Software Project Report: Lattice Watering

VALENTIN PICKEL, JONAS HEINEMANN, CHRISTIAN MÜLLER, and KAAN DÖNMEZ, Freie Universität Berlin, Germany

1 Introduction

31st of July 2022 - We may start from scratch by reminiscing the first meeting of all course participants on April 4th. This software project course should dive into the ecosystem Internet. Before collecting project ideas, 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 devices ranging from desktop computers to large-scale servers. Especially, us students should become aware of the aspects that matter when making design decisions with respect to communication solutions. The project *Lattice Watering* eyes out the first of both worlds.

This report presents an outline of our project, meaning that we will talk about the software and hardware system we created, whilst also talking about our design decisions and pitfalls during development. In the end, we will give a short non-scientific evaluation. The hardware documentation and code of the project can be found in [14].

1.1 Idea Outline

After suggesting the building of 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 authors of this report. 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 plants, but also to let the boards drive around on a large grid, hence Lattice Watering. As it turned out, the hardware requirements for such a project would be too immense, so after some discussion we stuck to the original idea of an automatic plant watering application, whilst not changing the initial name. The latter decision was made in coordination with the team after some discussion. We thought that the name provides a neat little story for the project.

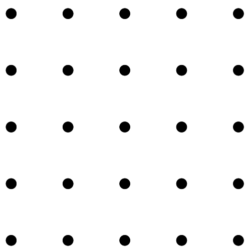


Fig. 1. A square lattice, corresponding to $\{0, 1, 2, 3, 4\}^2$.

In a nutshell, the idea is to be able to use a normal desktop computer to control several smaller microcontrollers that are scattered widely, that are connected to a humidity sensor and a pump. They can read out the humidity and toggle the pump. The watering may be controlled in two ways: We decided that a user should be able to setup automatic (timed) schedules or humdity thresholds, such that pumps would automatically turn on when the plant is too dry. Both functionalities should be configurable 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

git was used as our version control system, with a remote repository setup on *GitHub* with the popular free software license by the *MIT*. Our team members usually did not have fixed tasks, but utilized the agile method of *Kanban*. On GitHub, a Kanban board fig. 2 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 an assembly line-type of way, in the sense that possible future tasks are written down and saved up for when members are idling. 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. This means that tasks like reworking the frontend design are less accessible to backend developers, for instance. 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. We did not use a project plan.

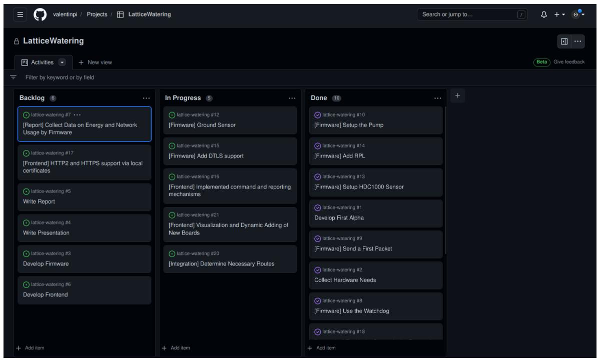
In the repository, we usually worked in several branches and merged branches together when changes were ready to be integrated in other branches. To make the switches between different parts of the codebase easier, we regularly wrote minimal documentations such that important parts of our codebase are always explained, whilst not investing too much time in the maintenance of small parts. We also tried to perform rationale management by explaining design decisions, which is why we have a small subsection for that inside of the `README.md` file. The team regularly met up on Monday to discuss current affairs. In the last two weeks before the presentation on the 18th of July, we also performed *integration tests*, meaning that we connected all systems together and overviewed the functionality of the entire system, especially with several boards, see section 3.

We consider it noteworthy that for uniformity in code, we set up some *coding conventions*, which are described in the `README.md` file. Such include not just whether to use the standardized integer types from `inttypes.h` in C, but also the use of code formatting tools and static analyzers. Although

Authors' address: Valentin Pickel, valentinpickel@gmx.de; Jonas Heinemann; Christian Müller; Kaan Dönmez, Institute for Computer Science, Freie Universität Berlin, Berlin, Germany.

they are few in this project, we hope that our use of them reflects that we tried take care of code quality at most times (except e.g. during prototyping), and using matured automatic tools for that eases normally adiministrative tasks. But we also missed out on a few things: Like developing tests and using continuous integration. But at this scale these tools did not seem necessary.

The tools we used further include `clang-format`, `just`, `make`, `npm`, Visual Studio Code and `gdb`. 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. We may improve on that in a future project. As mentioned we also used `gdb`, mainly for debugging the firmwares. For that, RIOT provided the convenient `make debug` command, which we used when there were errors that we did not understand. We also had issues with RIOT itself and its packages, as in [3] and [4]. We also had problems with other softwares, especially with frontend DTLS support, see section 4.2. Lastly, we want to note that we did not report everything we found, mainly due to semester stress and since we did not write them down, as that would require providing a proper documentation for reproducibility, which takes quite a lot of time besides the semester and deadline stress. The software also underwent many changes during development, which is no surprise due the design of our development process.



