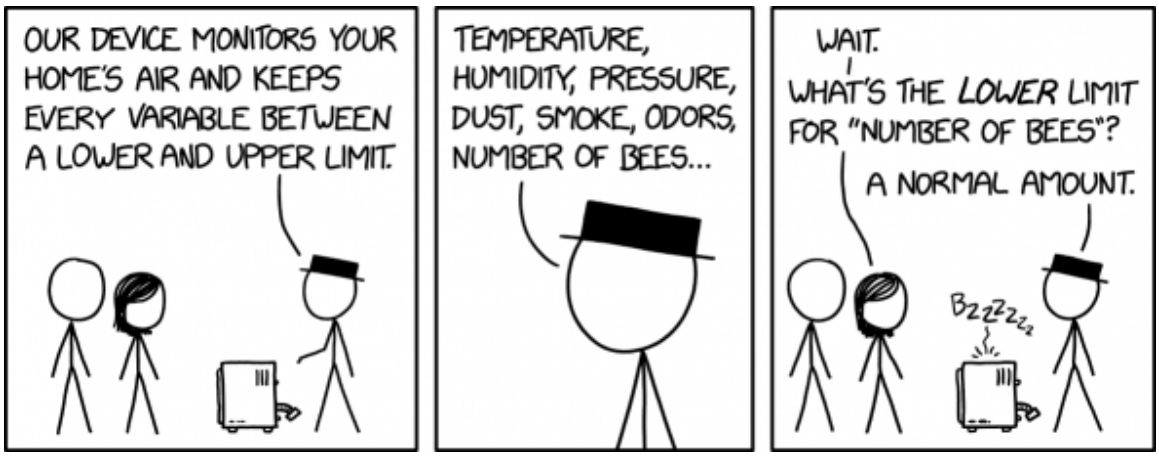
KU LEUVEN

GROUP T LEUVEN CAMPUS

Operating Systems

Final exam project: 2024



Bachelor Electronics/ICT

Course coordinator: Bert Lagaisse
Lab coaches:

- Ludo Bruynseels, Toon Dehaene

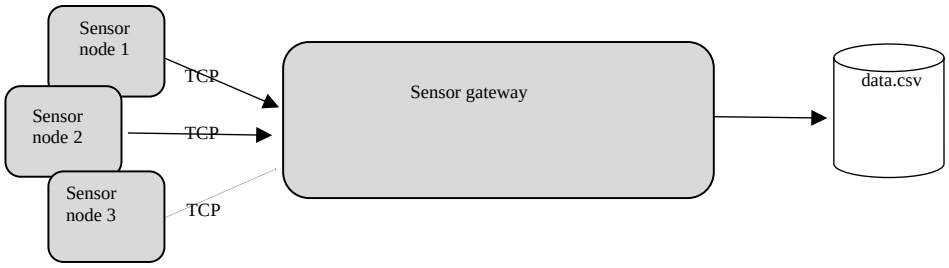
Last update: December 8, 2024

Academic Integrity

*This is an **individual** assignment! It is only allowed to submit **your own work**. You may discuss the assignment with others but you are not allowed to share work or use (part of) another's solution. If you include work (e.g. code, technical solutions,...) from external sources (internet, books ...) into your solution, you must clearly indicate this in your solution and cite the original source. If two students present very similar solutions, no distinction will be made between the 'original' and the 'copy'.*

