Terra:
Enabling Edge-Based Stream Processing for Resource-Constrained Environments

Kasper Hjort Berthelsen

Advisor: Philippe Bonnet
Submitted: December 2024

IT UNIVERSITY OF COPENHAGEN

Abstract

The world is becoming more and more data driven and with the rise of the Internet of Things, more and more data is being generated and transferred every day. The traditional way of data processing by collecting data by default in cloud data centres is becoming unviable due to increased network congestion and demands for privacy-aware processing and real-time responses. The concept of fog and edge computing was introduced to solve these challenges, but these often still require transfer of data from sensor nodes to processing nodes, and the option of processing data on the sensor device itself is still absent.

This work introduces TERRA, which enables query processing on resource-constrained sensor nodes at the very edge of modern sensor network databases. Integrated into a state-of-the-art network database, this approach significantly minimises network traffic, leading to a big reduction in energy consumption. Through a series of real-world experiments on physical devices, we evaluate TERRA's performance and experimentally derive an energy cost model that verifies TERRA's capabilities. In particular, we show a significant reduction in network transfer and therefore energy cost when pushing aggregate computation onto sensor devices.

Resumé

Verden bliver mere og mere datadrevet, og med Internet of Things popularitet bliver mere og mere data genereret og overført hver dag. Den traditionelle måde at behandle data på ved at indsamle data som standard i cloud-datacentre er ved at blive ulevedygtig på grund af øget netværksbelastning og krav om privatlivsbevidst og realtid databehandling. Koncepterne fog og edge computing blev introduceret for at løse disse udfordringer, men disse kræver ofte stadig overførsel af data fra sensorenheder til databehandlingsenheder, og muligheden for at behandle data på selve sensorenheden er stadig fraværende.

Denne afhandling introducerer TERRA, som muliggør processering af forespørgsler på ressourcebegrænsede sensorenheder på den yderste edge af moderne sensornetværksdatabaser. Integreret i en state-of-the-art netværksdatabase minimerer denne tilgang markant netværkstrafikken, hvilket fører til en stor reduktion i energiforbruget. Gennem en række fysiske eksperimenter på sensorenheder evaluerer vi

TERRAs ydeevne og udleder eksperimentelt en energiomkostningsmodel, der verificerer TERRAs evner. Især viser vi en betydelig reduktion i netværksoverførsel og dermed energiomkostninger, når vi skubber aggregatberegning ud på sensorenheder.

Acknowledgements

On paper a Ph.D is independent research carried out solely by me, the singular author. In reality, a great number of people contributed to the completion of this thesis.

First, I want to thank my supervisor **Philippe Bonnet** for your advice, support, and guidance. And I want to thank my two bachelor students **Laurits** and **Markus** for dedicating their time and project to helping me.

I want to thank my colleagues and friends **Morten**, **Ties**, **Dovile** and **Adrian** for your friendship and support, both at work and away. This thanks is extended to all my colleagues in the **DASYA** group.

I want to thank everyone at **DIMA** at **TU Berlin** for hosting me on my stay abroad and allowing me to work with NebulaStream. I especially want to thank **Viktor Rosenfeld** there for our regular talks and for continuing to follow my journey even after my time at DIMA, providing valuable guidance and enriching conversations up to the end.

I owe a thanks to the **IT University of Copenhagen** for allowing me to pursue non-academic passion projects like co-found and run **CS Coffee Talks** and the **PhD Club**. Through **CS Coffee Talks** I got to meet both students and professors in a casual setting which helped to keep me grounded and connected to student life.

At the **PhD Club** I want to thank **Laura**, **Jaike** and everybody else for being equally frustrated with the lack of an organisation for and by Ph.D. students to actually start one together. Your friendship and the meaningful conversations we have had helped me grow as a person. I cannot believe the progress we have made together, going from nothing to an organisation with oomph and influence, both at a local and national scale. Here we owe a huge thanks to the **PhD School** and **ITU** as a whole for their continued support, encouragement, and for taking us seriously.

I also want to give a big thanks to my girlfriend, **Kai**, whom I met through the PhD Club and who has been and continues to be incredibly supportive throughout the ups and downs of this journey. I would truly not be where I am today without you. Thank you.

I want to thank my friends from my masters:, **Jon**, **Mads**, **Frederik**, **Mathias**, and **Anders** for your support. Outside of academia I want to thank my roommate **Frederik** for your insightful discussions and help with maths. I also want to thank **Mads**, **Marie**, **Helene**, **Marc**, and **Skjøtt**.

Til sidst vil jeg gerne sige tak til min **mor** og **far**, mine **søskende** og min **mormor** og **farmor**. Selvom I ikke aner hvad det er jeg laver så har I altid støttet mig når det var svært.

Og til min **morfar** og **farfar** som jeg begge mistede sidste år: tak for alt.

All of the people mentioned above and many more are the reason why I am where I am today. In all the moments where I was prepared to throw in the towel, you have picked it up and returned it to me with love, support, and encouragement.

Thank you!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

The world is increasingly becoming more data-driven, and much of the data collection is driven by Internet of Things (IoT) devices. This has caused an explosion in generated data, leading to a larger and larger volume of data used for data analysis. The systems used for data management and analysis traditionally rely heavily on cloud resources and infrastructure, but this is becoming less viable due to network congestion, privacy concerns, and a need for real-time processing [1].

To alleviate these challenges, new computing paradigms were developed to push computation closer to the data producers thereby reducing latency, bandwidth, and energy consumption.

However, current solutions do not completely offload computation to data-producing devices but often use network resources such as modems, switches, and routers. As such, data must still be transferred from the data-producing devices to the processing devices.

This is problematic as these data producers are often quite resource-constrained with limited memory, processing, network, and energy resources to keep their acquisition and operating costs down.

This becomes especially clear if these devices are battery powered, as there exists a direct relationship between the data collected, transferred, and the lifetime of the device.

Although constraints in energy resources encourage early processing and reduction in data, there exists no single system today that utilise all compute resources from sensor device to cloud. The constraints of the sensor device differ significantly from the traditional environments in which data processing takes place and as such demands a different approach compared to traditional code offloading [2].

Care needs to be taken to make sure code offloading can be done under very limited memory and processing, and under the varied and unique network technologies that are used on these data-producing devices.

We take this different approach in TERRA, our solution to code offloading at the very edge. Integrated with a state-of-the-art sensor network database that utilise compute resources outside the sensor device, TERRA allows code offloading to be done on typical IoT platforms. As such TERRA is the missing piece of the puzzle that enables data management systems to offload code to sensor devices.

In this thesis, we introduce TERRA, discover potential energy savings for IoT devices, and investigate the effect of code offloading on the energy consumption of typical IoT devices.

1.2 Problem

We define the following hypothesis:

> *Code offloading to sensor devices in modern sensor network databases leads to a significant reduction in energy consumption.*

To verify this hypothesis, we need to answer the following questions:

R1 What is a modern sensor network database?

R2 How do we support code offloading to sensor devices in such a database?

R3 How does code offloading lead to a significant reduction in energy consumption?

1.3 Approach

Our approach is that of experimental computer science [3]. We will address the hypothesis by implementing an actual system to facilitate code offloading to sensor devices within a modern sensor network database and subsequently evaluating its performance.

The rest of the thesis will be structured as follows.

To answer research question **R1** we will in chapter 2 look at sensor databases in literature, their commonalities, and define what constitutes a modern one.

For research question **R2** we will in chapter 3 touch on the work related to code offloading in distributed databases, and in chapter 4 we describe, design and implement code offloading for a particular modern sensor network database, NebulaStream.

To answer research question **R3**, we will in chapter 5 investigate our code offloading mechanism in terms of energy and construct an energy cost model that describes the specific effect of code offloading on energy consumption. Finally, in chapter 6 we dwell on lessons learnt while developing and running experiments on a public testbed.

1.4 Contributions

In this thesis, we:

1. Define what entails a modern sensor network database.
2. Design, implement and evaluate a system that facilitates code offloading on sensor devices.
3. Integrate this system with a state-of-the-art modern network sensor database, NebulaStream.
4. Evaluate and discuss how enabling query processing directly on sensor devices substantially reduces the energy consumption associated with executing queries.

Chapter 2

Background

This chapter contains the background knowledge required to read this thesis.

We first describe what a sensor network database is by describing its defining characteristics. We then give an overview of prominent works within the field.

Then we briefly dwell on the network technologies underlying sensor network databases with a particular focus on LoRaWAN.

We end by describing evaluation techniques for sensor network databases.

2.1 Sensor Network Databases (SNDB)

A sensor network is simply a coordinated collection of sensors that collect, possibly process, and transmit sensor readings to monitor and analyse physical phenomena. Since it was first described by Bonnet, Gehrke, and Seshadri [4], it has been common to model a sensor network using the terminology of traditional relational databases. A Sensor Network can be modelled as virtual tables that contain all readings the sensor generates. This view has several benefits, such as unbounded tables and location transparency. Sensors produce unbounded streams of data, and application developers do not want to care about the topology of the network, but only about the data it contains[5]. However, with this relational database model, it might be tempting to assume sensor network databases contain the same guarantees that traditional relational database systems are known for. However, this is not the case[5, 6]. In traditional systems, the information stored is usually static, finite, and well defined. All queries are one-shot that take all data into account to produce an answer. Queries are not expected to be long-running, and this affects the query execution where locking and unlocking are used. Furthermore, queries are optimised according to a fixed-cost model and statistical information.

These techniques do not work on sensor databases, where each sensor continuously generates potentially infinite data. Data are not stored, but generated on the fly by the containing sensors. This data is inherently dynamic in nature and change according to its environment. Furthermore, sensors are usually low-energy, low-power, resource-constrained devices, and the sensor network database needs to take this into account when running queries to prolong the life of the individual sensors and ensure the stability of the sensor network. Especially in the case where these Sensor Networks are wireless (WSN) since wireless transmission consumes a significant amount of power.

So, in general, sensor network databases require a fundamentally different way to query, store, and process data compared to traditional database systems.

There exist many implementations of WSNs in production and in the literature, and while implementation details and techniques vary wildly between them, they generally have many common basic features, as described by Belfkih, Duvallet, and Sadeg [7]. In the below we will describe these common traits, by first expanding on the Data Model and the query language features they require. Then we will through the lens of network topology go through some of the bigger Sensor Network Database systems in literature. The list is by no means exhaustive, but covers the main paradigms used within the field.

A quick note on terminology. To keep it simple, we use sensor network databases interchangeably with wireless sensor networks, as we mainly focus on the database part.

2.1.1 Data Model

There are mainly two types of data in a Sensor Network Database[4, 5, 6]:

Sensor Meta-Data: Static information about the sensor that provides sensor reads. That could be processing-, storage-, transmission-capacity. Location if it is static. Network Address, etc. This also includes information and type (schema) about the physical phenomena being sensed/read by the sensor.

Sensor Reads: Dynamic data measured by the sensor. These are always related to physical phenomena in the sensor environment, and conform to the schema described by the Sensor Meta-Data. This could be e.g. temperature, humidity. Often time is attached to each measurement, making the virtual table representing Sensor Reads and time-series table.

The key here is that the **Sensor Meta-Data** is stored by the network, while the **Sensor Reads** are generated by the network and, depending on the implementation, could be ephemeral and not stored at all.

2.1.2 Query Language Features

Sensor Database Networks generally use an SQL-like language to query its network, but due to the differences defined above, the language is typically extended. Belfkih, Duvallet, and Sadeg [7] defines the following 3 general extensions:

Event-based Queries: Triggers a query on events based on sensor reads.

Lifetime based Queries: Defines there running-time and execution cycle.

Approximate Queries: Defines fault tolerances for queries. Since sensor reads often have some error attached, or adjust their power consumption based on fault tolerance, some sensor network databases natively support using this property in their query language

In addition to these extensions, the semantics and execution of SQL operators have been adapted to work on sensor network databases. The `FROM` clause refers to virtual tables, and if the `WHERE` clause refers to properties that are **Sensor Meta-Data** the filtering is usually done much earlier in the execution plan, and so on.

2.1.3 Architecture

According to Belfkih, Duvallet, and Sadeg [7] sensor network databases generally consist of two major components:

Base Station: A centralised node in the Sensor Network that serves to command and coordinate the Sensor Network.

Sensor Node: The individual node to which each sensor is attached.

