Lattice Watering

Christian Müller, Jonas Heinemann, Kaan Dönmez, Valentin Pickel

Report for the Software Project on Internet Communications, Summer Term 2022
Institute for Computer Science, Freie Universität Berlin, Submission Date: July 18, 2022

---

1 Introduction

We may start from scratch by reminiscing the the first meeting of all course participants on April 4th. This software project course should dive into the ecosystem internet. To do so, a rough split was set between the worlds of the *Internet of Things* (IoT), which deals with low power devices in large networks plagued with packet loss, and the conventional internet and its vastly more potent desktop computers and servers. Especially, us students should become aware of design decisions with respect to the communication solutions that our software should utilize. The project *Lattice Watering* eyes out the first of both worlds.

1.1 Idea Outline

After suggesting the building an IoT application for watering plants automatically on May 2nd, the software project *Lattice Watering* and its group formed on the 9th of May, comprising of the aforementioned members. The term *Lattice* is a reference to the studies of crystal/grid-like molecular structures in physics settings, which are often represented by two-dimensional structures called lattices. The first idea was not just to automate the watering of the plants, but also to let the boards drive around on a large grid. As it turns out, the hardware requirements for such a project would be too difficult to implement, so after some discussion we stuck to the original idea of a plant watering application, not changing the initial name.
In a nutshell, the idea is to be able to use a normal desktop computer to control several smaller microcontrollers, that are scattered widely. This may happen by setting automatic rules, or by the microcontrollers automatically turning on the water pumps as soon as humidity levels, which are measured by sensory, drop beyond a certain threshold. The latter aspect can be configured in a web frontend that is designed by us. Besides watering the plants, we also collect statistics like the humidity levels or network usage.

1.2 Team Organization and Applied Practices

For a detailled overview of the code hierarchy, one can look into the file `README.md` inside of the project directory.
git was used as our version control system, with a repository setup on *GitHub*. Our team members usually did not have fixed tasks, but utilize the agile method of *Kanban*. On GitHub, a kanban board was setup, in which members can add new issues and tasks to three categories: *Backlog*, *In Progress* and *Done*. This allowed our work to proceed in a assembly line-type way, in the sense that there are certain tasks that are saved up for later. However, in the nature of things, some members prefer to work closer to the IoT devices themselves, whilst others work e.g. with the web frontend. It is important to mention that we used *agile* methods, as we all had different experiences in the field of IoT communication we were studying, so agile methods and rapid prototyping seemed like good methods for this project.
Our methods of work also include *code conventions*, which are described in the `README.md` file. Such include using formatting tools and static analyzers, as well as code practices. Although they are quite few in this project, they reflect that we should take care of code quality at all times, and using matured automatic tools for that eases a normally adiministrative task.
Communication was done via the proprietary communication platform *Discord*, which was preferred over a decentralized solution like *Element/Matrix* due to all course members already using it.