Sensor Monitoring System

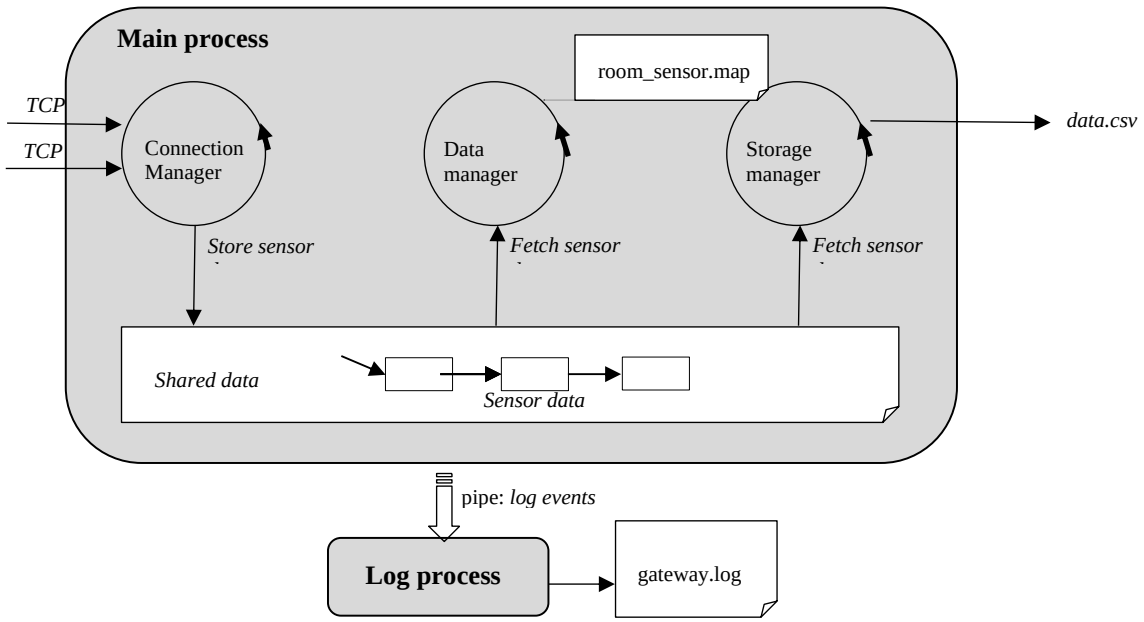
The sensor monitoring system consists of client-side sensor nodes measuring the room temperature, and a central server-side sensor-gateway that acquires all sensor data from the sensor nodes. A sensor node uses a TCP connection to transfer the sensor data to the sensor gateway. The full system is depicted below.



Working with real embedded sensor nodes is not an option for this assignment. Therefore, sensor nodes will be simulated in software using a client-side sensor-node (see sensor_node.c from plab3, which you can use to test your server implementation).

Sensor Gateway

A more detailed design of the sensor gateway is depicted below. In what follows, we will discuss the minimal requirements in more detail.



Minimal requirements. Read this completely before you start coding.

Req 1. The sensor gateway consists of a *main process* and a *log process*. The log process is started as a child process of the main process. Both processes run continuously.

Req 2. The main process runs three threads at startup: the *connection manager*, the *data manager*, and the *storage manager* thread. A shared data structure (the *sbuffer* from plab4) is used for communication between all threads. Notice that read/write/update-access to the shared data needs to be *thread-safe*! You are allowed and encouraged to change and extend both the *sbuffer.h* header file as well as the *sbuffer.c* implementation of the sbuffer module to fit your implementation of the server.

Req 3. The *connection manager* listens on a TCP socket for incoming connection requests from new sensor nodes. 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.: ./sensor_gateway 1234 3 starts a server with a connection manager listening on port 1234. The server will shut down after the 3rd client disconnects and all data is processed. All three clients must be served continuously until they disconnect. For each client-side node communicating with the server, there should be a *dedicated thread* to process incoming data at the server. This is similar to what you implemented in plab3.

Req 4. The *connection manager* captures incoming data from sensor nodes as defined in plab3. Next, the connection manager writes the data to the shared data structure. At compilation time, a *time-out* is defined in seconds (see Makefile). If the sensor node has not sent new data within the defined time-out, the server closes the connection.

Req 5. The *data manager* thread implements the server intelligence as defined in plab1. The room-sensor mapping is read from a text file "*room_sensor.map*". The data manager only reads sensor measurements from the shared in-memory data buffer, and no longer from a file. It further calculates a running average on the temperature and uses that result to decide on 'too hot/cold'. It doesn't write the running average values to the shared data buffer – it only uses them for internal decision taking.

Req 6. The *storage manager* thread reads sensor measurements from the shared data buffer and inserts them into a csv-file "*data.csv*" (see plab1). A new, empty data.csv should be created when the server starts up. It should not be deleted when the server stops.

Req 7. The *log process* receives log-events from the main process using a *pipe*. All threads of the server process can generate log-events and write these to the pipe. This means that the pipe is shared by multiple threads and access to the pipe must be *thread-safe*.