The overall sensor network can then be said to contain the following components:

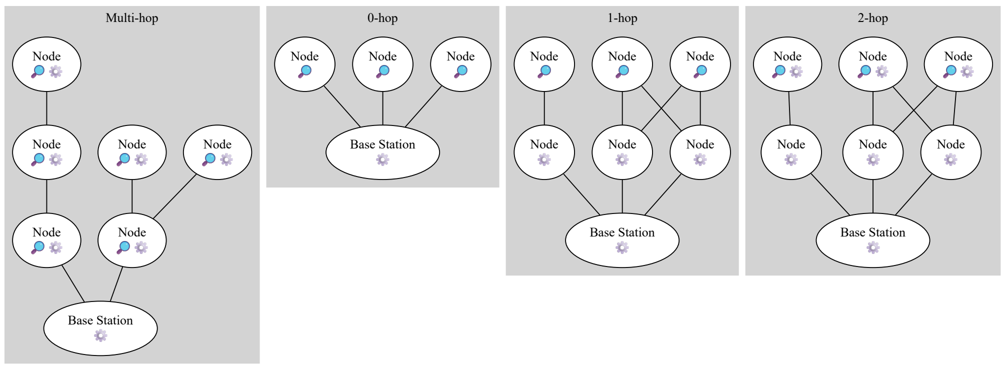

Figure 2.1: Overview of network topology. 🔍 represent a nodes ability to sense, while ⚙️ represents its ability to process queries. The amount of hops is the amount of ⚙️ data goes through before it hits a base station.

Base Station Query Processor: Receives the SQL-like query from the end user, parses it, and based on it generates an optimised query execution plan, allowing the query to be processed by the Sensor Network.

Base Station Network Catalogue: The overview of all participating sensors and the data they provide; The Sensor Meta-Data.

Query Routing Protocol: The protocol that dictates the strategy in which the query is propagated throughout the network.

Query Execution Plan: The combination of routing, placement, and timing techniques ensuring the query is properly executed with minimal cost and consistent and correct results.

Sensor Node Query Processor: The engine that executes the query received by each individual sensor node.

Sensor network databases all arrange themselves in a network topology, which affects how queries and results are propagated. Below we go through the previous and current Sensor Database Networks in the literature, in categories of their network topology. This is done since the network topology heavily affects how and if code-offloading is supported.

2.1.4 Sensor Network Databases & their Network Topology

Here we go through current and previous sensor database networks in terms of how many hops through layers a sensor read can go through, where query-defined computation can be done to it. We also try to present the different databases in chronological order, where possible. That is why we start with multi-hop databases and then go through zero to two hops. An illustration of the network topology can be seen in fig. 2.1. We indicate if a node has query processing capabilities with ⚙️ and if it is a node that performs sensor reads directly with 🔍. The hop indicates how many nodes with ⚙️ sensor reads have to travel through before hitting the base station. It should be noted that fig. 2.1 serves as a generalised representation. Specific sensor network databases may have unique restrictions, such as possessing only a single parent node, which would be different from the structure of the figure.

Multihop

The three main sensor network databases that use a multihop architecture for their sensor nodes are COUGAR [8, 4], TinyDB [9], and MaD-WiSe [10]. This is mainly ad-hoc sensor networks, where sensor nodes connect to each other dynamically in an acyclic graph.

COUGAR[11] was a system developed at Cornell University from the late 1990s to the early 2000s. It pioneered the idea of viewing sensor network databases as databases [7]. At first, it used a 0-hop architecture, where devices were connected directly to a base station and supported pre-programmed on sensor node function calls [11], but later the system was expanded to use an ad-hoc multihop topology with in-network processing for aggregates [8]. Here in-network processing is done by assigning a cluster-local leader node that collects sensor reads from all nonleader nodes and applies any aggregation computation on them. The non-leader nodes additionally collect their own sensor reads, but also combine them with reads from nearby nodes before sending their partial aggregates onwards to the leader. Hardware-wise, COUGAR runs on Sensoria WINSNG 2.0 nodes or, in a scaled-down version, on Berkeley Motes.

TinyOS is an open source, event-driven operating system designed for wireless sensor networks from the early 2000s [12]. It is an embedded operating system and therefore more akin to a framework in which you develop your application, and the final compiled binary can then run straight on supported hardware. We mention TinyOS by itself since it has made some interesting design choices to accommodate the limitations of microcontrollers, and it underpins two different sensor network databases which we describe below. TinyOS is developed in `nesC`[13], a variant of C with extensions that support event-driven execution in a component-oriented application design. Furthermore, `nesC` prohibits the use of C language features that would hinder accurate application analysis, such as function pointers and dynamic memory allocation. This allows the `nesC` compiler to statically detect data races and effectively eliminate dead code.

TinyOS heavily exploits this component-driven design to produce a very modularised OS where only the code needed is included in the final binary. Furthermore, the implementation of static analysis and the language-level restrictions facilitate significant optimisations, which diminish the number of executed code. This, in turn, maximises the amount of time the hardware can go into a low-power state.

However, these language restrictions also make it difficult to encode dynamic behaviour and change behaviour after deployment. This is something that is much needed in a Sensor Network. TinyOS tries to solve this with the Maté Virtual Machine[14, 12], which is a small efficient bytecode interpreter that can be included. It is a simple stack-based language that works on *capsules* of 24 instructions each. This is designed to allow for easy propagation of applications throughout a TinyOS sensor network. Multiple capsules can be chained together, and the language allows access to the network stack, persistent storage, and sensing. Due to Maté's design, it also provides a safe environment to execute custom software. Maté capsules cannot write to random memory locations or affect interrupts. This is a very useful property for embedded systems that rarely have kernel boundaries or other safeguards known from more traditional operating systems. TinyOS together with Maté can then provide a sensor network that is lean and efficient while at the same time providing run-time flexibility. But it is not a Sensor Network Database.

TinyDB[9] on top of TinyOS is the missing piece of the puzzle. From 2005, it applies and extends the COUGAR model of in-network processing with acquisitional query processing. That is, TinyDB now takes the acquisition of data into account, since in Sensor Database Networks this can be a costly procedure. In TinyDB it is possible to attach lifetime clauses to the queries as seen in figs. 2.2 and 2.3. In fig. 2.2, we see the SQL extension that dictates that the query should run for 30 seconds and sample the sensor every 2

```
SELECT nodeid, temperature
FROM sensors
WHERE temperature > 30
SAMPLE PERIOD 2s over 30s
```

Figure 2.2: Query with `SAMPLE PERIOD`

```
SELECT nodeid, temperature
FROM sensors
WHERE temperature > 30
LIFETIME 30 days
```

Figure 2.3: Query with `LIFETIME`

seconds. This query will then generate 15 tuples. In fig. 2.3 we get more intelligent. We simply tell TinyDB that we want the query to run for 1 month, and TinyDB will then calculate a suitable sample rate given the energy level of the network.

To do this, TinyDB looks at the cost of sampling, transmitting, and receiving from the applicable sensors and nodes and tries to estimate the maximum sample rate. Thus, this information must be available to the TinyDB query planner and must either be provided by the user at compile time or estimated. TinyDB offers support for event-based triggers to initiate and terminate queries. Considering that many microcontrollers come with external interrupts capable of waking them from a sleep state, using these triggers to activate a node for query processing allows much power to be saved. TinyDB also takes power cost into account when query planning and tries to delay sampling of expensive sensors compared to cheaper sensors when evaluating predicates. However, this does mean that different sensors are not sampled simultaneously, and TinyDB therefore includes the `NO INTERLEAVE` clause in its SQL language, to sidestep this optimisation if the samples are required to be done close together. In addition to the above, TinyDB also uses what they call Semantic Routing Trees. That is, they propagate information about the nodes themselves and the sensors they expose to their parents. This allows parents to exclude whole subtrees from queries if they are not applicable, further optimising energy cost and improving network congestion.

TinyDB, like COUGAR, also supports in-network processing for aggregate queries, providing the same benefits.

However, it has some weaknesses, such as relative errors in aggregate functions and loss of messages [7].

MaD-WiSe[10, 15] (Management of Data in Wireless Sensor network) is another Sensor Network Database built on top of TinyOS. It provides many of the same benefits, but the focus here is to allow the network to not only do same-type aggregation, but also allow the sensor nodes in the network to combine and process different types of data streams from different nodes.

TinyOS, and by extension TinyDB and MaD-WiSe, is at most supported on 14 different hardware platforms; however, it only supports a handful of sensors, and it hasn't seen much development in the last decade.

0-Hop

Here we look at systems optimised for dealing with streaming data that is fed directly to it, as opposed to the above, where data have to go through an arbitrary number of intermediate nodes.

Streaming Data Management Systems are systems similar to DBMSes but for real-time streaming data. Aurora[16] is one of the first such systems from the mid-2000s developed at Brandeis University, Brown University, and M.I.T., which works on data streams instead of static data. Data is processed as it is coming in from data sources, according to predefined and dynamically changing rules defined in a arrows and boxes framework, where it exists to applications on a tuple-by-tuple basis. Since data is now flowing in from data sources, there will be times of high load during which Aurora will not be able to process all incoming data in a timely manner. Therefore, Aurora incorporates Quality of Service levels for all its dataflows that allow it to gracefully reduce processing or drop tuples in times of high congestion. As Aurora is stream-oriented it naturally modifies its query algebra to work on streaming instead of static data, although Aurora does allow some of its data to be temporarily stored to allow for ad-hoc queries to run on. In terms of optimisation, Aurora will at any point in time have multiple data flows running simultaneously, and so the challenge is how to optimally remove unneeded data as early as possible, and ways to combine boxes or reorder to reduce, possible duplicate, work.

The ideas from Aurora have since then been expanded and worked upon in a wide variety of newer systems like Apache Kafka Streams[17], Storm[18], Flink[19], Spark Streaming[20] and many more. All of these systems are designed for the cloud in the sense that they support horizontal scaling for real-time processing of streaming data, with stateful and stateless processing. Their main differences are in the concrete way they process data, be it tuple or record at-a-time or in microbatches. These platforms have seen much use both in general and with streaming IoT data, which we will look at next.

CloudIoT is the name given to the direct integration between Internet of Things networks and the Cloud, as described by Botta et al. [21]. A paradigm that has become increasingly more popular in the last decade. In general, the key issue, as highlighted above, is limited resources for storage, processing, and communication, and the cloud as a paradigm is all about ease of scalability. For the cloud-native IoT systems, the cloud is offered as the solution to the problems of constrained hardware. Devices today can be capable enough to communicate directly to the cloud through various networks, spanning from personal area networks such as Bluetooth, through local to wide area networks like 5G and even long-range low-power networks like LoRaWAN, SigFox, or Wi-Fi HaLow. This means for end-users that they can easily integrate their sensors into their already existing cloud infrastructure and easily scale up or down depending on their needs. However, this centralisation of computing has the obvious flaw of bottlenecking at the cloud perimeter. All raw data must enter the cloud before processing. Due to this limitation, fog-centred computing gained prevalence, which we will discuss next.

1-Hop

In continuation of the cloud paradigm above, we now shift our attention to fog computing as described in Hazra et al. [22]. A term coined by Cisco to describe how heterogeneous devices could connect and collaborate to process and store data on their way to the cloud. The core idea being that data, even in the cloudIoT paradigm above, crosses many devices that contain processing power and storage capacity, like routers and switches, on its way to the final data centre. Often, multiple sensor nodes communicate with the same gateways. So, there are many opportunities for offloading processing and data storage. However, due to the limited capabilities of these fog devices, their use is limited to only data that are relevant according to the quality of service or data redundancy requirements.

Going further down from the cloud and fog computing paradigms, we have mist computing which represents computing directly on the sensor and actuator devices. A review of the literature on mist computing can be found in López Escobar, Díaz Redondo, and Gil-Castiñeira [1]. In general, mist computing describes the usage of compute resources placed at the sensor and actuator layer of an IoT-architecture. That is, using the compute that exists on the sensor devices themselves. While this does not conform to our illustration of the network topology in fig. 2.1, it fits our definition of a one-hop network with query processing on the sensing node instead of a middleware node.

It is valuable to use these resources for computation to reduce the load on the network as a whole, the latency of results, and to accommodate privacy concerns regarding the data collected. In particular in healthcare mist computing is often used to protect the privacy of the user of a device, and to ensure low latency on any action needed based on a response from a measurement.

Mist computing is an emerging field of research, and many architectures and a few specific implementations of it are proposed in literature[23, 24, 25].

We will below go through a few examples of fog and mist computing platforms.

Connected Streaming Analytics platform (CSA) [26] from Cisco is a fog platform. Here, fog infrastructure is being used by the stream processing engine being containerised and pushed onto fog devices. Concretely, Cisco created a low footprint containerised version of their stream processing engine that can run on Cisco hardware, such as routers and switches. This means that stream processing capabilities can now scale with the size of the network, given it is supported. However, it also requires the fog devices to support containers without compromising the integrity of their main function.

This stream processing engine supports a variety of streaming windows and joins on data streams using these fog nodes. One of their key ideas is the handling of joins in streaming context, since with an unbounded stream of data, the information needed to be retained in memory for a join is also unbounded. So Cisco presents two types of joins, called **best-effort join** and **coordinated join** that are designed to limit the scope of the join based on rows or time. The best-effort join simply allows the user to specify, in CSA's SQL variant, the amount of data from each joined table that should be considered when joining. In the coordinated join, these are based on timestamps, so instead of a join with e.g. the last 1000 rows, it is with data from the last minute. The consequence of this being that data has to be dynamically buffered on the fog node.