lattice-watering #7
[Report] Collect Data on Energy and Network Usage by Firmware

lattice-watering #17
[Frontend] HTTP2 and HTTPS support via local certificates

lattice-watering #5
Write Report

lattice-watering #4
Write Presentation

lattice-watering #3
Develop Firmware

lattice-watering #6
Develop Frontend

lattice-watering #12
[Firmware] Ground Sensor

lattice-watering #15
[Firmware] Add DTLS support

lattice-watering #16
[Frontend] Implemented command and reporting mechanisms

lattice-watering #21
[Frontend] Visualization and Dynamic Adding of New Boards

lattice-watering #20
[Integration] Determine Necessary Routes

lattice-watering #10
[Firmware] Setup the Pump

lattice-watering #14
[Firmware] Add RPL

lattice-watering #13
[Firmware] Setup HDC1000 Sensor

lattice-watering #1
Develop First Alpha

lattice-watering #9
[Firmware] Send a First Packet

lattice-watering #2
Collect Hardware Needs

lattice-watering #8
[Firmware] Use the Watchdog

lattice-watering #18

Fig. 2. The Kanban project board. (State: 13th June, 18:32)

2 Hardware

One major motivation for us was to work close to hardware. The boards and sensory we used came mostly from suggestions by Hauke Petersen [6], a scientific coworker in the department and lead developer of RIOT OS. In our network, see section 3, we essentially use three types of devices: A host computer, which is a powerful device in comparison to our small embedded devices that hosts the frontend, a border router that translates traffic of a *SixLoWPAN* network, that is the name for a set of protocols used in low power radio based communication networks, and so-called nodes. The latter two types of boards use class 2 embedded devices (see RFC 7228) with about 32kB of SRAM memory. A sharp eye may criticize that the border router is also quite a weak board, but during development we tried to mitigate this possible performance problem by reducing its role in the communication network to a pure routing one. In total, we have the following hardware list:

(1) One personal computer with the software set up.
(2) Two SAMR21-XPRO boards, one border router and one node.
(3) Two micro USB cables, optionally a battery connector or powerbank.
(4) One DRV8833 motor driver board.
(5) One electronic pump, see the file mentioned below. It should come with a long tube.
(6) One capacitive moisture sensor, see the file mentioned below.
(7) Nine female jumper cables.
(8) Five male jumper cables.

This entire list can be found in `HWSETUP.md` inside of the repository, with descriptions on how to connect the jumper cables. In the repository, one can also find instructions on how to setup the software. However, we assume that the reader is already familiar with RIOT OS and has the appropriate toolchains and `udev` rules set up.

As for the hardware itself, we want to share only one story. There was a very specific problem at the start of the project, which halted development shortly, but we managed to catch up as we were quite early when we obtained our hardware. The problem was that when connecting the pump to the node boards themselves, the ones of model `SAMR21-XPRO`, the pump induces a current of 200 mA. This is too much for the GPIO pins to handle, as one can read in the manual for the board, which we have provided in the `man` folder of the repository. Specifically, one can read in `man/Atmel-42223-SAM-R21_Datasheet-1065540.pdf`, p. 1055, that the pins can only handle about 92 mA, which means that connecting the pump directly will probably burn out the circuitry. Hauke made us aware of that, we shall cite from him that one should *always* also consider the power requirements of the hardware when building such a system, as one can otherwise quickly destroy something this way. One also notices that reading manuals was also part of this project, but mainly in the very early stages, as later on we got higher in the level of abstraction of our work, as we then mainly designed the frontend.

Together with Hauke, we discussed possible solutions to this problem, of which at first the most promising one can be seen in fig. 3. We can see that in this schematic the pump is connected to the board via the GPIO connection and both the 5 volt and ground pins. We shall not go into the details of the electronics of that circuit and why it would work under ideal conditions. According to Hauke, this is rather a matter of experience. What the may note is that the essential idea is that current flows between the `5V0` and `GND` pins to drive the pump if and only if as soon as an electric signal on the `PB03` pin is raised and thus the transistor is switched. We built the circuit, but as it turns out the transistor we had available failed. So Hauke proposed that we obtain some predesigned motor boards, which are documented in `HWSETUP.md`. Those

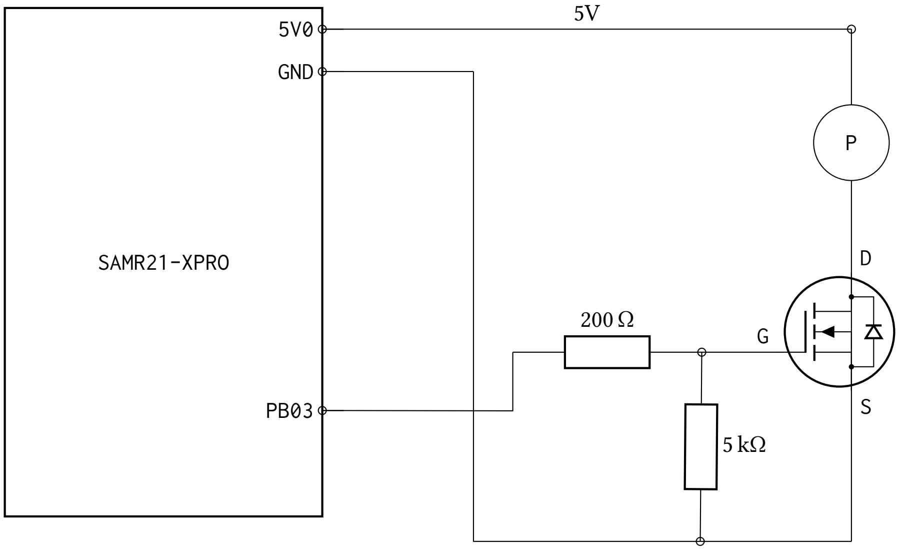


Fig. 3. Original circuit idea for the pump. This is a rough sketch, and does not exactly represent the circuitry involved. (P, S, G, D) = (Pump, Source, Gate, Drain). Note that the transistor on the right also has an embedded diode, which explains the arrow symbol on its right. We credit Hauke Petersen with the design of this circuit.

boards already have this circuit idea embedded in one board, meaning that there are some resistors and a transistor built onto one board and we just have to connecting everything properly.