Req 8. A *log-event* contains an ASCII info message describing the type of event. For each log-event received, the log process writes an ASCII message of the format <sequence number> <timestamp> <log-event info message> to a new line in a log file called "*gateway.log*". Do not include "<'' and ''>" in the log lines. Each time the server is started, a new empty gateway.log file should be created. Hence, the sequence number should only be unique within one execution of the server. The log file should not be deleted when the server stops.

Req 9. At least the following log-events need to be supported:

1. From the connection manager:
   a. Sensor node <sensorNodeID> has opened a new connection[1]
   b. Sensor node <sensorNodeID> has closed the connection
2. From the data manager:
   a. Sensor node <sensorNodeID> reports it's too cold (avg temp = <value>)
   b. Sensor node <sensorNodeID> reports it's too hot (avg temp = <value>)

1 Remark that the sensor ID is only known after the first data packet is arrived.

c. Received sensor data with invalid sensor node ID <node-ID>

3.From the storage manager:

a. A new data.csv file has been created.

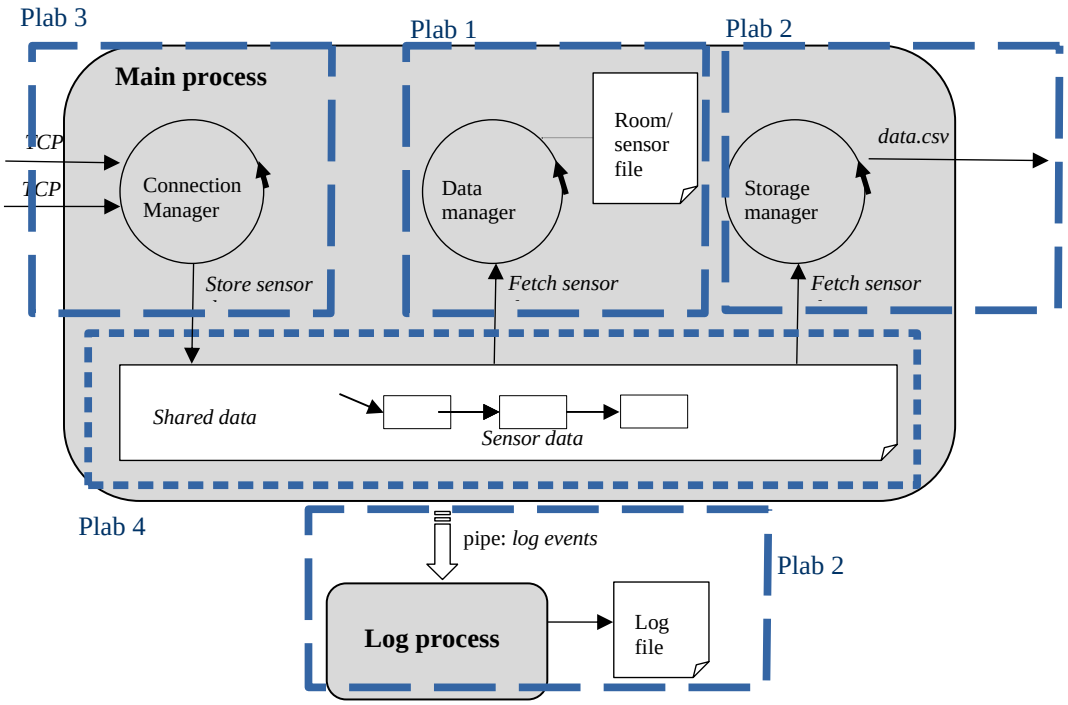
b. Data insertion from sensor <sensorNodeID> succeeded.

c.The data.csv file has been closed.

Req 10. You are not allowed to use semaphores, only mutexes and condition variables.

How to start? Hints, tips and possible approach.