Antelope is a database system designed for constrained devices [27]. This differs from the above sensor network database by including query processing directly on the sensing device and not in the fog layer. Antelope would be an example of an enabling technology for mist computing as it facilitates query processing directly on a sensor device. It in itself does not constitute a sensor database network, but for evaluation purposes they developed one called *NetDB* in which a client can query one or more server nodes running Antelope. We mention Antelope since it provides some key contributions, and we include it here since it represents one query processing layer between data sensing and a possible base station. Antelope provides a DBMS for constrained devices, which could be useful in larger and more feature-complete sensor database networks. Constrained devices often have unique constraints, such as limited in-place writes, memory, and energy. Antelope also includes a virtual machine for execution of stack-based propositional logic called *LogicVM* that quickly and efficiently allows Antelope to select passing tuples. In addition, it includes energy-efficient indexing techniques. Experimentally, they show that the use of Antelope significantly reduces energy costs due to cutting network transfer of raw data.

NebulaStream is an IoT and data management platform in development at the Technische Universität Berlin and DFKI and is described in Zeuch et al. [2]. It tries to address shortcomings of existing cloud or fog-centred stream processing engines. These engines often have trouble dealing with geo-spatially distributed and heterogeneous data producers, which are typical for IoT scenarios. And it does this by embracing a holistic approach from data creation to final storage by integrating cloud, fog, and sensor networks into a unified platform. That means that the system is at all times aware of all nodes and the resources it contains within its network and can use these for query planning, load balancing, fault tolerance, and data partitioning. These nodes connect to each other in an ad hoc tree, where each of the nodes presents its own connections, data streams, capabilities, and limitations. However, NebulaStream operates with three layers: Sensor, fog, and cloud layer. The sensor layer only handles data collection, fog does in-network processing, and cloud does final or fallback processing if it was not able to be done in the fog layer. When queries are submitted through the NebulaStream Coordinator, the query is then optimised on a current snapshot of the network to produce an execution plan with the goal of performing as much data reduction as possible, to prevent unnecessary data transfer that would lead to congestion and to fully exploit the compute capabilities throughout the data travel from sensor to cloud. However, due to the heterogeneity of the nodes in NebulaStream, each node does not have the same compute capabilities as the others, complicating the production of an execution plan.

NebulaStream in its current state does not support the most resource-constrained but also widely used data producers: microcontrollers [28]. That is, it does not make use of the mist computing paradigm. NebulaStream requires local query compilation using either `GCC`[29] or `clang`[30] and running these on microcontrollers is infeasible. NebulaStream is also only built for Linux or MacOS, and its dependence on these is a notable constraint. Although Linux is among the most adaptable and widely supported operating systems, this restriction prevents NebulaStream from operating on cheap low-power devices commonly used at the very edge for data gathering. It is also not feasible to port NebulaStream to these devices that often don't even feature a memory management unit or have a very different thread model, if they even support threads. NebulaStream simply cannot run on devices with speeds in the hundreds of megahertz and RAM in kilobytes. For these devices to be included in the NebulaStream platform, a stripped-down version of the NebulaStream Node must be developed.

2-Hop

Here, we look at technologies where data can be processed in both the sensor and the middle layer before they reach a base station.

To our knowledge, no sensor network database currently utilise a 2-hop architecture.

Fluid Computing is the collection of all the paradigms described above: cloud, fog, and mist [31]. It is a very new term and is not fully embraced in literature. Generally, it is the unification of cloud, fog and mist resources into a single architecture that can then fully utilise all available compute, network and storage resources. This would constitute a 2-hop network, since data can be processed at all layers.

Terra + NebulaStream is our suggestion for a 2-hop network. To contextualise the solution offered in this thesis, we mention it here briefly. We augment NebulaStream with a stripped-down client that supports a limited subset of its query operators, allowing it to filter data as early as possible, thereby reducing energy cost. This will enable NebulaStream to make use of mist computing resources and fully embrace the fluid computing paradigm.

We will go through TERRA later in chapter 4.

2.1.5 A Modern Sensor Network Database

A sensor network database is not limited to utilise only the resources of the nodes that sense the data. With the introduction of cloud computing, there is an ever-increasing amount of resources available to perform processing on. However, the distances between the data generators and processors are also ever increasing. Fog computing was introduced to counteract the negative effects of this increased distance by including available resources already existing in the network layer. We have moved from ad-hoc sensor networks where data flowed through an arbitrary number of nodes until it hit the base station to sensor networks where the base station is placed in the cloud, and data has to flow from very local networks of data collectors, through gateways to the cloud. Today, sensor network databases must account for and exploit this topology. This includes handling the heterogeneous nature of resources from edge to cloud by supporting data processing and filtering on the very edge until the cloud.

Modern sensor network databases must:

- exploit the compute resources at the sensor device/edge layer to allow simple data reduction and filtering at the earliest stage possible.
- exploit the compute resources at the fog layer to allow data aggregation and joins across different streams of data.
- exploit the compute resources and scalability at the cloud layer.

In other words, modern sensor network databases must follow the fluid computing architecture. This has a multitude of benefits including reducing network congestion, energy usage and latency and increasing data privacy.

To the authors' knowledge, there currently exist no sensor database system fully exploiting all layers of this topology, and therefore matching this definition. The work presented in this thesis introduces this extreme edge layer to NebulaStream, which makes it conform.

2.2 Networking

A large part of sensor network databases is the network. After all, the key idea relies on communication between sensor nodes, and while much sensor network database research is spent developing protocols to facilitate the transfer of computation and data, not much is spent discussing the various underlying network technologies that enable this communication. However, the opportunities and constraints presented by the underlying technologies have a large impact on the capabilities of the sensor network.

2.2.1 Network Architecture

Many different network technologies exists today, both wired and wireless, which have different trade-offs. Especially for IoT the wireless technologies are interesting as these devices are often mobile and battery driven, which puts demands on the efficiency of the communication. Consequently, low power consumption is often a key factor considered when selecting a network technology. Other properties that need to be taken into account are the distances expected between sender and receiver, if the communication is one-way or both-way, the network topology needed, and so on. A comprehensive comparison of different network technologies is done in Orfanos et al. [32] and Mroue et al. [33].

In this thesis, we focus on Low Power Wide Area Network (LPWAN) technologies to cover smart city use cases, and we note that power efficiency and range are prioritised. In Mekki et al. [34], a comparison is made between the leading LPWAN technologies: LoRaWAN, SigFox and NB-IoT and we refer to this paper for details. In general, all three follow the star topology in which end devices connect directly to base stations that function as gateways between LPWAN devices and IP-based networks. As LPWAN networks, they generally provide ranges in tens of kilometres. Payload sizes vary, with SigFox having the smallest with 12 bytes, LoRaWAN with around 240 bytes, and NB-IoT with 1600 bytes.

SigFox and LoRaWAN operate on licence-free radio bands while NB-IoT is based on the LTE protocol and therefore requires a license to operate. A major advantage of LoRaWAN is that it allows private networks, allowing individuals to roll out their own LoRaWAN base stations and networks.

A study of the energy efficiency of the three network technologies is done in Singh et al. [35] and they conclude that LoRaWAN is generally more energy efficient than SigFox and NB-IoT.

In chapter 4 LoRaWAN is chosen as the network stack used for this project due to its energy efficiency, payload size and support and therefore we will now spend some time going through LoRaWANs inner workings.

We will therefore now briefly present LoRaWAN as a technology [36].

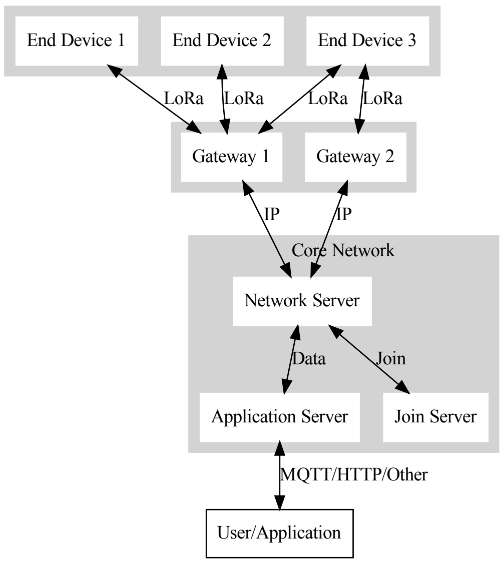

Figure 2.4: The LoRaWAN network stack

2.2.2 LoRaWAN

It is a star-of-stars topology consisting of:

End Devices: Devices usually collecting data to be transmitted over LoRa

Gateways: Devices that bridge LoRa and IP-based communication. Forwards the LoRa packets to the Core Network

Core Network: consisting of:

- **Network Server:** Central server that manages the network, deduplicates messages, and forwards it to the application server
- **Application Server:** Manages the connections between applications and the LoRaWAN network. Decrypts data and forwards it to the end user applications
- **Join Server:** Handles the authentication and encryption key management when devices join the network

A diagram of the topology can be seen in fig. 2.4.

The use of LoRa as communication technology is what enables the long range of LoRaWAN but requires the gateways to translate the physical LoRa layer to IP-based communication. To ensure the privacy of the data being communicated, the communication is mostly end-to-end encrypted, with data being decrypted at the application server in the core network, when transmitted to the end user. Note that this does mean that no processing can be done on data after it leaves the end device and before it leaves the core network, typically hosted in the cloud. Some research is looking at opportunities for how to enable processing on gateways, but this is not supported by the standard [37].

Usually, the application server has an array of integrations with which the end user can transfer data from and to their devices on the LoRaWAN network. In fig. 2.4 it is done via MQTT or HTTP, but there could be many other formats.

As mentioned above, LoRaWAN is designed to have low power consumption. Although transmitting and listening to LoRa messages require relatively low power, it is still quite costly, and as such the protocol is designed to minimise the time the radio is on. This is illustrated by the 3 device classes the LoRaWAN network operates with:

Class A: Communication is always initiated by the end device. The end device can only receive downlink messages in two receive windows (RX1 and RX2) after an uplink message. This means that any downlink communication by the core network is queued until it receives an uplink message, and then it is precisely pushed to match one of the two windows.

Class B: Like Class A, but there are fixed downlink windows where messages can also be sent to the end device. It requires gateways to send out a beacon signal for the device to synchronise with.

class C: like class A, except that it always listens.

All LoRaWAN compatible devices are required to support class A, where B and C are optional.

LoRaWAN also allows its devices to modify its bandwidth, coding rate, and spreading factor to optimise data transfer given its environment.

Bandwidth: The width of the radio signal typically 125 kHz, 250 kHz and 500 kHz.

Coding Rate: Forward Error Correction. In other words, with a coding rate of 4/5, for every 4 bits of data there will be a 5th bit for reduncancy. The coding rates supported are 4/5 - 4/8.

Spreading Factor: LoRa is based on chirp spread spectrum, which in a nutshell means that each bit is encoded as a "chirp" of changing frequency. The spreading factor dictates how slow or fast these chirps are. The slower the chirp, the longer the range. The supported values are SF7-SF12. Lower is faster [38, spreading factors].

All of these settings are combined into an array of presets called Data Rates (DR), which are set and vary for each region LoRa operates in, to accommodate for different radio regulation. Data rates usually range from DR0 to DR6, but it varies by region. for the EU868 region the data rates mean that the speed ranges from 250 bit/s at DR0 to 11 kbit/s at DR6 [39].

The regional parameters also define the maximum payload size per packet, and this also varies depending on the data rate and other factors, but in general the maximum payload a single packet can carry is 51 B for DR0-2, 115 B for DR3 and 222 B for all higher [39, sec. 2.1.6].

Since energy usage is heavily dependent on the chosen data rate, LoRaWAN has an inbuilt mechanism to automatically decide on the most efficient data rate for a given device in a given environment. This is called adaptive data rate (ADR) and is done in collaboration between device and network. It will try to use the fastest data rate possible while ensuring messages are still received by both device and network. We refer to LoRa Alliance [40] and the specification of the concrete LoRaWAN provider for details. In the case of the Things Network, these can be found at The Things Network [38, Adaptive Data Rate].

When a device wants to connect to the LoRaWAN network, it has several ways to do so:

Activation By Personalization (ABP): Pre-sharing the network keys and the device address ensures that the device does not have to go through a key-exchange process and can start talking to the network right away. However, this means that the device is tied to that specific network and maintains the same security session since the keys are never replaced. The information needed is:

NwkSKey: Network security key, used to encrypt network communication.

AppSKey: Application security key, used to encrypt information to the application.

DevAddr: The address of the device.

Over The Air Activation (OTAA): Here the device goes through a join process where network security keys are negotiated between devices and an address is dynamically assigned. This is generally more secure, as security keys can be revoked and renegotiated, and the device is not tied to a specific network but can change provider if needed. The information needed are:

Application Extended Unique Identifier (AppEUI): Id identifying the entity processing the join request.