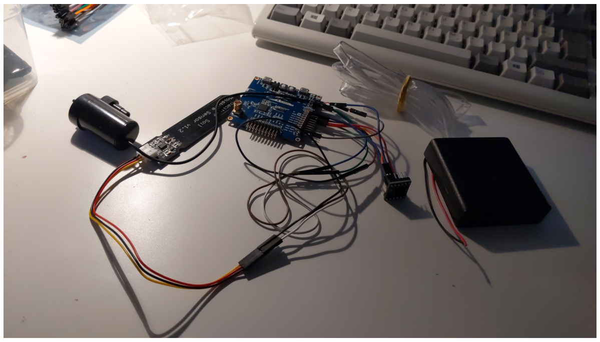


Fig. 4. An almost fully connected node. We can see that it is quite a cable clutter. In the final presentation we suggested to put everything in a casing, so that it looks more compact. On the right, before the battery case, we can see the driver board that connects the pump and the board, and on the left the capacitive moisture sensor that one would put into soil to measure humidity. It is not *fully* connected, because for one we would place the moisture sensor into soil, the pump into water, whilst connecting it with the plastic tube shown on the right, and then we would connect the board to a power source. This could be a battery connector, here the large case on the right that can hold up to four AA batteries, or a simple micro USB cable.

To get a feeling for what an almost fully connected node looks like, one may refer to fig. 4.

3 Network Structure

The structure of our network is illustrated in fig. 5. In section 2, we described that for our devices we had a host, a border router and nodes. We want to precise that a bit.

The network is divided into two parts. There is the connection between the border router and the host. It is a USB connection in our case, but it uses a technology called `ethos`, which is a small module/program that RIOT OS developed for setting up Ethernet networks based on serial connections. The user operating the host starts the `ethos` program on a serial port with the border router and that program sets up a layer 2 connection with the board by binding the port and transfering data inbetween. The physical layer connection is provided by the USB connection. At the same time, the USB port provides the border router with power, so we do not need to connect it to a battery or a plug seperately.

We have wrapped the `ethos` program inside of a small `Makefile` command, which one can inspect in `br/Makefile`. Besides starting the program, the command also creates a virtual network tap device, which is only natively possible on Linux at this moment, as neither Windows nor Mac support that functionality. As for other OSs, FreeBSD supports such devices [1]. The virtual network device is bounded by `ethos` and gets assigned the IP `fc00::`, which is an IPv6 *Unique Local Address* (ULA). With that, we follow the recommended conventions for unique local addresses described in RFC 4193. The border router assigns itself the fixed address `fc00::1`, such that we may easily debug network issues. So the network prefix `fc00::/64` at this moment is made up of two devices.

We introduce the second network. The nodes are designed to be low power devices that can be scattered around a small area

[1] No reference here as one can look this up quickly.

such as a small house. They connect themselves to the fixed IP address `fc00:0:0:1::`, which is the address that we assign the IEEE802154 network interface of our border router. Note that it has such a radio interface for receiving IP packets from the nodes and forwarding them to the frontend. The addresses of the nodes are made up of the prefix `fc00:0:0:1::/64` and the last 64 bits replaced with the MAC address of the node. So these IP addresses are predetermined. The professor already suggested using an application specific specifier, so this could be an improvement over this rather rough way of designing the layer 3 addressing.

So we have now described how the devices are setup. But we want to shine a bit more light on the other protocols that are involved in the network. For layer 3, we only use *IPv6* as it is the more modern standard and since RIOT does not support IPv4. We also learned that IPv6 replaces some old technologies with newer ones. For instance, ARP is replaced with IPv6s very own Neighbor Discovery routines.

To perform routing inside of the lossy IoT network, we use the `Routing Protocol for Low-Power and Lossy Networks` (RPL). RFC 6550 describes it. The short idea is that we may create routing topologies for far areas by designating some network entities as roots such that the nodes can build up graph theoretic tree structures, which are just *Directed Acylic Graphs* (DAGs). The routing now happens in the way that a node has exactly one entity it can route a packet to. Note that the RFC also suggests some useful topologies for these routing schemas, such as a *Destination-Oriented DAG* (DODAG), where there is only one root. This is useful in our case, as it simplifies the routing by quite a lot. It is clear that the border router serves as an appropriate root for us.