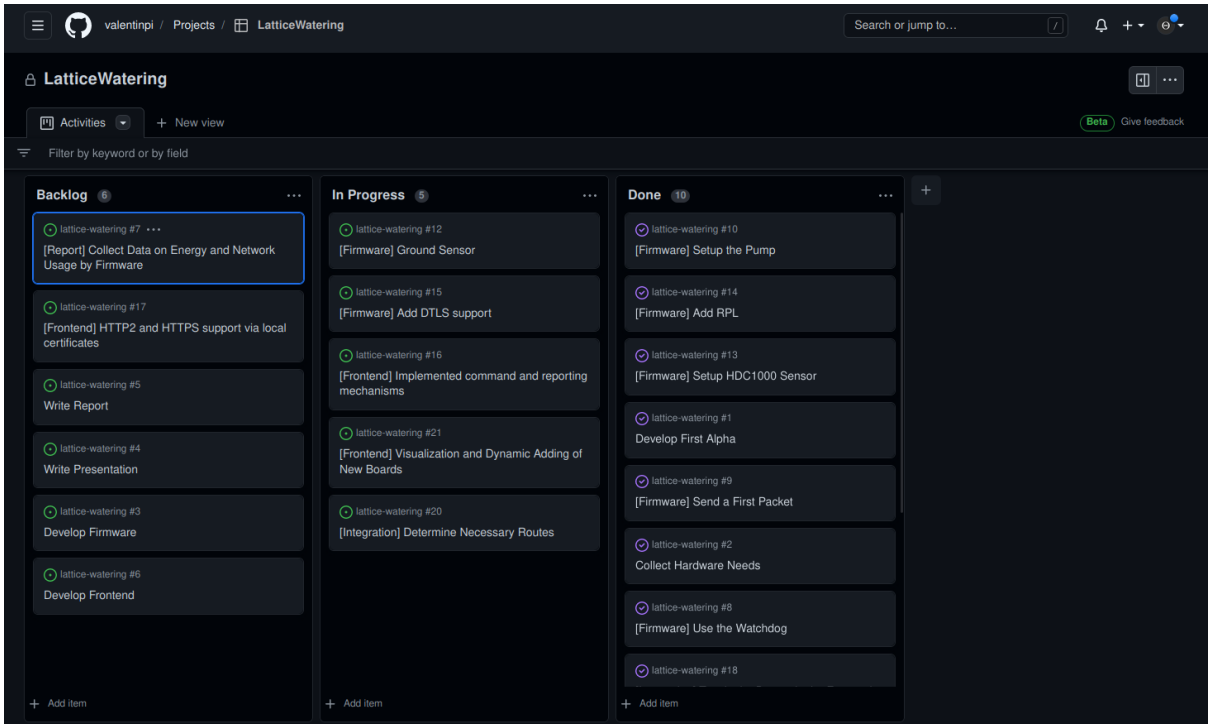

Figure 1: The kanban project board. (State: 13th June, 18:32)

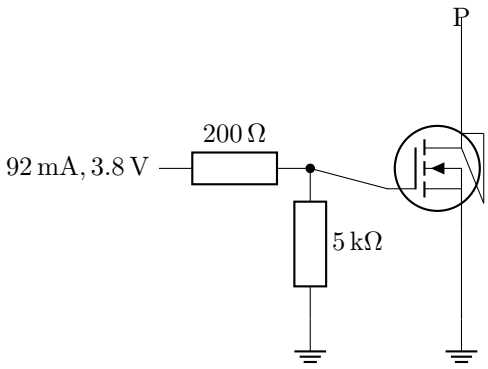

Figure 2: Circuit for the pump P.

2 Hardware

We shall not go into the details of the electronics of that circuit and why it would work under ideal conditions. We built the circuit, but as it turns out the transistor we had available failed.

3 Network Structure

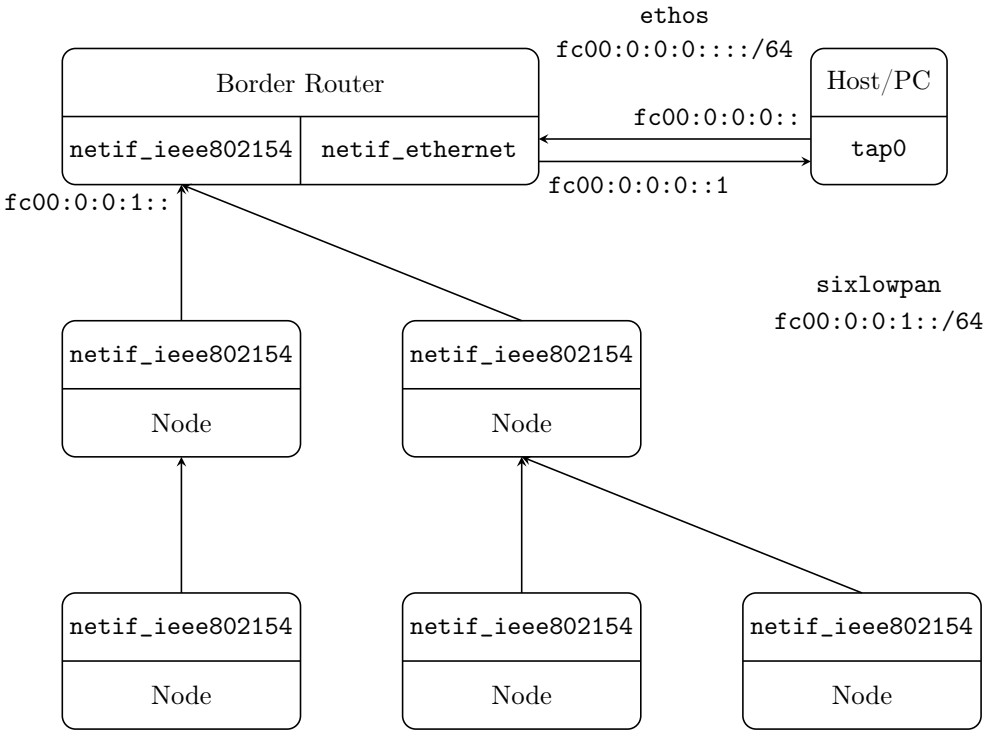

Figure 3: The Network Topology. The device interfaces are named after our code. The `tap0` device is named after the typical Linux entry. The network stack is based on COAP, DTLS, UDP, 6LoWPAN IPHC and FRAG, RPL, IPv6. The global IP addresses of the nodes inside the `fc00:0:0:0:0:0:0:1::/64` network are chosen by appending the Layer 2 addresses to the network prefix. The DAG structure of the RPL network is shown schematically.

