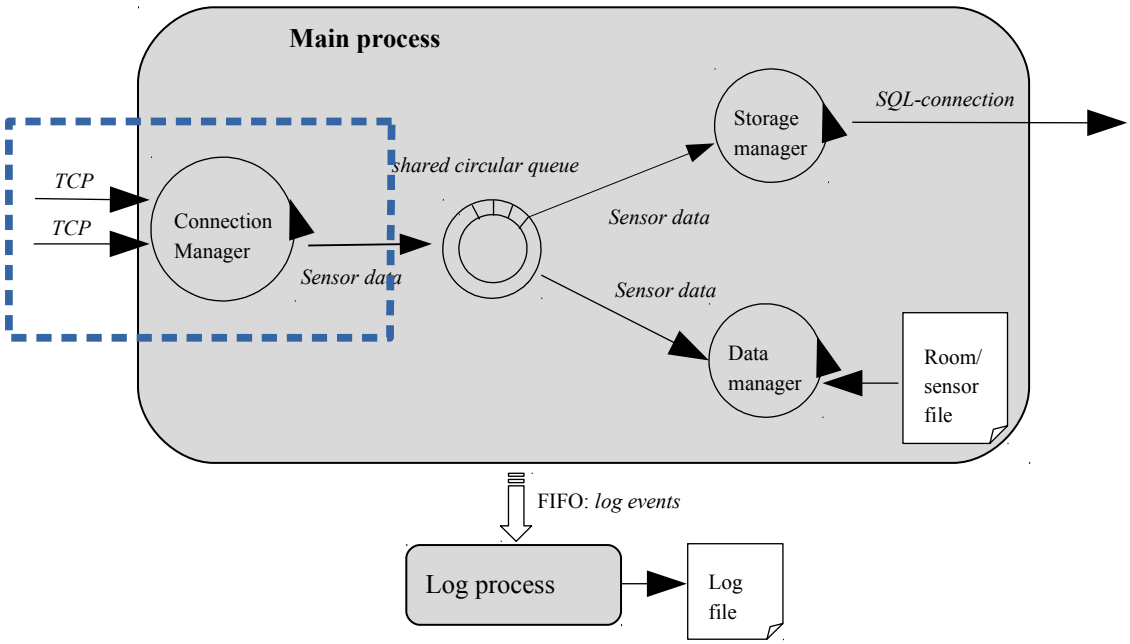
Lab 8 – Programming with Processes

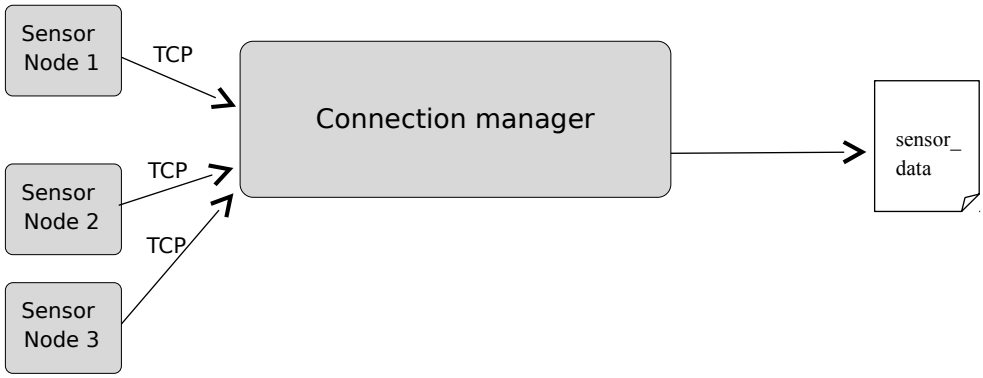
For your information

The picture below visually sketches the final assignment of this course. The relationship of this lab to the final assignment is indicated by the dashed blue line.



Exercise: Processes and network communication

Implement a connection manager that listens for incoming sensor data from sensor nodes and writes all processed data to a sensor data file. TCP sockets are used for the communication between the connection manager and the sensor nodes. We refer to the course "Data communication and computer networks" for more information on TCP and socket calls. For testing purposes, the local loopback network 127.0.0.1 is preferred.