Device Extended Unique Identifier (DevEUI): Id representing the device.

Application Key (AppKey): Secret key used to encrypt communication.

In summary, LoRaWAN is an interesting enabling technology for IoT deployment in the context of sensor network databases, since its use would allow these networks to span wider geographical distances with overall lower power consumption. However, the design of LoRaWAN places heavy and asymmetrical constraints on the amount of uplink or downlink of transmitted data and the latency of messages.

2.3 Evaluating SNDB Performance on a Testbed

The work presented in this thesis will be evaluated at the end of the thesis. Therefore, we briefly dwell here on testing strategies and using public IoT testbeds.

When evaluating systems, there are 3 broad ways to do so [41]:

Analytical Modelling: Where you represent your system using precise mathematical formulas to represent and analyse its performance.

Simulation: Using a model to replicate the system's behaviour and predict its performance.

Measurement: Taking measures from an actual system under various conditions.

All of these ways have their own pros and cons. Analytical modelling is quick and cheap, but requires in-depth knowledge of the system and its environment to properly model and simplify it. Simulation is dependent on less simplification and is better for modelling unexpected or edge-case behaviour, but does require supporting simulation software. It still requires a lot of domain knowledge to properly set up a simulation and its parameters. Finally, we have measurements which are the most accurate since it does not model but is an actual system to test on. However, this can be expensive to set up and configure.

Fortunately, for common testing scenarios, there can be a public testbed. That is, someone already did the hard work of setting up a real-world environment with ready devices and configurations. FIT IoT-LAB is such a testbed for IoT software [42]. A collaboration between several French universities, it exposes hundreds of IoT devices to bare metal programming over either a web interface or a command-line tool. The IoT devices themselves are attached to a control node that controls the device, reprograms on user request, and monitors its serial, or network activity, or power consumption. It even offers remote GDB debugging capabilities. The control node is controlled by a small Linux gateway.

IoT-LAB offers several different hardware platforms with a wide variety of microcontrollers, sensors, and network connectivity [43, Boards/Overview]. Ranging from Bluetooth Low-energy, IEEE 802.15.4 or LoRa to mention a few. Although LoRa can be used by itself, if it is used in the LoRaWAN network stack, it does require a nearby gateway to be present. IoT-LAB does at their Saclay site have a gateway situated near its LoRa capable devices, however, experience show that it has frequent downtimes lasting several weeks.

IoT-LAB offers in-depth power consumption monitoring. As described in FIT IoT-LAB [43, tools/consumption monitoring], it does this using the `INA226`[44] Current and Power Monitor. This is a current shunt and power monitor that measures current, voltage, and can be used to calculate power. Depending on the quality of the signal needed, the `INA226` can be configured with different amounts of samples, which are averaged, or conversion times. In general, averaging multiple samples or increasing the conversion time will result in more stable and accurate readings. However, it will also increase the time it takes to sample the reading. The conversion time dictates how often samples are taken and the averaging mode dictates how many samples are averaged to produce a power measurement. The final sampling rate of `INA226` is then the following:

$$\text{Power Measure} = \text{Conversion Time} \cdot \text{Averaging Mode} \cdot 2 \qquad (2.1)$$

IoT-LAB offers conversion times from 140 µs to 8244 µs and averaging from 1 to 1024 samples, which means the lowest and most inaccurate measurement rate is 280 µs.

In summary, the IoT-LAB testbed is a unique European large-scale open experimental IoT testbed allowing researchers to do everything from low-level software or protocol testing to more advanced high-level wireless services or sensor network testing.

2.4 Summary

In this chapter, we provide an overview of sensor network databases. We do this by first defining common characteristics such as the data model, query language, and architectural features. Sensor network databases are often modelled using the same terminology as traditional databases with slight modifications. Sensors are seen as producers of unbounded data streams, and the network as a whole is modelled as virtual tables.

We then go through the history of sensor network databases in terms of their network topology, where we start back in the late 1990s with multihop ad-hoc networks and end up today with 1 and 2-hop networks.

Through this historical overview, we find common characteristics and define what we mean by a modern sensor network database. A sensor database that can use all resources from the sensor-device itself to intermediate resources all the way to the cloud.

Then we dwell on the network technologies needed to underpin the different protocols of which sensor network databases are made. We focus on LPWAN networks and do a deep dive into LoRaWAN, an LPWAN protocol that is heavily optimised for long-range and low-power communication.

Finally, we introduce testing strategies for evaluating the performance of sensor network databases. In particular, we focus on the measurement strategy using public testbeds, where physical experiments are run and data is collected to evaluate the performance of the system.

In the next chapter, we will go through related work. This idea of offloading code is not unique to sensor network databases, but is also researched within the field of distributed databases which we will take a look at next. Here we also briefly discuss power consumption with relation to sensor network databases.

Chapter 3

Related Work

Here we look at some related work to sensor network databases and their power consumption. We also take a look at code offloading in the related field of distributed databases.

We start in section 3.1 by looking at power consumption in sensor network databases. We begin by examining how different operating systems for sensor devices approach power consumption optimisation.

We then move on to discuss sensor network databases and their power reduction techniques. These systems are designed to manage large amounts of data generated by sensors in real-time and often rely on acquisitional query processing and metadata about energy cost and sampling time to optimise power consumption. This is relevant to help answer research questions **R2** and **R3** as power consumption is a key metric to optimise for in sensor network databases and central to the problem this thesis addresses.

Next in section 3.2, we dive into the world of code offloading in distributed database systems. We examine the trade-offs between the different methods of operator execution: query shipping, function shipping, and hybrid shipping. We do this since the challenges of query placements are similar to the ones in sensor network databases, and since there is a lack of literature on query placement in modern sensor network databases, we look elsewhere in adjacent fields.

3.1 Power Consumption and Sensor Network Databases

Low power consumption is a key goal of sensor devices. This is because these are often wireless, battery powered, or rely on environmental energy harvesting, which cannot ensure a stable enough supply of energy.

This means that low power consumption has to be considered at all levels of a sensor network. Let us first look at the operating systems.

In TinyOS they optimise for power consumption by having the core loop be event-driven and all modules being split-phase with callbacks. This is done to ensure that no spin-locking takes place and to reduce the amount of concurrent tasks running, reducing overhead. This ensures that the CPU can go into a low-power mode as often as possible. They also provide interfaces for developers to enable their own code to support deeper CPU sleep modes when RAM or other peripherals are disabled [12].

This approach works well for TinyOS, which, at the time, outperformed other more real-time OSs in energy efficiency [45].

However, in another RTOS, Contiki [46], a module is provided to estimate the power consumption of the device in operation [47]. This is based on a simple linear model that looks at the time spent in the low-power and running mode of the processor and the transmit and receive mode of the radio. For this to work, the module requires calibration where the current draw of each component is measured. The system then uses timers to record the time spent in these different modes. This incurs a small $\approx 0.7\%$ processing overhead, but allows the OS and applications to customise their behaviour based on the current and past power consumption of the device.

However, this model has limitations, and the estimated energy consumption is not the same as measured. That is why research on instrumenting WSNs with current sensing hardware is being conducted. In Jiang et al. [48] a hardware power metre designed for easy integration into existing sensor devices is presented. It is designed for ease of use of board developers and with a low power usage in mind.

In later work, by Hartung, Kulau, and Wolf [49] they instrumented the sensor nodes of an outdoor IoT testbed with their own designed oscilloscopes to accurately measure their energy consumption in response to a changing environment. They also allow sensor nodes themselves to *tag* power measurement data by pulling pins on the oscilloscope high or low. Interestingly, since data can be collected with such speeds as to very quickly saturate the network, they keep data from the oscilloscope on their device and only transfer an index of the data to a central server - implementing a rudimentary sensor network in itself.

In the architectural and software design of Sensor databases, energy-awareness is also a focus. In TinyDB they include several features to optimise and estimate power consumption [9]. Of note is acquisitional query processing, where sensors are only read if they are part of a query, and the frequency of sampling can be varied in response to external requirements. TinyDB also relies on metadata about the energy cost and time required to sample data as part of its query optimisation.

As sensor network databases start to utilise resources in the fog- and cloud environments, focus shifts from not only optimising reads and communication at the sensor device layer, but also at the middle layers. In Frontier [50], a 1-hop edge processing platform, they strive for power efficiency by maximising throughput of their network of raspberry Pi's. This they do by duplicating operators across multiple nodes to have multiple paths for data to flow through and be processed on, making it both resilient to network failures and reducing network congestion.

That is, as more layers are taken into account in sensor database networks, several techniques need to be used to optimise and conserve energy [2].

3.2 Code Offloading in Distributed Database Systems

Related to sensor network databases are actual database management systems. Especially traditional distributed databases, since they also have to deal with large cost of data access since the data needs to be queried over a network. Many of the components are similar since sensor network databases usually share large parts of their domain-specific language with normal database systems, and since this language is declarative, the need for a query optimiser exists, to translate the parsed query into actionable code running on the underlying data. Here we briefly discuss such a system using Kossmann [51].

Traditional databases execute queries using the iterator pattern, where each operator conforms to the same iterator interface so that they can easily be chained. The optimisation engines' job is then to translate the parsed query into the most efficient series of operators. This can be done by query rewriting where optimisations that do not care about the data are done. Examples of this are simplifications of expressions and elimination of redundant predicates. Later, optimisations are performed with respect to data placement. To do this, a cost model is used that estimates the total cost of CPU and I/O, which could include seek, transfer, and latency on local data, and network I/O in the case of distributed data. Here, distance and bandwidth are taken into account together with serialisation when judging the per-byte cost. Additionally, the workload of each distributed machine must also be considered as more congested machines are slower than idle machines. This might also mean that distributing work across multiple machines might give faster response times due to parallelism, even though it entails more overall work as communication now needs to be done on top of computation. This is a core problem of distributed databases and of sensor network databases.

How do you optimally exploit the resources of the distributed database system while taking into account the communication overhead? It all depends on where and how queries are executed on data and that can be broadly categorised as query/function, data, and hybrid shipping. We will go through these next.

3.2.1 Query shipping / Function shipping

Here we simply ship the query we want to execute to the machine that holds our data and let it run and report back results. This is very simple for simple setups, but in a distributed setting some middleware is needed to join data if it is stored on different machines. Here, there can also be some discussion on where to optimise the query, as data-independent optimisations could be carried out by the submitter of the query before submission.

3.2.2 Data Shipping

This is the opposite of query shipping, where data is shipped to the machine with the query. Often these data are heavily cached on the client to allow for speedy queries on data from previously executed queries. The advantages here are that transferring data from a database server to a client is often much cheaper computationally than executing a query directly on the server. This is especially the case if a technique such as Remote Direct Memory Access (RDMA) is used, as discussed in Liu et al. [52] which we will touch upon in the next section.

3.2.3 Hybrid Shipping

Both approaches above have their advantages and disadvantages, and no single technique is usually optimal. Query shipping is preferable when the data processed is large or the server running them is powerful. But if servers are busy computing and have I/O left, data shipping scales better. Hybrid shipping tries to achieve the best of both worlds by dynamically deciding on the placement of operator execution depending on the condition of the database system and the specific query. This is done by the optimiser, which tries to place operators in a way that minimises latency or load. This does complicate the optimiser as it now has to take more factors into account when optimising a query. For the optimiser to do this, it also needs some overview of the condition of the database system. This can be done through statistical guesswork or by querying the servers in the system. Kossmann [51] discusses multiple general optimisation techniques, but worthy of a brief highlight is the two-step optimisation technique. This first step optimises a query by deciding the join order and methods, and is done up front, while the second step decides the placement of operators, and is done just before execution. This is to accommodate changing database system conditions to spread the load of queries and to allow, for example, to decide at that point whether to utilise function or data shipping. Then, this decision must take into account the current placement of the data, e.g. if it is cached on the client or not.

More recently RDMA allows data shipping to be a more attractive technique, since it allows remote servers to completely sidestep the CPU on the machine hosting the data, reducing the overhead substantially. Liu et al. [52] presents an analysis of when data shipping generally is preferred when RDMA is used together with data sampling. Unsurprisingly, data shipping proves to be better when servers are busy or if the result set of a query is large, while function shipping wins out if large amounts of data need to be read. Liu et al. [52] describes the cost of function and data shipping as in the following equation. They use the terminology of a single coordinator connected to multiple workers and as such considers data shipping when the coordinator reads data from workers and executes the query.

$$\text{COST}(DS) = C_{\text{Read}} + C_{\text{Sample}} + C_{\text{CExec}} \tag{3.1}$$

$$\text{COST}(FS) = C_{\text{Sample}} + C_{\text{WExec}} + C_{\text{Write}} + C_{\text{CAgg}} \tag{3.2}$$

Where eq. (3.1) is the cost of data shipping, consisting of the cost reading data from the workers, sampling the data, and performing the query on the coordinator. Equation (3.2) is then the cost of the shipping of the function, where we sample data, execute the query on the workers, write the result to the coordinator, and aggregate that result on the coordinator. From this we can also see that function shipping is more expensive when the cost of executing on workers becomes expensive as is the case with heavy load, so C_{WExec} is large, and when the resultset, and C_{Write}, is large. On the other hand, function shipping becomes cheaper with more workers, which reduces the cost of C_{WExec}.