4 Application Structure

In total, we have developed *five* different applications which all work together.

- `fw`: The firmware that is deployed on the nodes themselves reads out sensor values and sends them the host.
- `br`: The border router acts as a mediator between the lossy SixLoWPAN and the local Ethos network with the host. It allows the nodes that are reachable to send messages to the host computer.
- `proxy`: The proxy receives DTLS traffic from the nodes, decrypts them, appends a target IP and forwards them to the backend software written in Node.JS.
- `front`: The backend software serves the webpage for the frontend and manages the database.
- `www`: The website written in JavaScript, which may or may not be considered a seperate application, offers the user

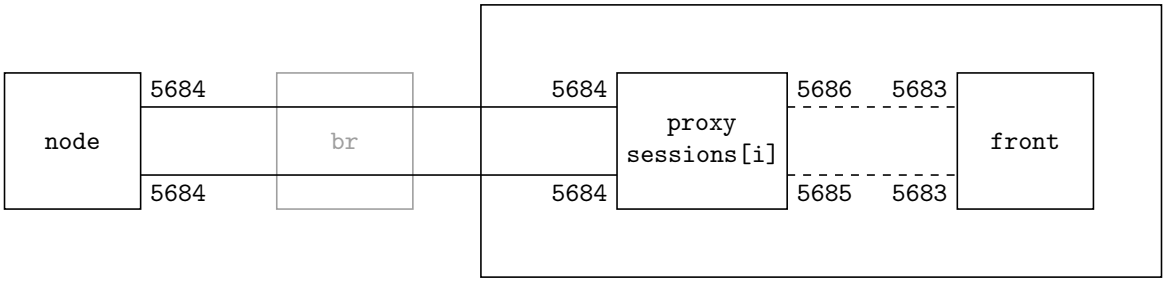

Figure 4: The DTLS proxy. Secure channels are represented by thick lines, insecure channels by thin lines. We have denoted the UDP ports. The `br` is greyed out as it only routes the packets. Note that each node has its own DTLS session registered in the proxy, the index `i` is for illustration. There can be at most 16 as of writing. Also the `proxy` has two sockets for communication with the backend, simply due to constraints with the C FFI in Rust which we were not able to fix by this time.

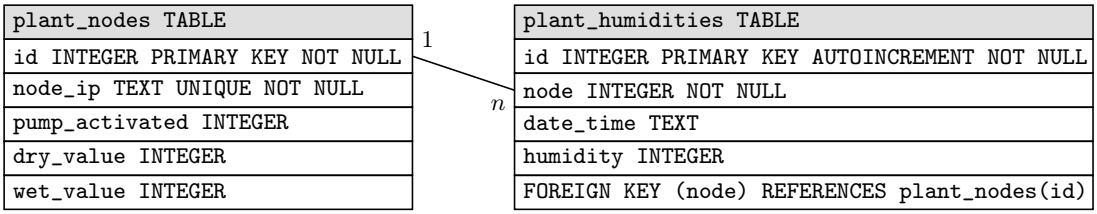

Figure 5: The SQLite database schema. The left table persists the current configurations for each node. The network structure guarantees the uniqueness of the IPs, so it suffices to use them as an identifier in our use case. One may consider replacing this with an application-specific ID which would have to persist on the nodes themselves, e.g. via EEPROM. One can find this exact structure in `front/db.js`.

To enable DTLS for our application, we had to first include the `gcoap_dtls` package in RIOT with a PRNG (Pseudo Random Number Generator) and an appropriate library, in this case `tinydtls`. However, Node.JS support for DTLS is in quite a rough state as this time, as none of the libraries we were trying out worked with. To be precise, we tried the following libraries:
- dtls -> Das hattest du inkludiert, aber darin ist gar nix drinnen. S. https://www.npmjs.com/package/dtls
- openssl-dtls https://www.npmjs.com/package/openssl-dtls - node-mbed-dtls https://www.npmjs.com/package/node-mbed-dtls - werift-dtls -> Eventuell funktioniert das jetzt, ich sollte das nochmal anschauen https://www.npmjs.com/packa
dtls - nodejs-dtls -> Enthält nur einen Client. https://www.npmjs.com/package/nodejs-dtls - @nodertc/dtls
-> Enthält nur einen Client. https://www.npmjs.com/package/@nodertc/dtls
- node-dtls-proxy https://www.npmjs.com/package/node-dtls-proxy - goldy https://github.com/ibm-security-innovation/goldy
There was also a short unsuccessful attempt at writing a very small proxy using python-dtls, but this was conceptually flawed, as the proxy needs to know where to route the packets from the frontend. In the end, as the board uses `tinydtls`, we decided to use Rust to write a small proxy using `tinydtls-sys`.

5 Closing Words

List of Figures