These three technologies presented two extremes for us during development. For one, setting up `ethos` and the border router routing table, to route between the IEEE802154 and Ethernet interfaces, yielded unneeded error searches while we were trying to apply the knowledge from the tutorials that were given on the RIOT OS page. The most usual error was that packets just were not routed between the nodes and the host. We checked that by using the ICMP modules from RIOT from shell and sending pings between our fixed IP addresses, see above. Using a snooping tool such as *Wireshark* on the host, we also checked the traffic in hopes of finding out our issues. We cannot replicate all that went wrong during this phase. The professor suggested that we should make proposals on how to improve the RIOT tutorials, but due to stress we have not yet done so[2]. Perhaps the underlying issue also was that the team was just not familiar enough with RIOT and the protocols involved. That is the first extreme: Huge difficulties whilst setting up `ethos` and IPv6 routing. The second extreme was the very pleasant experience in setting up RPL, which just consisted of designating the border router as a DAG root and then just loading the appropriate modules into both `br` and `fw`. The former can be done just using:

```
gnrc_rpl_init(netif_ieee802154->pid);
gnrc_rpl_root_init(0, &ieee802154_ip, true, true);
```

The combination of *ethos* and the more-or-less hardcoded network structure was on purpose as it is the most convenient and least complicated way of setting it up. The only thing to keep in mind was to update the firewall rules on our Linux devices, for instance via `nft`:

```
# nft --handle list ruleset
# sudo nft add rule inet filter input \
           position 9 udp dport 5684 accept
```

As for the other protocols used, our communication principle was to design it as energy efficient and with as small and as few regularly sent packets as possible. Thus we ruled out any form of state based communication besides DTLS as we cannot compromise on security: For one, an attacker might be interested in damaging expensive plants to hurt the owner financially and emotionally. Our threat model assumes that no node is compromised, which is why we used Pre-Shared Keys, see section 4.2 for more details. We do not claim that this assumption always holds, especially since we have not secured our own hardware by e.g. soldering the USB connections. An attacker could read out the board memory directly by using `openocd`, which e.g. the RIOT OS Makefile system uses for flashing the boards.

Back to the topic of networking. In the standard ISO/OSI model we skip layers five and six, besides what protocol management introduces in form of certain saved cookies by the DTLS protocol, and turn to layers four and seven. The former is not very interesting: We use UDP-based application protocols. For all communication between the nodes and the host, we use non-confirmable CoAP packets that are DTLS encrypted. The exact methods between that can be called using CoAP on the node or on the host are described in the file `METHODS.md`. Note that since each node uses DTLS with the host, meaning that they both share a secure session, these RPC calls are all secured and unique to one communication. More precisely, in RIOT we use the abstraction GCoAP or GCoAP-DTLS, which allows us to define method listeners directly and send out packages quite easily, whilst possibly introducing a slight bit more of packet duplication, as we do not directly use RIOT OS Packet Buffers. Instead of CoAP, there is also MQTT-SN, a different application-layer protocol. We did not choose it, as for one RIOT does not provide such a good integration with DTLS as with GCoAP, which essentially only requires the developer to register the needed keys, but also one can read in articles that packet loss may be quite huge with MQTT-SN when near the broker, as [15] has studied. [16] also gives evidence of CoAP being able to handle traffic floods much better. MQTT-SN might perform slightly better while saving a few bytes, but the difference is not huge. Also, we are much more familiar

[2] And we are not quite sure how to, as our solution may not work for everybody.

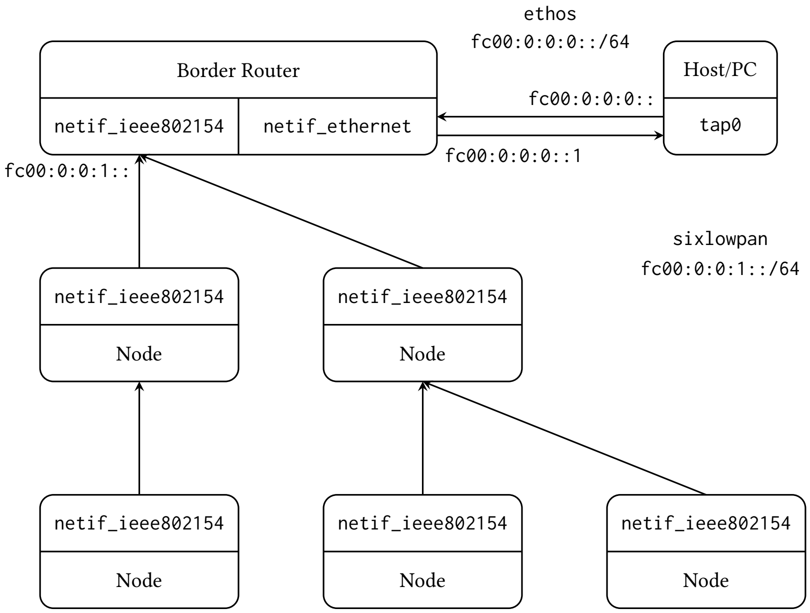


Fig. 5. The Network Topology. The device interfaces are named after our code. The `tap0` device is named after the typical Linux entry. The network stack is based on COAP, DTLS, UDP, 6LoWPAN IPHC and FRAG, RPL, IPv6. The global IP addresses of the nodes inside the `fc00:0:0:1::/64` network are chosen by appending the layer 2 addresses to the network prefix. The DAG structure of the RPL network is shown schematically. This also means that the communication between the border router and the closest nodes in the DODAG is $1 : n$.

with the HTTP-like structure of CoAP, so working with it was quite pleasant.