3.3 Summary

In this chapter, we touch upon the power consumption considerations for sensor networks and go through the different considerations and techniques used by different networks to conserve energy. We discuss how RTOSes optimise for low-power operation. We also discuss how different sensor networks use different techniques to reduce the overall power consumption of the network by including sampling time and cost as part of their cost model or by optimising throughput. This is relevant since power consumption is at the core of the problem we address in this thesis, in particular research questions **R2** and **R3**. However, the literature here concentrates on either the power consumption of the OS or the sensor and does not look at the relationship between OS, sensor, and transmission cost in relation to the code offloaded. There is a lack of work in the state-of-the-art that examines this relationship and this thesis aims to address that.

Because of this, we then look towards the work done in distributed query processing as it encounters some of the same challenges as we encounter in sensor database networks, namely, with operator placement and how to model the energy cost of executing queries. Indeed, in these distributed database systems the condition of the network and the cost of execution and network transfer on spatially distributed machines is something that needs to be modelled in a cost model to guide operator placement. There are three strategies for operator placement: data shipping, function shipping, and hybrid shipping which is a combination of the two. The cost models for data and function shipping are represented as sums of the constituent data transfer and processing costs.

Since our work concerns itself with execution on sensor devices, we will use function shipping and follow the same procedure for a cost model in chapter 5. However, that is not to say that NebulaStream, which we integrate with, uses function shipping. Rather, they use the hybrid shipping approach where a cost model is needed to decide which shipping approach is best for each case. This again highlights the need of an energy cost model to describe the cost of function shipping in a sensor network context.

We will introduce our solution,Terra, and the integration with NebulaStream in the next chapter (4). Here we discuss our system for enabling code offloading in a modern sensor network database using NebulaStream. To do this, we define requirements for Terra, analyse them to come up with a design which we then prototype and implement. In the following chapter (5) we produce an energy cost model to model and evaluate the power consumption of Terra.

Chapter 4

TERRA - Design & Implementation

Here we introduce TERRA, our system augmenting a modern sensor database with code offloading to sensor nodes. That is, TERRA will be the software running on the microcontroller that also polls data from the physical sensor hardware. We will use TERRA to answer research question **R2** and later research question **R3**. The modern sensor network database we will augment with TERRA is NebulaStream, introduced in section 2.1.4. This is also why we require TERRA be able to run machine learning models, since that is a requirement from the NebulaStream team.

We present TERRA by first describing the functional and non-functional requirements of such a system in section 4.1. We then discuss aspects of its integration into NebulaStream in section 4.2, followed by an analysis given the constraints discussed in the previous sections in section 4.3. Then we present the design of TERRA in section 4.4 and show its implementation in section 4.5.

4.1 Requirements

Here we describe the functional and non-functional requirements that drive the design of TERRA. These describe the high-level form and function of the system. We begin with an overview of the functional requirements, then explore the major ones in detail. We then do the same for the non-functional requirements.

4.1.1 Functional Requirements

Here we define the functional requirements of Terra. They dictate the behaviour and functions of Terra.

Terra must:

F1 Data Ingestion: be able to ingest data from multiple, possibly different sensors

F2 Data Pre-processing: be able to run a pre-defined machine learning model on ingested data

F3 Query Ingestion: be able to receive queries over the network dictating how data is processed

F4 Data Processing: process data as it is ingested as dictated by requirement **F3**

F5 Data Export: be able to report the results of processing a specific query

Data Ingestion

Here, we expand on requirement **F1**. Since Terra aims to support many different sensors, it should also support many different data types.

With regard to **Data Ingestion** Terra must:

F1.1 **Scalar Ingestion:** be able to ingest scalar numerical values from sensors. For example, temperature or humidity readings.

F1.2 **Composite Ingestion:** be able to ingest composite numerical values from sensors. For example, GPS.

Data Processing

A key part of Terra is to enable on-device data processing. As such, here we expand the **F4 Data Processing** requirement. These requirements are inspired by the capabilities of traditional SQL-like queries.

With regard to **F4** Terra must:

F4.1 **Data Transformation:** be able to do one to one arithmetic transformations on data

F4.2 **Data Filtering:** be able to conditionally prevent data from being transferred or further processed

F4.3 **Data Aggregation:** be able to aggregate data using window functions like `SUM`, `COUNT`, `AVG` among others, in tumbling or sliding windows

F4.4 **Data Combination:** be able to combine data from multiple sources in all of the above computations

4.1.2 Non-functional Requirements

In this section, we specify the non-functional requirements for TERRA. These define the conditions under which TERRA will operate. The aim is for TERRA to be highly adaptable across various applications and hardware platforms, avoiding unnecessary dependencies that could restrict its compatibility.

NF1 **Hardware Compatibility:** The system must be able to run on a wide variety of microcontrollers and boards.

NF2 **Hardware Optimisability:** The system must be adaptable to each hardware platform to fully utilise its capabilities.

NF3 **Hardware Adaptability:** The system must be adaptable to make it run on new hardware platforms.

NF4 **Sensor Compatibility:** The system must be able to support multiple different sensor types and technologies.

NF5 **Hardware Availability:** The system should require as few hardware peripherals as possible to ensure broad compatibility.

NF6 **Power Efficiency:** The system should be optimised for low power consumption to enable it to run on battery-backed devices for extended periods of time.

NF7 **Network Security:** The system must ensure third-parties cannot access data during transfer.

NF8 **Network Coverage:** The systems network must support distances of at least hundreds of meters in line of sight, to support NebulaStream use-cases like smart cities.

4.2 Integration with NebulaStream

Terra is going to utilise NebulaStream as the overall stream processing engine and as such will receive the queries it needs to execute directly from NebulaStream. NebulaStream, as an existing data management platform, already contains an optimising query planner that, given the current network topology and conditions, optimises and pushes queries down to nodes. Integrating Terra into NebulaStream requires hooking into these systems. Section 2.1.4 introduces NebulaStream, but to explain how a system like Terra would integrate, we need to go more in-depth. Recall that NebulaStream is a system of a coordinator and one or more workers. The coordinator manages and orchestrates the network. It contains a source catalogue that contains all the physical data sources that NebulaStream supports [28, NebulaStream/General Concepts]. In addition to these, all physical sources are mapped to one logical source. Physical sources represent individual data producers, while logical sources represent the collection. ”Streetlamps” could be a logical source, while ”streetlamp-1” would be a physical one, contained in the logical ”streetlamps”. Each physical source is associated with a specific data source, which is a concrete provider of data for that physical source. Currently, NebulaStream supports providers from external streams such as Kafka, MQTT, or OPC, or physical files like binaries or CSV. Terra would need to be represented here in NebulaStream, when configured.

Terra also needs to hook into the query planner of NebulaStream to receive and affect the query plans NebulaStream produces. To explain how, we need to understand how NebulaStream internally works, and to do this we follow a query from its submission to results:

1. Initially, a query from a user is submitted to NebulaStream through the Coordinator.
2. Combined with the known logical sources in its catalogue, the query is converted into a logical query plan, which is sent to the Optimiser.
3. The Optimiser takes the logical query plan and the current conditions of the network, the topology plan, from the Topology Manager and creates an execution plan. This plan defines when and where the operators are executed, but also includes type-inference, filter pushdown steps, and other optimisations [28, Development/Query Submission]. NebulaStream supports different placement policies for

its operators. At the time of implementation, they have a top-down and bottom-up placement strategy that, respectively, priorities operator placement closest to root or closest to edge. This plan is then sent to the Deployment Manager.

4. The Deployment Manager translates the execution plan into individual node execution plans and deploys them to the relevant nodes.

5. On each relevant node, the execution plan is received and might be subject to more hardware-local optimisations.

6. Results are then continuously reported back through the network to the user. Additionally, a Monitoring component keeps track of the nodes performance and network congestion to re-deploy the query if the state changes.

Somewhere in this flow, Terra should integrate. But before we go to the next section to discuss when and where the integration to be, we still need to look at the internal representation of operators and expressions, since this is relevant for Terra's possible representation.

No papers have been published on this aspect of NebulaStream, so we will base this on the source code [53][1].

NebulaStream is written in C++ and uses an object-oriented design to represent its operators and expressions. Operators are represented as classes, and these classes can contain expressions that are an abstract syntax tree of expression objects or more precise derivations thereof. In addition to expressions, which can be predicates or arithmetic, operators also have an input or output schema associated with it to indicate where input data or results are stored. The above is also reflected for the serialised representation of operators and expressions.

So when NebulaStream performs query placement or query optimisation, it is all about manipulating these trees of operators and their contained expressions, and only transfering parts of those to specific workers.

Now that we know the requirements of Terra and a little about the workings and design of NebulaStream we are ready to analyse these and look at possible design opportunities of Terra.

[1]The source code is request access only, but will be published at a later date

4.3 Analysis

Here we discuss different avenues of opportunities we have for designing and implementing Terra in light of the requirements given and NebulaStreams inner workings. In section 2.1.4 we briefly mention some of the shortcomings of NebulaStream that prevent it from running on microcontrollers. The reason these shortcomings exist is because of how the NebulaStream hardware and network optimise its queries. For hardware, it performs an on-the-fly compilation of generated code utilising `GCC` or `clang` to optimise the execution plan for the specific worker platform. For network, there is no distinction between a branch and a leaf node, so workers must handle both cases and be able to accept incoming connections from multiple other workers. This naturally presents some demands on the underlying hardware that most microcontrollers cannot fulfil. Even if the NebulaStream Worker code could be ported to a microcontroller platform, it would not have the prerequisites to run on it.

The other side of the coin is that the environment or setup in which NebulaStream assumes its workers are run does not apply so well to microcontrollers. NebulaStream workers assume at least equipment on the level of Raspberry Pi's, with TCP/IP connections. Microcontrollers can find themselves deployed in remote areas with not enough electric power to power a Raspberry Pi or in an environment that cannot facilitate a TCP/IP connection.

That is why a new stripped-down worker has to be developed, and this is what Terra is.

This means we have to cut some of the features of the worker to feasibly run on microcontrollers. The two big ones are no local query compilation and assume that Terra is always the leaf node of the network. That is why these features are not included in the functional requirements in section 4.1. To maximise power savings, Terra should go into the lowest possible power state of the underlying hardware as often as possible. This should cover requirement **NF6**.

Requirements **NF1** to **NF4** all concern Terra's support for hardware, sensors which is very OS dependent. We will not implement Terra on bare metal, but utilise an operating system, in the microcontroller sense, to build on top of. The choice of this OS will be discussed in section 4.5.

Besides the above, there are overall four challenges to address:

Communication: How do we represent and transfer queries and results?

Query execution: How do we represent and execute queries?

ML inference: How do we represent and infer neural models?

Integration: How do we represent TERRA in NebulaStream?

We will now discuss these in this order.

4.3.1 Communication

TERRA needs to communicate with NebulaStream to receive queries and transfer results. The functional and non-functional requirements for communication are requirements **F3** and **F5** and requirements **NF7** and **NF8** and to a degree requirements **NF3** and **NF6**.

To address these, there are two major decisions to make. Through which technology should communication occur and in what binary format? These are interconnected, as the constraints in one affect the other. To answer these questions, we first look at the communication options available and used for microcontrollers. NebulaStream uses TCP/IP which requires a somewhat stable connection. This is not directly compatible with several of the often used network technologies used for IoT devices. In section 2.2, we briefly discuss IoT networking and its unique requirements. Our focus quickly converged on LoRaWAN, as it fulfils the requirements we have for TERRA. It is an LPWAN technology that is focused heavily on low power and is based on an open standard. It fits well with NebulaStreams use case as a system for smart cities [2]. But so do alternatives such as SigFox and NB-IoT as described in section 2.2. However, as described there, LoRaWAN is generally more energy efficient than the alternatives and due to its open standard and sufficient payload size we chose this as our used network stack. LoRaWAN is also supported on the testbed we will later run our experiments on, see section 5.3.1. However, requirements may change, and to support other hardware platforms or network options, TERRA will be designed with low coupling to allow easy adaptability to other network stacks.

Using LoRaWAN would also illustrate the challenges of including the very edge as a resource for code-offloading, and to make sure TERRA makes available use cases that are outside the feature set of NebulaStream currently but within the scope, we focus on low-power, high-range solutions.

