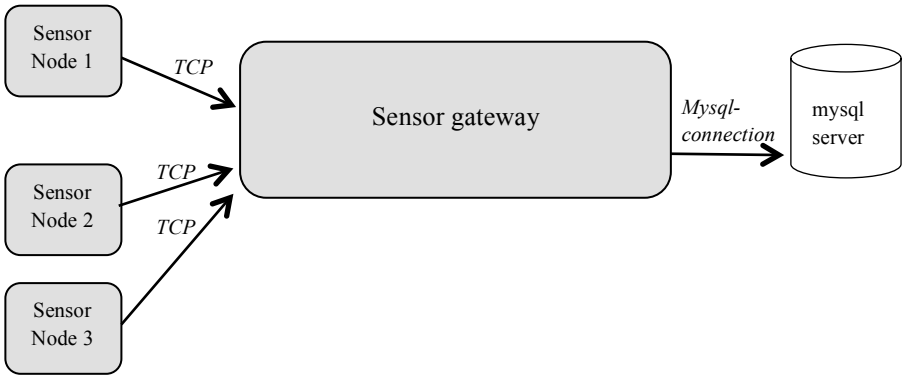
Lab 8 – Programming with Processes

Exercise: Processes and network communication

Implement a sensor gateway that listens for incoming sensor data from sensor nodes and writes all processed data to an SQL database. TCP sockets are used for the communication between the sensor server 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 available TCP client/server source code shows an example of TCP socket communication. First start the server, next start the client. The client will send a text message to the server. The server will 'echo' this message back to the client (this is called an echo server and is useful for testing purposes). You can start from this code and modify it. In the given code, the server, however, supports only one client connection at the same time.



The sensor 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.

Use the sensor node simulator from lab 6 as a starting point for the implementation of a sensor node, but modify the code a little bit such that:

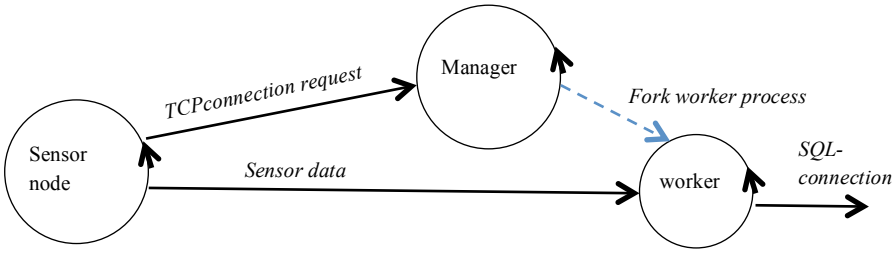
1. A sensor node opens a (real) TCP connection to the sensor gateway and this TCP connection remains open during the entire lifetime of the sensor node. All data (packet format is defined in lab 6) is sent over this TCP connection. A sensor node is started up with the IP-address and port number of the sensor gateway as command line arguments, e.g.: ./sensorNode 127.0.0.1 1234
2. The sensor node ID can be set at compile-time with the preprocessor directive SET_ID=<some_ID>.
3. The frequency at which the sensor node generates data should be set at compile-time with the preprocessor directive SET_FREQUENCY=<some_value> where <some_value> is expressed in seconds. A default frequency is used if this preprocessor directive is not set at compile-time.

Start from the sensor gateway implementation of lab 6 and modify it such that all received data packets are time-stamped, the parity is checked and next the data is written to the MySql database as implemented in exercise 3 of lab 7. Hence, you don't (need to) use the pointer-based data structure and the compression tool as defined in lab 6.

The sensor gateway 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 sensor gateway monitors every TCP connection in a separate process. The main or manager 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 set and static pool of processes. The sensor gateway 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 has to keep track of the PIDs of all children and needs to check from time to time (without blocking on-going file downloads and new incoming connection requests) if one of the children already exited. Use 'waitpid()' to do this and check out the 'WNOHANGUP' option of this command.

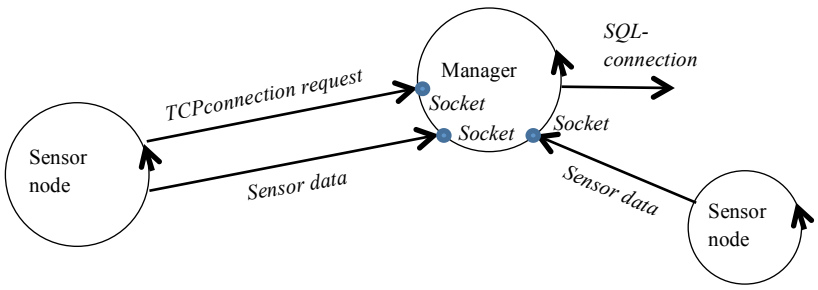
The sensor gateway is started up as a terminal application with a command-line argument to define the port number on which the sensor gateway has to listen for incoming connections, e.g.:

> ./sensorGateway 1234

starts a sensor gateway listening on the 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 gateway 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.

THE SOLUTION 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=c99' option generates no warnings or errors*
2. *Running 'cppcheck --enable=all' on the source code results in no warnings or errors*
3. *The tests below should not fail.*

Before running the tests, do the following:

The implementation of a sensor node is done in 'sensorNode.c'

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 the MySQL table

Test Case 2: connect with a second sensor node

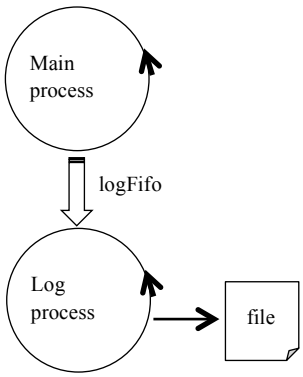
check if data for both nodes is added to the MySQL table

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.

Exercise: Fifo IPC

Implement a program that runs two processes: a main and a log process. The log process receives log-events from the main process using a FIFO called "logFifo". If this FIFO doesn't exists at startup of the main or log process, then it will be created by one of the processes. The main process generates log-events and writes these log-events to the FIFO.



A log-event contains an ASCII info message describing the type of event. A few examples of log-events are:

- A sensor node has opened a new connection
- The sensor node has closed the connection
- A data packet has failed the parity check
- Unable to connect to SQL server.

For this exercise, you can simulate the generation of these log-events using the sleep() and rand() functions, but it's clear that these log-events are actually coming from the sensor gateway.

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 on a log file called "gateway.log".