We note that all of the traffic between the host and the nodes is also encoded in the CBOR format, which is a binary, more compact alternative to JSON. We use the `nanocbor` package on our nodes.

The frontend is made up of a Node.JS backend and a small webpage for controlling individual boards. More on that in the next section. We only note that the communication between the Node.JS server and the webpage is based on HTTP and TCP, which is quite usual. Sadly, we still use HTTP 1.1 there, as the Node framework `express` does not support newer versions, but at least the traffic is `gzip`-compressed. A future improvement could be the use of local certificates for the development server, just such that on the developer machine this communication is also secured. It is not necessary.

4 Application Structure

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

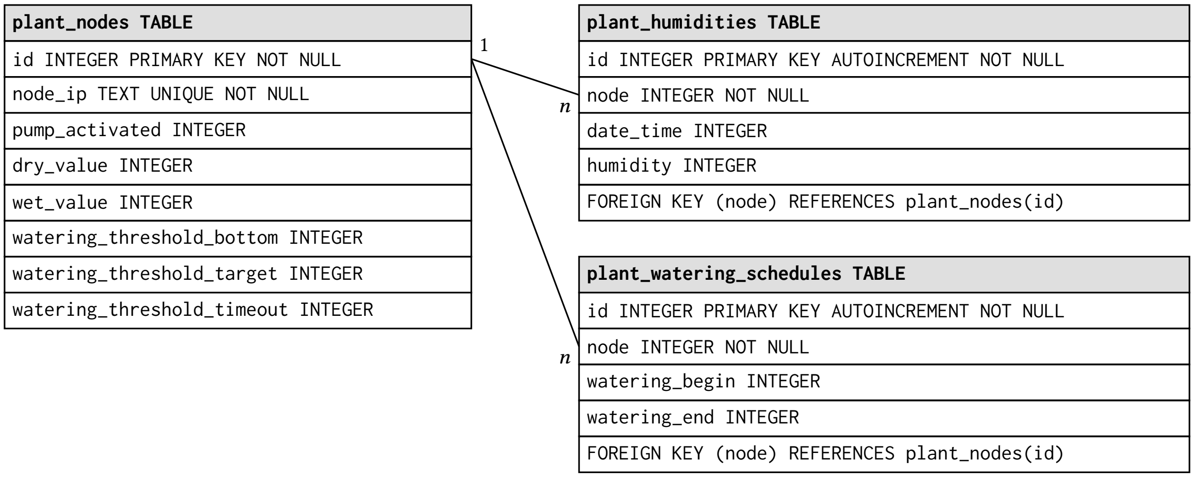
- `fw`: The firmware that is deployed on the nodes themselves reads out sensor values and sends them the host.
- `br`: The border router acts as a mediator between the lossy SixLoWPAN and the local `ethos` network with the host. It allows the nodes that are reachable to send messages to the host computer.
- `proxy`: The proxy receives DTLS-encrypted CoAP traffic from the nodes, decrypts them, appends a target IP and forwards them to the backend software written in Node.JS. Analogously, it takes non-encrypted CoAP packets from the frontend, removes an appended IP and forwards them to the appropriate node.
- `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 a list of all connected nodes, an overview of their humidity values and options for calibration and automatic plant watering.

The frontend also stores the humidity data and the current status of nodes according to the data regularly transmitted on the `/data` route, usually every five seconds. Figure 6 shows the database schema. We use SQLite, a very small and fast single user database, which suffices for our usecase. For something larger we may need multiple users and thus a different database, which should not be too difficult given the complexity of our current database operations, which one can inspect in `front/db.js`.

4.1 Embedded Work: `br` and `fw`

Both firmwares are kept very simple. Both are based on RIOT OS with its default GNRC protocol stack. We considered using other stacks such as OpenWSN and OpenThread, but GNRC was preferred due to its simplicity and its prevalence in RIOTs examples.

The `br` firmware sets up a small border router as a RPL root, with a route between the `sixlowpan` and `ethos` networks and with a watchdog. It also responds to ICMP packets for pings.



plant_nodes TABLE
id INTEGER PRIMARY KEY NOT NULL
node_ip TEXT UNIQUE NOT NULL
pump_activated INTEGER
dry_value INTEGER
wet_value INTEGER
watering_threshold_bottom INTEGER
watering_threshold_target INTEGER
watering_threshold_timeout INTEGER

plant_humidities TABLE
id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL
node INTEGER NOT NULL
date_time INTEGER
humidity INTEGER
FOREIGN KEY (node) REFERENCES plant_nodes(id)

plant_watering_schedules TABLE
id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL
node INTEGER NOT NULL
watering_begin INTEGER
watering_end INTEGER
FOREIGN KEY (node) REFERENCES plant_nodes(id)

Fig. 6. The SQLite database schema. The upper left table persists the current configuration and status for each node. The network structure guarantees the uniqueness of the IPv6 addresses, so it suffices to use them as an identifier in our use case. One may consider replacing this with an application-specific ID as we have described above, which may for one have to persist on the nodes themselves, e.g. via EEPROM. One can find this exact structure in `front/db.js`. The other two tables store the humidity values of the plants over time, such that we can access them later on, and the current fixed-time watering schedules. For simplicity, the latter is given in seconds after midnight for each day. Remember that we offer threshold-based and schedule-based watering. We have two 1 : *n* relationships.