Since it is designed for IoT and sensor devices, it assumes that there is going to be much more uplink data compared to downlink, and it by default does very little to ensure message delivery. A key constraint is that downlink by default can only happen just after an uplink, and that has to be taken into account in the Terra design. The effect of this is that Terra, if it has no queries to execute, would have to send messages to check if there are queued downlinks available. The choice of LoRaWAN covers requirements **NF6** to **NF8**. Ensuring the network component of Terra has low cohesion will cover requirement **NF3**.

We now turn to the question of binary format. This is closely tied to query execution, which we will look at next, but for now we focus on the serialisation and on-the-wire format.

Due to LoRaWAN's limited packet sizes and slow bitrates, the binary format has to be compact to save power. Furthermore, it has to be serialisable by both NebulaStream and Terra. NebulaStream already uses Protocol Buffers [54], a compact binary format maintained by Google. Although there exist other text-based or small binary formats, protocol buffers are space-efficient, fast, and generally suitable for constrained devices [55, 56, 57]. Other formats, like Apache Thrift or flatBuffers, might be marginally more efficient or suited, but for compatibility with the existing NebulaStream, we judge Protocol Buffers to be good enough.

Protocol buffers are schema-based, so a schema needs to be defined that represents both queries and results or any other needed communication between Terra and NebulaStream. The representation in protocol buffers might serve as Terra's internal query representation, based on how the Terra query execution engine is built, which could eliminate the need for a potentially costly conversion. We will discuss this in the next section on query execution, but in that case, we address requirements **F3** and **F5**.

The single-packet constraint does put a hard limit on the size of queries and results we can transfer. Transferring data over multiple packets is a possibility, but since there is no in-built support for packet reassembly, and it is not the main objective of this thesis, we leave this as future work.

In conclusion Terra should take the unique constraints of LoRaWAN into it's network design to ensure timely downlink of queries, but also not be too coupled to LoRaWAN as to make replacing or adding support for new network stacks in future difficult.

4.3.2 Query Execution

Now we divert our attention to the core of Terra. Handling of the partial queries it is going to receive from NebulaStream. This handling will at a high level be much like current workers that:

1. Receive queries and deserialises them
2. Possibly perform local optimizations per query or across queries
3. Execute them
4. Serializes and returns results

Before we take a look at the binary format and the execution engine of queries, we briefly discuss what kind of computation needs to be done. Requirement **F4** and its subrequirements describe the kinds of computation that Terra needs to do. It also indicates that the processing is triggered by new data. We quickly note here that this means no data will be collected and processed if no query is running in Terra. This is by design and follows acquisitional query processing. We also note that while NebulaStream strives to implement adaptive sampling rates [2], it is not implemented yet, so while dynamically adjusting sampling rates in Terra would be nice, it is not a requirement.

The subrequirements of **F4** specify further that Terra must be able to do data combination, projection, transformation, filtering, and aggregation. This will require the execution of both arithmetic and Boolean expressions. It will also require the ability to save intermediate results across executions to support aggregation.

Requirement **F1** governs the types of data we need to process. Both scalar and composites, both integers and floating point numbers. Terra should be able to represent the above in a space-efficient and easy-to-execute format.

Let us first look at the binary format of the queries. In the previous section, we judged protocol buffers to be a suitable format for network transfer. However, that applies to all transfers. The format of the queries themselves has not yet been decided. There are three primary ways to represent computation on the wire:

1. Cross compiled machine code
2. Intermediate representation executed by a Virtual Machine
3. Parametrized functions

We will now briefly discuss these in that order.

An avenue would be to cross-compile the generated query execution code on the NebulaStream Coordinator and send the resulting binary blob to

Terra, which then would load it into memory, cast the address to a function pointer, and execute it. This would produce highly optimised machine code that would be very energy efficient to execute. For this to work, there would have to be a lot of coordination between the cross-compiled code and Terra with regards to size and location of inputs and outputs, but this could be given as input to the function. Additionally, if Terra provides specialised or locally optimised functions for use by the cross-compiled binary, then the address of these functions has to be passed to the compiled binary, either as parameters or by patching the binary after receiving it. As long as this is well defined at both Terra and NebulaStream, this approach is viable. However, there are several problems with this approach.

- It requires the complete toolchains of all devices used that Terra can support to be installed and managed on the worker.
- It requires NebulaStream to be aware of which hardware platform a specific node is using, and the constraints that node might have regarding its RAM usage, stack size, and so on and to take this into account when generating the node execution plan
- It requires Terra to blindly trust the code that is provided by NebulaStream. Embedded devices often do not have any protections, so any code dynamically provided to Terra would have full access to all resources and could do anything from accidentally crashing the system to actively destroying it. Since Terra is designed to be deployed in potentially remote locations this would be very bad
- Binary sizes can vary wildly, but would likely be larger than intermediate representation or parametrized functions. The binary size could possibly be compressed to save space, but that would then require a possibly expensive decompression step

As an alternative to the cross-compiled machine code, use an intermediate representation executed in a Virtual Machine. Like Maté, we briefly presented in chapter 2. Maté is closely tied to TinyOS, and is not useable in our case. There are other virtual machines and code representations built for microcontrollers we could use like MicroPython [58], WebAssembly [59, 60], JavaScript [61] and even eBPF [62]. This would produce safer and smaller code, but still with the same or similar expressibility as machine code. However, these VM's while optimised for microcontrollers are still quite heavy in binary size, and so is their respective code that needs to be transferred.

Using parametrised functions instead, we can significantly minimise the required instructions, thereby optimising code size.

With these we would predefine in TERRA the operations available, and running an operation would simply be transferring an index and the parameters needed for that operation over the wire. Since NebulaStream has a limited number of operators, each with a limited set of parameters, this would be a very space efficient representation. However, representing expressions this way is cumbersome and less efficient since parsing and executing require traversing a tree structure.

TERRA could utilise a combination of the 2 above techniques. Use parametrised functions for supported NebulaStream operators and an intermediate language, preferably stack-based for ease of implementation and efficiency, for the expressions needed.

This would be the best of both worlds: Space efficiency of parametrised functions, with ease of implementation of stack-based arithmetic and logical expressions. It is also similar to what is done in Antelope with the *LogicVM* virtual machine [27] as presented in section 2.1.4.

The last piece of the puzzle is the mapping of sensor data. Recall that NebulaStream requires logical schemas assigned to data producers. This means that NebulaStream up front need to now know what data a TERRA node offers. TERRA also needs to know on which data NebulaStream wants to operate. This means that there must be a mapping of sensors to inputs that are identical in both TERRA and NebulaStream. As mentioned in section 4.2, a TERRA node would be part of a logical source in NebulaStream. A logical source has one or more fields, each represented by a name and a data type. TERRA would need to map each sensor to one of these fields. The naive approach would be to simply reuse the id of NebulaStream and transfer it as part of the query. However, this is a variable-size string which would be wasteful to transfer. It would be better to map each string to a single integer to save space and make sure that the same mapping exists in TERRA. It could possibly exist implicitly as the initialisation order of sensors in TERRA, since this is fixed at compile-time.

Since data is produced by sensors, one could also use a sensor environment model, a description language, to classify sensors and actuators in TERRA [63]. This would avoid the need to preload a common configuration that assigns sensors and logical schema fields to each other, since TERRA would know that it needs e.g. temperature data and would then read its temperature sensor. However, this will not work in the presence of duplicate sensors and would require NebulaStream to use the same model to describe its fields.

4.3.3 ML Inference

Supporting inference of machine learning models is luckily relatively simple, so fulfilling requirement **F2** is not an issue. This can be done using Tensorflow Lite [64], to convert a traditional Tensorflow model into a model compatible with microcontrollers. This model can then be included into a C/C++ program and inference can be run through it using a provided library.

4.3.4 Integration

Finally, we discuss the nature of the integration of Terra into NebulaStream. In section 4.2 we went through the inner workings of NebulaStream to uncover NebulaStream specifics that we need to design around and to see where possible integrations could be. Here there are several options depending on how Terra-aware we want NebulaStream to be. Currently, NebulaStream operates with workers and a coordinator. Terra is a stripped-down worker. This concept does not exist in NebulaStream, so for a deep integration there would have to be a significant architectural change.

However, another opportunity presents itself, with the choice of LoRaWAN in section 4.3.1. Essentially, since LoRaWAN is a self-contained network from end devices to the core network, there is only going to be a single integration point between LoRaWAN and NebulaStream, which is the application server. Terra could then be represented as a single data source in NebulaStream, covering all devices within the LoRaWAN network that shares a logical schema. Since most major LoRaWAN providers and NebulaStream support MQTT integration, this would be rather straightforward. But Terra is not only a data source, but also a worker, and as such NebulaStream needs to modify queries and operator placement based on its capabilities. So, there needs to be an integration into the query placement and rewriting engine of NebulaStream, but without all the dependencies that being a traditional Worker entails. This could be done by representing the incoming data from Terra as a data source and then, conditional on the source being present in the query, hooking into the query optimisation process early to capture and separately pull out any operations that can run on Terra, before handing the modified query back to the conventional

NebulaStream optimisation process. Figure 4.1 shows an example of such a rewrite rule, with ”LP Source” being the LoRaWAN data source. Essentially, go from source to sink in the query plan and move operations to Terra until no longer possible. The data arriving at the source would then already have had these operations done to it, and from NebulaStream side these operations would not exist and would not be part of any later processing. This procedure would also contain the area of NebulaStream needing to be aware of Terra substantially, which would ease integration.

However, it also has some downsides.

- Implementation-wise, it might require hooking into, and modifying parts of NebulaStream that were not designed to be modified. We will discuss this in the following sections.
- It is dependent on computations needed to run on data from Terra is placed directly after the source, and not joined with other data right away.
- Depending on how early into the optimisation process we hook, we might also miss some optimisations, or optimisations could be made that would allow more computations to be performed on Terra. However, since a key restriction on Terra is that it only operates on its own data and does not receive data from outside or other nodes, this would be limited to optimisations such as filter pushdowns based on constants.
- The LoRaWAN data source would be placed on a single normal NebulaStream Worker, which could be a bottleneck and a redundancy problem. One could scale out horizontally and have multiple workers connected to the same MQTT broker, but that would introduce duplicate data if no partitioning is done.

4.4 Design

Here we describe the overall design of Terra at a high level, and we reserve the next section to talk about the implementation details. This section is a description of the concrete choices made based on the discussion in the previous section. We start by describing the design of Terra; the application that will run on the sensor device. After that we look at the integration in NebulaStream.

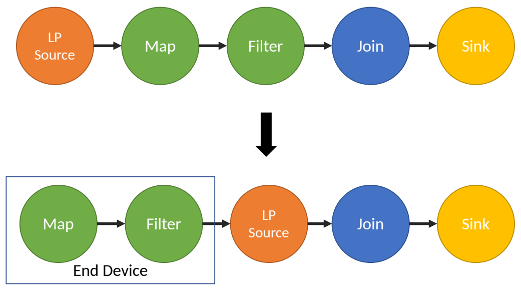

Figure 4.1: Suggested rewrite of query plan

4.4.1 Terra

Terra is going to continuously fetch and process data, and depending on the outcome of the processing, return results and check if new queries exist. To give an overview of Terra we present the program flow in fig. 4.2. In the following sections, we will go through the different responsibilities of Terra one section at a time.

Sensor Readings, Number Handling, & Tensorflow Lite

Terra is going to support readings from one or more sensors. Each of these sensors can produce one or more outputs. For example, a temperature reading is usually a scalar value, while a GPS position is usually latitude, longitude with a possible altitude. In Terra we store these readings in what we call the ”environment”; an array in which we can store numbers that can then later be read from and written to by operators, which are explained in the next section.

The environment's role is as working memory of operators, and in many ways function as such. However, we track the origin of numbers in the environment, be it from sensors or queries. This is done to make sure that the data exports contain only the results after processing, as per requirement **F5**.

Terra stores numbers in a type-agnostic fashion, such that the environment can contain both integers and floating-point numbers of different sizes. This is done both to simplify implementation, but also to reduce the size of numbers for network transfer. However, type information is saved so that any arithmetic is optimised.

So when sensors are read, it populates the environments with the readings sequentially starting from address 0 up to a maximum size that is preconfigured at compile-time.