1. Your main task is the development of the full server-side sensor gateway. The sensor_node client should not be changed and only should be used for testing your server.
2. The sensor gateway is a kind of integration result of the code of clab3 and plab1 to plab4. The picture below shows the relationship between the different components of the sensor monitoring system and the lab sessions.
3. You only need your *dplist* implementation inside the data manager. If it is not fully functional yet, think about a fall-back strategy to mitigate this annoying dependency. You can always integrate it later when the overall server is finished.
4. Read all the requirements carefully and underline the main keywords.
5. Make a development plan on which requirements you will tackle first. Prepare additional, intermediary build targets in the make file to develop the server incrementally, step by step. Define your own milestones and a timing for them. Don't try to integrate all code at once. You will find yourself removing code again to detect a bug in your full implementation.
    1. E.g, you could first try to integrate the connection manager with the sbuffer.
    2. E.g, then the data manager with the sbuffer as well as storage manager and sbuffer.
    3. You can still use the input files from the previous labs to test the partial integrations, before you put everything together.
6. It should be clear from the description of the requirements which pieces of code of the previous labs you can reuse, with some changes, for the sensor gateway. You can change the implementation and headers of all predefined files, but the full implementation should be contained in the predefined files (see Makefile and next section on deliverables).
7. Prepare tests to validate your intermediary steps during your development process. Don't wait with the testing until you integrated all code of all requirements. Use the debugger to validate assumptions you made. Printf-debugging will overwhelm your output screen after a while.



Deliverables and acceptance criteria

In the Makefile of the provided source code you will find a description of what exactly and in which format (exact source code files, directory structure, build targets, etc.) you need to prepare.

Once you have finished the assignment, you upload your solution on **Toledo** as **lab_final.zip**. Use the **zip** target in the **Makefile** to produce this zip file. The source code files must follow the names and folder structure as defined in this zip file (including the *lib* sub-folder):

- main.c
- connmgr.c
- connmgr.h
- datamgr.c
- datamgr.h
- sbuffer.c
- sbuffer.h
- sensor_db.c
- sensor_db.h
- config.h
- **lib**/dplist.c
- **lib**/dplist.h
- **lib**/tcpsock.c
- **lib**/tcpsock.h

Your solution should build using the exact build target '***all***' in the make file. You will need to respect the exact source code file names, and you should not add any code files. Your code for the full solution must be implemented in the predefined source code files. Otherwise it will not build or run during our tests.

Our testing strategy
To test the quality of your code, we will first use the "make all" build target to do a strict compilation of your code with the most strict options (-Wall -std=c11 -Werror).
We then check two major test scenarios, one with 3 parallel clients and one with 5 parallel clients. The 3 parallel clients will be launched with a few seconds of time in between. The 5 parallel clients will be launched immediately after each other. These test scenarios are available in the student source code as test3.sh and test5.sh.

Changing the client code for other Linux distro's

While you are not allowed to change the behaviour of the provided client, you might need to add some includes or compiler directives to compile the client on your specific Linux system, e.g. using the -std=c11 option in the makefile. Moreover, the srand48 function might cause problems on some Linux distro's, but adding the _GNU_SOURCE directive in the beginning of the test client will solve this:

```
#define _GNU_SOURCE
```

Test Thoroughly
Write test code – also non-trivial test code – yourself and thoroughly test your code before uploading your solution. For instance: test your code with multiple sensor nodes running concurrently using very small sleep times between 2 measurements.
Include a *debug* target in your Makefile where you compile all source files directly including debug info. Use these binaries with debug info to test and debug your solution (using Clion). Using only *build and run* in your IDE might hide or mask compilation errors like forgotten includes or undefined variables. Build with make and the provided Makefile. We already included some additional build targets in the Makefile to get you started (e.g. sensor_gateway_debug).

Your solution will only be accepted for grading if the following criteria are fulfilled:

- It is submitted on **Toledo, in the correct assignment, before the deadline as specified on Toledo;**
- Your solution **compiles, runs and generates at least some meaningful output** (e.g. measurements in the data.csv file, some log lines in gateway.log, ...);
- It **compiles and runs for at least the scenario with 3 clients (test3.sh)**. This is the bare minimum to pass this project.
- Your solution must be a reasonable try to implement the minimal requirements. This excludes, for instance, solutions that consist of an (almost) empty .c file, incomplete code, or code that has no or very little relationship to the exercise, code that implements a different assignment (e.g. only milestone 2 or 3, or, e.g. from a previous academic year), etc.;
- Your source code is structured and readable. The following, for example, is not acceptable: you don't logically structure the code into source and header files, you apply bad naming of variables and functions, …