The `fw` firmware is slightly more complicated in that it also sets up some sensory, GCoAP routes and DTLS encryption. For the evaluation of the firmwares we refer to the very last section.

4.2 `proxy` and DTLS

The `Datagram Transport Layer Security` protocol aims to introduce security of communication to the unsecured UDP protocol. DTLS (RFC 4347) behaves similiar to TLS in that before a communication may take place, a participant first has to negotiate secure ciphers for communication, but not in that the packets are assured to arrive in the correct order or fully. However, the protocol guarantees the correct order for the received payload data.

To enable DTLS for our application, we had to first include the `gcoap_dtls` package in our `fw` in RIOT with a *Pseudo Random Number Generator* (PRNG) and an appropriate library, in our case `tinydtls`. We did not use WolfSSL as it is currently not supported by the `gcoap` RIOT abstraction for CoAP. We used the SHA512-based PRNG, assuming better security, although we introduced a higher computational cost with that.

Sadly introducing DTLS to our communication turned out to give us some of the worst trouble we had in the project, because Node.JS support for DTLS is in quite a rough state as this time, as none of the libraries we were trying out worked. Note that there is an official issue for introducing DTLS to Node [13], but the developers there are not very active at the moment. And it is not clear if this will be resolved anytime soon, especially since the new protocol QUIC emerged. To be precise, we tried the following libraries[3]:

- `dtls` [1]: This NPM package just does not contain any code.
- `openssl-dtls` [2]: Node FFI bindings that work, but during development, OpenSSL did not work with the ciphersuite `TLS_PSK_WITH_AES_128_CCM_8`, which is used by `tinydtls`. The cipher negotiation fails for to us unknown reasons. We have opened up one issue in that repository, see [5].
- `node-mbed-dtls` [7]: Based on `mbed-dtls`, but with the same issue as with `openssl-dtls`.
- `werift-dtls` [8]: Crashes upon receival of a Client Hello packet, also very poorly documented code. As it seems, it is part of a bigger set of tools for WebRTC.
- `nodejs-dtls` [9]: Only contains a DTLS client.
- `@nodertc/dtls` [10]: Only contains a DTLS client. Development for a server seems to be ongoing on the `main` branch on GitHub, but with no recent activity
- `node-dtls-proxy` [11]: Crashes upon receival of a Client Hello packet.
- `goldy` [12]: Cipher negotiation with a node fails, also it seems that the proxy only forwards traffic between two peers, but we want to be dynamic in that multiple nodes can connect to the server and we do not need to start a proxy for each one seperately.

There was also a short unsuccessful attempt at writing a very small proxy using `python-dtls`, but this was for one at first conceptually flawed, as the proxy needs to know where to route the packets from the frontend. Also, receiving Client Hello packets did not work. In the end, as the board uses `tinydtls`, we decided to use Rust to write a small proxy using `tinydtls-sys`. Figure 7 shows the structure of the proxy, which acts in the middle of the communication between the

[3] We found the first seven on www.npmjs.com/package.

frontend and the nodes. Note that the proxy is running on the host computer. We chose Rust as the developer that assigned himself this task wanted to get to practice it more, and it seemed to be a suitable language due to Rust safety guarantees. Although one may argue that Rust is more difficult to learn, but the proxy complexity is not that high. Because Rust bindings exist for RIOT, we could also have developed the firmwares in Rust, but not every team member is familiar with the language and we did not have that much time.

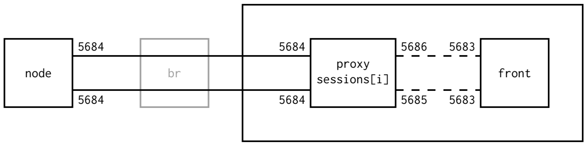


Fig. 7. The DTLS proxy. Secure channels are represented by thick lines, insecure ones by dashed lines. We have denoted the UDP ports. The `br` is greyed out as it only routes the packets. Note that each node has its own DTLS session registered in the proxy, the index `i` is for illustration. There can be at most 16 as of writing, but that can be easily changed with a Rust constant. 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.

When a board wants to send data through the `/data` CoAP POST method of the host, it first tries to establish a secure DTLS session. For that, it sends a Client Hello packet and then does cipher negotation such that the future communication partners know how to encrypt and decrypt packages. One special property of our communication is that we use *Pre-Shared Keys* (PSKs). Via an *identity message*, the server tells the node which pre-shared key to use. We only have one, so we called our identity `default`. The PSKs themselves are just random data, here 16 bytes, as RIOT does not support longer keys. We automatically generate them using a script, and recommend frequent regeneration via the instructions provided inside of `README.md`.

4.3 For the Browser: `front` and `www`

A small backend using `Node.JS` and a website, see fig. 8, were developed. The backend operates a database with `SQLite` to store all data received and the corresponding IPv6-Addresses. The HTTP (not CoAP!) methods the backend offers are only documented in code, but not in a seperate file, as these are internal functionalities. It will register newly connected IPv6-Addresses, if not already present, as nodes in the `plant_nodes` table and store their current humidity and current time in the `plant_humidities` table. The frontend will show all registered nodes in an overview, whilst also depicting their current humidity and IPv6-Addresses. This could be changed to show only nodes registered in the last `X`-minutes to portray only active and currently used plant nodes. All database accesses are done by the backend and only the backend which will send the information to the frontend if requested. We used `async` functions whenever a database request was issued since JavaScript would not wait for the query to finish and use a `null` as the output. `async` functions allowed us to make promises to only continue working after a query has been finished.