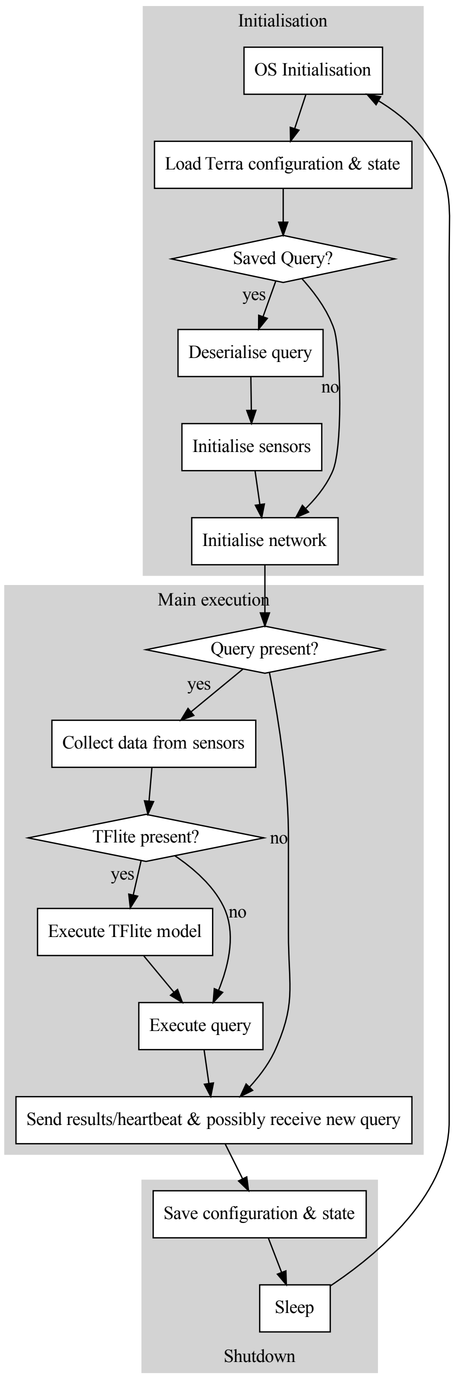

Figure 4.2: Decision flowchart of TERRA's execution

Instruction	Operation	Instruction	Operation
`CONST`	Constant number	`MOD`	$a \bmod b$
`VAR`	Variable	`LOG`	$\log(a)$
`AND`	$a \wedge b$	`POW`	a^b
`OR`	$a \vee b$	`SQRT`	$\sqrt{a}$
`NOT`	$\neg a$	`EXP`	e^a
`LT`	$a < b$	`CEIL`	$\lceil a \rceil$
`GT`	$a > b$	`FLOOR`	$\lfloor a \rfloor$
`EQ`	$a = b$	`ROUND`	$\text{round}(a)$
`ADD`	$a + b$	`ABS`	$\|a\|$
`SUB`	$a - b$	`LTEQ`	$a \leq b$
`MUL`	$a \times b$	`GTEQ`	$a \geq b$
`DIV`	a/b		

Table 4.1: Expression instructions and their operations

After the sensor reads, if there is a Tensorflow Lite (TFLite) model, execute it on the values in the environment, and any results are appended to the existing values. The TFLite model exists as a sort of pseudo-sensor. It produces sensor values; however, it is dependent on input from other sensors, which is unique. All other sensors can be read in parallel if supported by the communication protocol.

Network Format & Query Representation

Terra implements a simple stack language to support arithmetic and logical operators. To use the same language and execution for both, we use the rules of C to describe Boolean values. The integer 0 is false, all other values are true. An expression is then a list of instructions and numbers executed on a stack. The result of an execution is the top-most value on the stack after all instructions have been executed. Almost all instructions read their operands from the stack, with the exception of `CONST` and `VAR` which respectively push a constant number to the stack, or push the number stored at a specific index from the environment to the stack.

The full list of supported instructions is shown in table 4.1.

It follows reverse polish notation when working on the stack, which is read from left to right. That is, if the stack contains numbers a, b in that order, the instruction `SUB` will then pop those two numbers, execute $a - b$, and push the result onto the stack.

As an example, if we have a temperature sensor that measures in Fahrenheit and we want to convert it to Celsius, we use the formula $((F-32)\cdot 5)/9$. If the sensor read is stored in environment position 0 then the list of instructions would be:
`VAR, 0, CONST, 32, SUB, CONST, 5, MUL, CONST, 9, DIV`

These expressions can then be included in operations that are chained together to form queries. The stack in which expressions operate is cleared after each expression; however, the environment persists during the whole query. Terra is going to support the following operators, who all mimic their NebulaStream counterparts:

Map Operation: A tuple to tuple conversion and consists of:

- An expression
- In which environment variable the result should be stored

Filter Operation: Consisting of:

- A boolean expression, the result of which indicates whether computation should be cancelled or not

Window Operation: Consisting of multiple fields that indicate:

- Aggregation information:
  - Type: `MIN, MAX, SUM, AVG, COUNT`
  - Which environment variable to aggregate on
  - In which environment variable to save the result
- In which environment variable to save the first observed value
- In which environment variable to save the last observed value
- If the inclusion of values into the window is:

  Tumbling: Defined by a non-overlapping size

  Sliding: Defined by a potentially overlapping size and slide length

  Threshold: Defined by a predicate on measured data and minimum size

  These sizes can be based on both the measurement time and the count. For example, we could define a tumbling window of size 5 measurements or of size 2 minutes. For the threshold, it is based on a predicate on data values, so an example would be $F \leq 20$ where F would be a sensor reading. Then aggregation would only be executed on values passing the criteria, and the window would close at the first non-passing data, if the window size is above the minimum defined.

Finally, we need to address the format of the results sent from TERRA to NebulaStream. These are simply a list of numbers. Here we need to make sure the indices of these processed numbers match up with what NebulaStream expects. Since query processing in TERRA is defined by NebulaStream, this comes for free. As fig. 4.1 illustrates, TERRA simply executes a moved operation, so the placement of the result of that operation is identical to what the NebulaStream Worker would have done had TERRA simply sent raw data. As such we do not need to track when and where processed data is stored as TERRA's behaviour mimics NebulaStream's.

Configuration & State Management

To support requirements **NF1** to **NF4**, requirement F4.3, and requirement **F2** TERRA needs to be configurable in its resource usage, and it needs to save state across executions. We are first going to look at the configuration options we expose with TERRA. These are options you define when you build TERRA, and are going to be fixed and nonmodifiable throughout the devices lifetime.

Execution Epoch in seconds: How often is TERRA going to execute.

Stack Memory size: How large of a stack is used for expressions.

LoRaWAN configuration: As described in section 2.2 LoRaWAN requires a bunch of parameters be set so the network can connect and identify the individual devices. In TERRA we only support **Class A** devices using **OTAA**, and thus TERRA requires the **Region**, **App EUI**, **Dev EUI** and **App Key**.

Default Query: We give the option to provide a query at build time to save on the initial transfer cost.

Forced listen: As described in section 2.2, for **Class A** devices LoRaWAN dictates an uplink before a downlink. Initially, there might be no running query in TERRA, or the query running might have very selective selectivity. To ensure that new queries can be received in those situations, we provide TERRA with the option to send heartbeats to facilitate the acquisition of new queries. This configuration option dictates that a heartbeat is sent after *n* executions, where *n* is the configuration option.

Sensors: Which sensors are enabled in TERRA and their configuration. Note here that order matters, as this dictates the order in which the data would be read into the environment.

Max window operators: Since window operators by definition require state to be saved across executions, we define the maximum amount of concurrent operators that we support.

In addition to the build-time configuration, we need to store some state across TERRA executions. This involves reading and writing from persistent storage to allow TERRA to sleep between executions, as these low power states often do not preserve RAM.

This state, besides the configuration above, will be:

LoRaWAN State: The negotiated keys, and additional state required by LoRaWAN.

Epoch counter: A simple counter, incrementing for each epoch executed.

Running Query: The current running query of TERRA. We save this in serialized form to save on RAM.

Window State: The state of each window operator. We will expand on the window operator in the following section, but the state saved is:

> **Aggregation value:** The partial aggregate.
>
> **Start value:** The first value seen in the window.
>
> **End value:** The last value seen.
>
> **Count:** Count of values seen.
>
> **State:** The current state of the window.

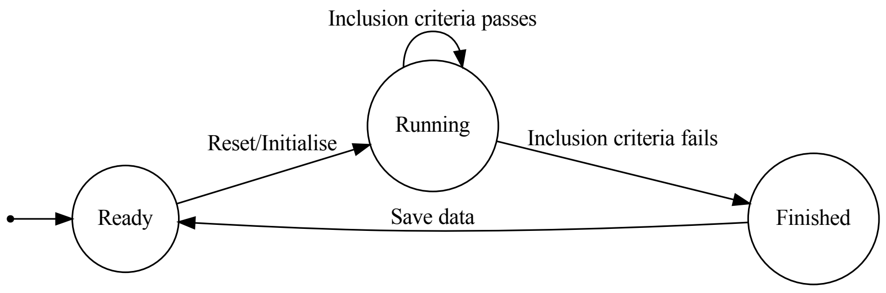

Figure 4.3: General window state machine

Query Execution Engine

Here we go through the query execution engine of Terra. That is, the way we execute expressions and operators. Expressions are executed according to the rules described by the query representation, and are simply executed one instruction at a time. Expressions themselves do not write or modify any state. That is handled by the operators they are a part of. For the actual calculations, recall that numbers are handled in a type-agnostic fashion, but with type information preserved, so when performing mathematical operations, it is done according to the rules of the underlying language.

A query is a series of operations that operate on the same environment. In the following, we go through each operation.

The Map Operation is done according to its description above. It has its expression executed, and the result of the computation, the top of the stack, is stored in the environment index dictated by the map operation. Note that this means that a map operation will always only output a scalar value. Multiple results can be emulated by chaining multiple map operations.

The Filter Operation is much like the map operation except that the expression is a predicate and its result is not stored, but checked for truth. If the result is true, nothing happens and the next operation in line will be executed. If the result is false, the execution is halted and the whole query is marked cancelled, meaning that a result will never be transmitted.

The Window Operation is slightly more complicated than the others. All window types are implemented as simple state machines and follow the diagram shown in fig. 4.3. All state changes are triggered by an execution, and therefore new data. The states execute as follows:

Ready: Here we initialise state and fields, and immediately go to the ”Running” state.

Running: Here we check if the inclusion criteria is met, and if so we execute an aggregation, and save it. If the inclusion criteria is not met, we transition to the ”Finished” state.

Finished: Here we save the final aggregate, start and end values in the environment for submission and transition to the ”Ready” state.

For the three types of windows we support, we simply modify the inclusion criteria. Note that with the sliding window type, we can have multiple windows that overlap each other. To support that, we simply store an array of windows and run them serially on the same data. All aggregation functions we support, support using a scalar state value and a count to store its partial aggregations.

4.4.2 Integration into NebulaStream

Here we discuss the design of the NebulaStream integration. This integration consists of two parts, as hinted at in the analysis. One part covers the handling of the connection to the Core Network and what it entails. And another part covers the capturing and rewriting of queries. These parts and their interaction can be seen in fig. 4.4.

LoRaWAN Proxy Source

This will be the connection handling part on the NebulaStream side. A NebulaStream data source must primarily be able to receive data which will then be processed by NebulaStream. To do this, our data source must in its configuration get the information needed to connect to the core network. Additionally, it needs to map the incoming data from the sensors to a correspondingly configured logical source, so the configuration must also include this mapping. We discussed this mapping and alternatives at the end of section 4.3.2, and the mapping we do is the one we describe there. It is simply a list of logical source field names, where their position corresponds

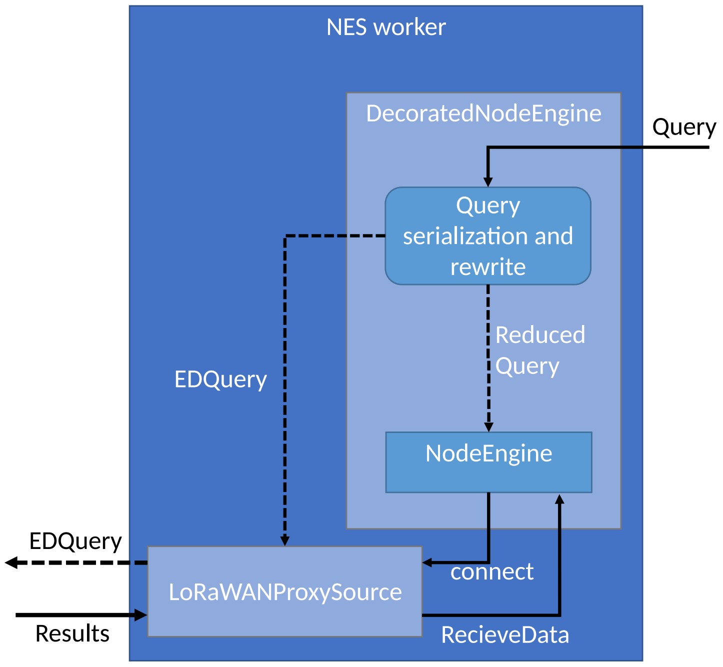

Figure 4.4: Suggested integration into NebulaStream

to the sensor to which they map. As an example, say, we have a sensor device that records temperature and humidity, defined in that order. We also have a logical schema that names first value "humidity" and second value "temperature". We would then in our data source configuration define the sensors: `temperature, humidity` to indicate that the value given at index 0 corresponds to the logical schema "temperature" field and the one at index 1 to the "humidity" field.

Since the data source already provides a connection to the core network, we reuse this for transferring queries. The only additional information that we need for that is the names of the devices so that we can send queries to them. So, we include a list of devices in the configuration.