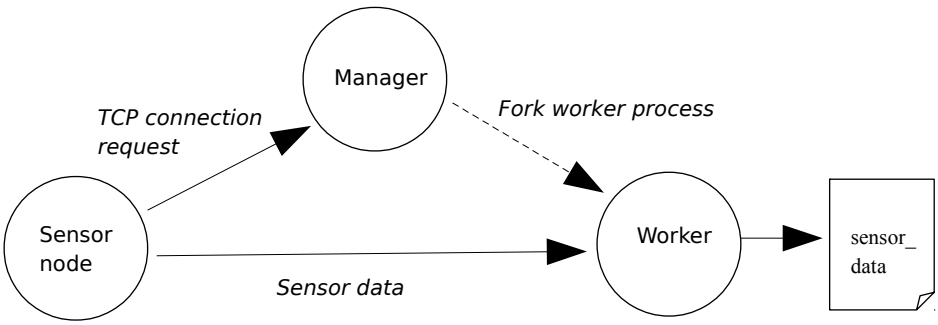


The connection manager may **not** assume a maximum amount of sensors at start up. In fact, the number of sensors connecting to the sensor gateway is not constant and changes over time. A sensor node opens a TCP connection to the connection manager and this TCP connection remains open during the entire lifetime of the sensor node. All data (packet format as defined in lab 6) is sent over this TCP connection. The code for a sensor node is given, together with the code of a simple server. In the given server code, however, only one client connection is supported at the same time. A sensor node is started up with the sensor node ID, the sleep time (in seconds) between to measurements, the IP-address and port number of the connection manager as command line arguments, e.g.: ./sensor 101 60 127.0.0.1 1234 will start a new sensor node that measures every 60 seconds the temperature and sends the result tagged with sensor ID 101 and a timestamp to the connection manager listening at 127.0.0.1:1234.

The connection manager should be able to handle multiple TCP connections at the same time. There are several solutions to solve this problem and you need to implement two different ones.

Solution 1: A multi-process sensor gateway

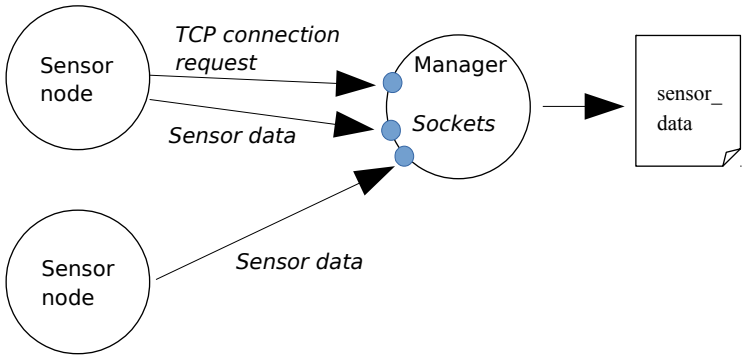
In this solution, the connection manager monitors every TCP connection in a separate process. The main process listens on one port for incoming connection requests from a new sensor node and creates a new worker process for each new connection request to handle the sensor data processing and database writing using the socket returned by the accept() call. The creation of the worker processes must be done dynamically, i.e. do not work with a fixed static pool of processes. The connection manager must be truly multi-processing: a new request must be handled even when some sensor data processing from another sensor node is on-going.



Keep in mind that the parent should 'wait' on the 'exit' of its children to avoid the creation of zombie-processes. This means that the parent needs to keep track of the PIDs of all children and needs to check from time to time (without blocking new incoming connection requests) if one of the children already exited. You can use a dynamic array or a pointer list data structure (see your list library from lab 5&6) to maintain all PIDs (and, if needed, other data). Use 'waitpid()' to do this and check out the 'WNOHANGUP' option of this command.
The connection manager is started up as a terminal application with a command-line argument to define the port number on which it has to listen for incoming connections, e.g.:
> ./a.out 1234
starts a connection manager listening on port 1234.

Solution 2: A single process sensor gateway

In this version only one process is used to handle multiple TCP sockets. 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 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 sensor connection manager is blocked on one socket while at the same time there is incoming data on another one. Multiplexing I/O using poll() or select() is a classical solution to solve this kind of problem. Use a dynamic array or a pointer list data structure to maintain all open sockets. Recall that a maximum amount of sensor nodes may **not** be assumed and that sensor nodes can open or close connections at any time. The connection manager keeps track of a 'last active' timestamp for every socket. If no activity was monitored on a socket after TIMEOUT time, the socket is closed and removed from the dynamic data structure. It should be possible to define TIMEOUT at compile-time with the preprocessor directive SET_TIMEOUT=<some_value> where <some_value> is expressed in seconds. Compilation should fail and an appropriate error message should be printed when this preprocessor directive is not set at compile-time.



SOLUTION 2 OF THIS EXERCISE NEEDS TO BE UPLOADED ON TOLEDO BEFORE THE DEADLINE.

Your solution is only accepted if the following criteria are satisfied:

1. *Compilation with '-Wall -Werror -std=c11' option generates no warnings or errors*
2. *Running 'cppcheck --enable=all' on the source code results in no warnings or errors*
3. *Run valgrind and check there are no memory leaks*
4. *The tests below should not fail.*

Before running the tests, do the following:

The implementation of the manager process is done in 'gateway.c'

Both, sensor node and manager will include the 'tcpsocket.h' (which you may not alter) in the same folder.

Test Case 1: run gateway and connect with one sensor node

check if data is being added to a sensor_data file

Test Case 2: connect with a second sensor node

check if data for both nodes is added to the sensor_data file

Verify that there is no maximum amount of sensor connections set in your code (connections are stored in a dynamic memory structure)

DO NOT upload code that doesn't satisfy these criteria.