The backend uses packages such as `coap` and `express` to establish the communication to the node and to the website served. `chartjs` is used to generate a static image which is served to the frontend. We designed it this way for lowering website load. The frontend to backend communication is done via HTML GET and POST requests. After every function call we send the status 204 back to the frontend, which stops the browser from loading a different or current site.

The websites' HTML and CSS were written manually. We use small HTML templates that are inserted using DOM operations for dynamic behavior and JavaScript to issue database requests and the current time.

5 Evaluation and Closing Words

We shall give some thoughts about the final result of the project and some highly non-scientific evaluation results.

5.1 Discussion on Production-Readiness and Marketability

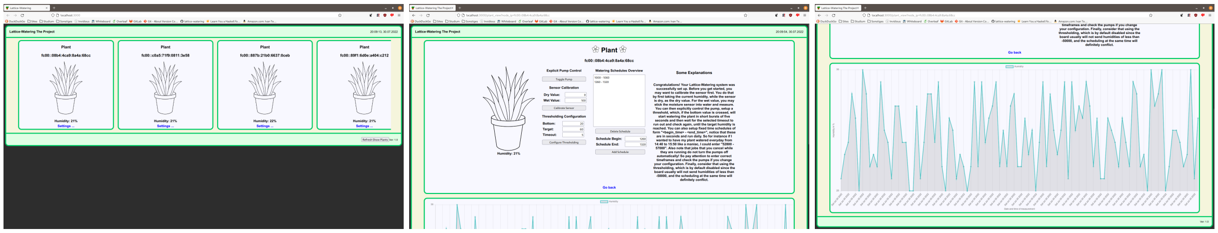
As explained in the section on hardware, the node devices that are arguably the most important parts of the system as they themselves water the plants and take measurements are quite unorganized. For a product look, one needs to consider putting each node in a case and fastening the connections, possibly by soldering more of the cables. Especially during the final weeks we had boards failing to connect the pump due to loose contacts over the jumper cables.

Regarding production-readiness, the product needs much more testing, especially user tests and automated tests. Whilst we claim to have developed some measures for implementation robustness, we do not claim high correctness. An Over-The-Air update system has also not been developed, although as the professor has pointed out, this is quite an open problem in IoT.

5.1.1 Software Robustness

Our implementation has a few properties that make it a bit more robust, than a first look suggests, in terms of security and scalability:

- Only a certain amount of devices can open up a session with the DTLS proxy. A malicious client is simply blocked if too many nodes are registered.
- We have taken care of minimizing packet size as much as possible. The average packet in the `sixlowpan` network is about 116 bytes large, see section 5.1.2.
- A watchdog is running on each node, restarting the node when it does not get updated after a while. This even helped us during debugging, when strange bugs lead to the board freezing. The main reason for deploying a watchdog is simply the scenario when a plant is being watered and the freezes freezes due to a yet unknown implementation error. It forcefully gets restarted then and turns off the pump.
- When the frontend loads data for visualization, it does not load *all* data from `plant_humidities`, but only a



Dry Value: 0
Wet Value: 100

Bottom: 20
Target: 60
Timeout: 5

1000 - 1060
1260 - 1320

Schedule Begin: 1260
Schedule End: 1320

Congratulations! Your Lattice-Watering system was successfully set up. Before you get started, you may want to calibrate the sensor first. You do that by first taking the current humidity, while the sensor is dry, as the dry value. For the wet value, you may stick the moisture sensor into water and measure. You can then explicitly control the pump, setup a threshold, which, if the bottom value is crossed, will start watering the plant in short bursts of five seconds and then wait for the selected timeout to run out and check again, until the target humidity is reached. You can also setup fixed time schedules of form "<begin_time> - <end_time>", notice that these are in seconds and run daily. So for instance if I wanted to have my plant watered everyday from 14:40 to 15:50 like a maniac, I could enter "52800 - 57000". Also note that jobs that you cancel while they are running do not turn the pumps off automatically! So pay attention to enter correct timeframes and check the pumps if you change your configuration. Finally, consider that using the thresholding, which is by default disabled since the board usually will not send humidities of less than -50000, and the scheduling at the same time will definitely conflict.

timeframes and check the pumps if you change your configuration. Finally, consider that using the thresholding, which is by default disabled since the board usually will not send humidities of less than -50000, and the scheduling at the same time will definitely conflict.

Fig. 8. The frontend in three pictures: We have the overview on the left. New boards are registered when they send their first `/data` packet. The IP and the most recent humidity value is displayed. In the middle, we see the configuration overview. We can toggle the pump manually, calibrate the sensor and either configure a threshold or schedules for watering. Two schedules are configured here. The rightmost picture shows the humidity graph for the last 50 entries. Note that this screenshot uses fake entries for demonstration, we have tested the functionality manually with real boards.

certain amount. This could be improved even further by only loading the very last datum upon startup.

5.1.2 Network Statistics

Please note that we do not claim full correctness or reproducibility for the following statistics. They are non-scientific and should just reflect our experience with the devices.
Leaving a system of one node, a border router and a host running for about an hour, calibrating the sensory once and toggling the pump twice, we collected the following data: The node transferred 92537 bytes in 795 unicast packets and 71 packets with no failures, whilst receiving 14886 bytes in 192 packets.
Surely more devices could lead to much more packets being exchanged, for instance due to IPv6 neighbor advertisements or discovery and due to RPL routing activities. We also had no transfer failure, which is highly unlikely when the distances get larger and the network thus becomes lossier.

5.1.3 Energy Statistics

Please note our starting words in section 5.1.2.
Letting a single node run on a battery connector with four AA batteries yields a lifetime of about five days. There are several factors that influence this result, such as, again, more devices in the network and their induced routing activity; but also the interval of sending data might be too high, the fact deep sleep functionalities were not used, the PRNG used might be too much of a computational load, the watering frequency as the pump consumes a lot of power and others. Strategies for improving the battery life also consist of rechargable batteries or even solar panels as the plants need light anyways.

5.2 Reflection on our Deployed Practices

Development was not always easy, as during early phases we often got stuck with the hardware and RIOT. One remarkable difficulty was in setting up the humidity sensor. As we did not find a good way of interpreting the digital values coming from the analog connection with the sensor and it needing to apparently connect to the wrong pin, - See `HWSETUP.md`: Its VCC port must be connected to ID for some reason. - we even had to transfer this task over to the user, which is quite unsatisfying. Communication problems regarding confusion on who assigned oneself which tasks and merges that break certain branches of our repository did not make our lives easier. During the last two weeks of development quite a lot was not ready in the frontend, like the scheduling functionality, but we successfully caught up on that in some very chaotic and fast sprints. This is not desirable.

5.3 Possible Future Enhancements

We list a few possible enhancements in the code.

- Port `br` and `fw` over to Rust.
- Either one figures out what the problem with the strange fluctuations of the capacitive sensor are, or one switches it out for a more reliable one.
- Let the proxy use a more secure RNG for `tinydtls` than the default implementation.
- The frontend should be ported to TypeScript.
- The website looks a bit outdated, one should revisit its design.

There are more improvements and issues, one can look into the repository for that.

5.4 Conclusion

We explored a small part of the world of RIOT OS when discovering its network stack and also learned about the difficulty of making several softwares and their developments working together. The final product, by our very non-scientific tests, seems to possess some desirable qualities such as being secure, fast, bandwidth-saving and energy-efficient. As for the functionality of the system itself, we may remind us that sadly the thresholding mechanism for watering still does not work fluently, but the scheduling mechanism does. It has been tested several times. Coupling that with properties such as the automatic registration of new devices and the collection of statistics whilst possessing very slowly increasing storage requirements, we may note that the system is working quite well and provides one possible way of watering ones plants.

References

[1] `dtls`. NPM Package: https://www.npmjs.com/package/dtls [last access 25th July 18:37].
[2] `openssl-dtls`. NPM Package: https://www.npmjs.com/package/openssl-dtls [last access 25th July 18:45].

[3] `[gcoap-dtls]` Posting a message yields a stack overflow on the `samr21-xpro` with ECC. GitHub Issue: https://github.com/RIOT-OS/RIOT/issues/18292 [last access 30th July 18:23].

[4] samr21-xpro: Strange `printf` issue. GitHub Issue: https://github.com/RIOT-OS/RIOT/issues/18262 [last access 30th July 18:25].

[5] No start line but the certificate is correctly formatted. GitHub Issue: https://github.com/jue89/node-openssl-dtls/issues/4 [last access 30th July 18:29].

[6] Hauke Petersen. https://www.mi.fu-berlin.de/en/inf/groups/ilab/members/petersen.html [last access 30th July 18:31].

[7] `node-mbed-dtls`. NPM Package: https://www.npmjs.com/package/node-mbed-dtls [last access 25th July 19:32].

[8] `werift-dtls`. NPM Package: https://www.npmjs.com/package/werift-dtls [last access 25th July 19:33].

[9] `nodejs-dtls`. NPM Package: https://www.npmjs.com/package/nodejs-dtls [last access 25th July 19:35].

[10] `@nodertc/dtls`. NPM Package: https://www.npmjs.com/package/@nodertc/dtls [last access 25th July 19:41].

[11] `node-dtls-proxy`. NPM Package: https://www.npmjs.com/package/node-dtls-proxy [last access 25th July 19:43].

[12] `goldy`. GitHub Repository: https://github.com/ibm-security-innovation/goldy [last access 25th July 19:44].

[13] Node DTLS Discussion. GitHub Issue: https://github.com/nodejs/node/issues/2398 [last access 25th July 20:27].

[14] `lattice-watering`. GitHub Repository: https://github.com/valentinpi/lattice-watering [last access 30th July 18:05].

[15] GÜNDOGAN, C., KIETZMANN, P., LENDERS, M. S., PETERSEN, H., FREY, M., SCHMIDT, T. C., SHZU-JURASCHEK, F., AND WÄHLISCH, M. The impact of networking protocols on massive m2m communication in the industrial iot. *IEEE 18* (2021), 4814–4828.

[16] HERRERO, R. MQTT-SN, CoAP, and RTP in wireless IoT real-time communications. *Springer 26*, 6 (2020), 643–654.