Decorated Node Engine

To capture and modify the incoming queries in the NebulaStream Worker that contains the above-described data source, we decorate the node engine in that worker. This node engine will capture any incoming queries and, if they use the LoRaWAN Proxy Source as a source of data, they will try to serialise the query from start to finish. As soon as a operation or expression is hit that cannot be serialised, the serialisation stops, and the successfully serialised part of the query is sent to the LoRaWAN Proxy Source to be sent through the network to Terra, while the rest of the query is passed down to the actual node engine, and is subject to the full NebulaStream pipeline. This works, since we can only serialise operators that are supported on Terra, so any unsupported operation will fail serialisation.

When the query is later executed on the Worker, it will fetch the data from the LoRaWAN Proxy Source that has already been processed by the offloaded part and forward those data to the rest of the query on the worker, making the whole process transparent to the rest of NebulaStream.

4.5 Implementation

To implement the above-described design, we start by defining our communication format and modifying NebulaStream, which we discuss in the next sections. Then we construct a proof-of-concept in MicroPython. Through a bachelor thesis by Laurits Bonde Henriksen and Markus K. R. Johansen[65], this proof-of-concept was rewritten into C, which we used to further develop Terra into its current form.

The code for the prototype and final version of Terra is accessible here:

MicroPython Prototype: `https://github.com/FlapKap/nebulastream-ed-runtime`

Terra: `https://github.com/FlapKap/Terra`

The NebulaStream integration is implemented as a branch in their repository and can be accessed here when the code is published: `https://github.com/nebulastream/nebulastream/tree/3159_add_lorawan_support`.

4.5.1 Protocol Buffers

The serialisation format is key for the communication and handling of queries and results. The whole description can be viewed in the repository[2], but here we give an overview. The complete representation can also be seen graphically in fig. 4.5[3]. We define each operation as a separate message that covers the fields defined in the design. Notable is the `Data` message that

[2]`https://github.com/FlapKap/Terra/blob/master/app/terra/terraprotocol.proto`

[3]Generated with modifications by `protodot` (`https://github.com/seamia/protodot`)

covers instructions and numbers. It uses `oneof` functionality of protocol buffers to contain a specific type of number, be it an integer, an unsigned integer, float, or an instruction. Expressions are then just a list of `Data` instances.

4.5.2 NebulaStream Integration

The above described design is implemented in NebulaStream by defining and including C++ classes and `YAML` configurations for the LoRaWAN Proxy Source and the Decorated Node Engine. The implementation of these follows from the design description. Concretely, we make the following additions to NebulaStream:

`LoRaWANProxySource`: The data source in NebulaStream that handles incoming data from Terra and pushes queries to Terra. This is done by setting up an MQTT connection to the chosen core network and exposing methods to send and receive messages. The received messages are unpacked and transferred to the rest of NebulaStream as any other source. Initially, we implement support for The Things Network[38] and ChirpStack[66]. The Things Network is a publicly accessible global LoRaWAN network, and ChirpStack is a self-hostable LoRaWAN network stack for private deployments. To support different LoRaWAN providers in the future, we implement `LoRaWANProxySource` with an abstract `NetworkServer` class that can then be implemented to fit to specific providers, so to abstract away the network details from the rest of the LoRaWAN Proxy Source. Configurations for these follow the NebulaStream configuration standard in YAML, and an example can be seen in listing 1. This contains the information needed for the `LoRaWANProxySource` to connect to the LoRaWAN network, and to map the incoming data from Terra to the logical source.

`EndDeviceProtocol.proto`: The protocol buffers description the communication between Terra and NebulaStream. See section 4.5.1 for details.

`EndDeviceSerialisationUtil`: De/serialisation logic for NebulaStream. Here we implement the function `serializeQueryPlanToEndDevice` that, if the query plan starts with a `LoRaWANProxySource`, traverses the query plan from start to finish, serialising all supported operations while removing them from the query plan in place. When an

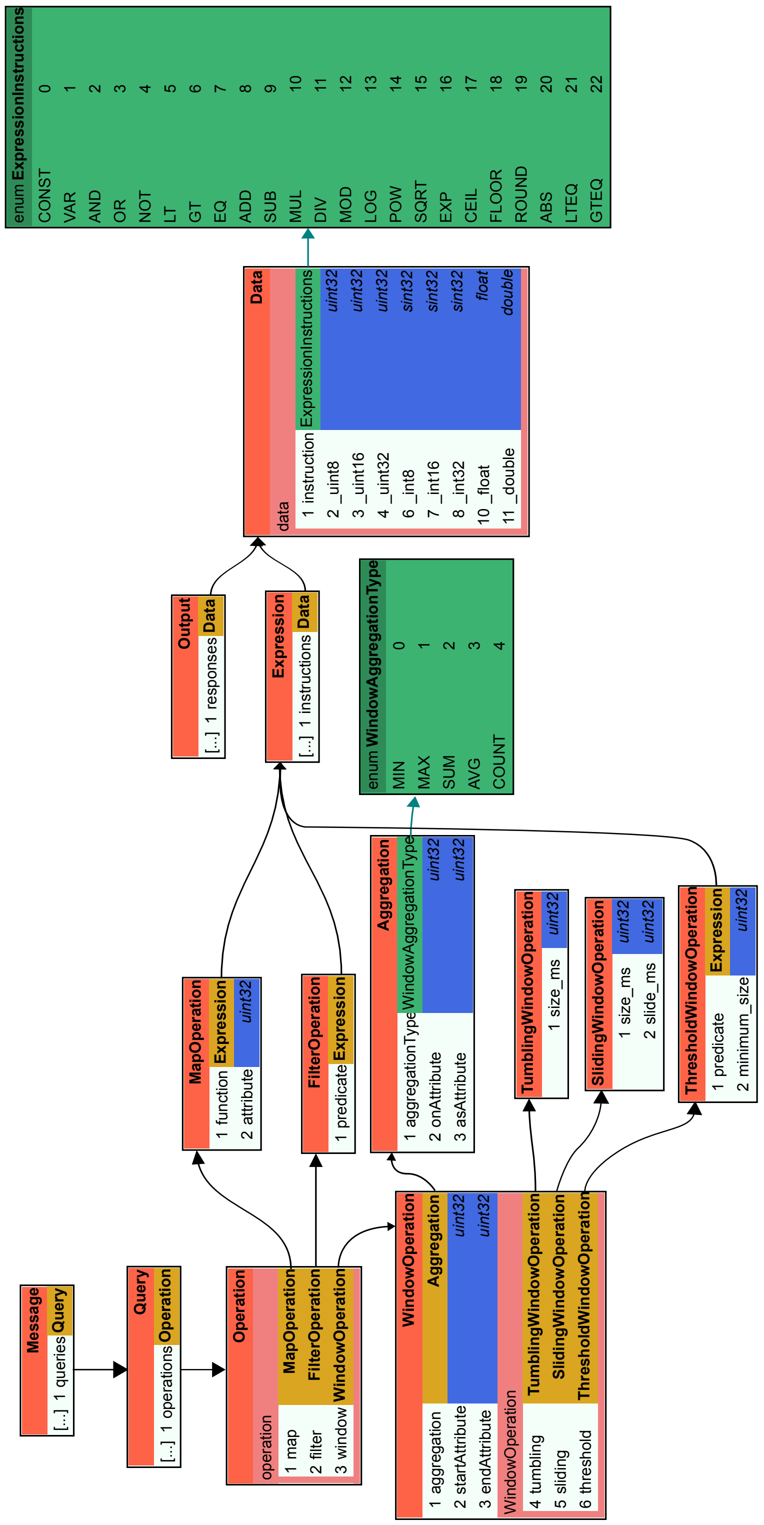

Figure 4.5: Protocol buffer schema used for TERRA

unsupported operation occurs, we simply stop and return the serialised query for Terra.

`DecoratedLoRaWANNodeEngine`: Our decorated engine. This wraps the original node engine, but is created with a reference to the LoRaWAN-ProxySource. When a query is registered it is serialised through `EndDeviceSerialisationUtil` and the serialised query for Terra is passed to the LoRaWANProxySource, while the now reduced query for NebulaStream is passed to the real `NodeEngine`.

In addition, we also add `LoRaWANProxySourceType` and `LoRaWANProxySourceDescriptor` which are typed representations of the configuration shown in listing 1.

To support the above, we need to add references, methods, and types to describe the `LoRaWANProxySource` in the following places:

- `SerializableOperator.proto`
- `PhysicalSourceType.{c|h}`
- `ConfigurationNames.{c|h}`
- `SourceCreator.{c|h}`
- `PhysicalSourceFactory.{c|h}`
- `ConvertLogicalToPhysicalSource.{c|h}`
- `LowerToExecutableQueryPlanPhase.{c|h}`

To allow for decoration of `NodeEngine` we need some of its methods to be virtual.

`NodeEngineBuilder` is were we decorate the node engine if a `LoRaWANProxySource` is part of the worker configuration.

In addition to the above, we also add unit tests.

And with these changes NebulaStream is ready to support Terra.

We note that since development of NebulaStream and Terra is happening side by side, the latest release of NebulaStream does not support the latest version of Terra. We develop Terra against the common protocol buffer format.

```yaml
physicalSources:
  - logicalSourceName: logical_source_name
    physicalSourceName: physical_source_name
    type: LORAWAN_SOURCE
    configuration:
      networkStack: TheThingsStack
      ## set url to MQTT endpoint of networkstack
      url: eu1.cloud.thethings.network:1883
      userName: <username>@ttn
      password: <password>
      appId: <app-id>
      ## Files needed for secure mqtt connection
      CAPath: /Users/user/cert/
      certPath: /Users/user/cert/cert.pem
      keyPath: /Users/user/cert/key.pem
      ## configure known devices
      deviceEUIs:
        - eui-70b3d57ed005e88a
      ## configure which fields the sensors from \terra{} corresponds to in the schema
      sensorFields:
        - temperature
        - acceleration
```

Listing 1: Configuration of a LoRaWAN physical source in a NebulaStream worker

4.5.3 MicroPython Proof of Concept

As a proof of concept, the first Terra version was developed in MicroPython[58] on a Pycom FiPy board. It follows the design specified in section 4.4, and serves to verify its feasibility. However, since MicroPython runs on a limited number of platforms and is a managed language, it is hard to port and control the exact low-level behaviour. For that reason, the MicroPython project was ported to C and expanded upon.

4.5.4 C Port

Here we present Terra's implementation. At first, we need an OS for Terra to be implemented in. That is why we start with a discussion on the choice of operating system before we continue with the implementation details.

But before that, we dwell on a couple of global design choices. As we want to release Terra under an open source licence, the libraries and frameworks we include should be compatible. Terra itself should only use static allocations. This is both to ensure broad compatibility and also to ensure easy verifiability and reasoning of resource usage. As suggested in section 4.3.1 it would make sense to reuse the protocol buffer representation as much as possible to prevent unneeded conversion between types, and higher memory requirements.

OS Choice

Requirements **NF1** to **NF4** and **NF6** require operating system support. These are essential for Terra, so choosing an appropriate OS requires careful selection. There are several works comparing operating systems for IoT devices and their intricacies[67, 68]. In this thesis, we will not go through all the operating systems currently available but focus solely on the most prominent ones: TinyOS, RIOT, Zephyr and FreeRTOS.

TinyOS[12] TinyOS was previously discussed in chapter 2, where we noted its use of its own variant of C and its restricted hardware compatibility. This event-driven, nonpreemptive, monolithic operating system lacks LoRaWAN support. Consequently, despite its presence in the Sensor Network Database arena, we find TinyOS to be unsuitable for our needs.

FreeRTOS[69] An operating system mainly developed by Amazon but still released under an open MIT licence. FreeRTOS is more of a scheduler than an operating system[70]. It contains no hardware abstraction layer making portable programming cumbersome [67]. This makes it difficult to live up to requirement **NF1**. However, there are plenty of resources on porting FreeRTOS to other platforms as that is unavoidable due to the lack of the abstraction layer. Additionally FreeRTOS does cater towards low power use-cases by allowing the developer to disable the normal clock tick that are otherwise used to regularly check for, and schedule, higher priority task. By disabling this tick or running in tick-less mode, FreeRTOS can go into deep sleep for longer periods of time to conserve energy. However, the lack of a hardware abstraction layer makes this OS a nonstarter, as portability is a key design constraint of Terra.

Zephyr[71] While Zephyr is not included in the above-referenced survey papers, we include it here, as it is the largest IoT operation system by number of collaborators and commits. It is a highly customisable monolithic and modular OS with its own development tools. It is backed by the Linux

Foundation and has several prominent supporters, including Google, Intel, Meta, and others. It enjoys wide hardware support with over 750 supported boards and over 220 supported sensors.

RIOT[70] RIOT, much like Zephyr, is a highly customisable and modular monolithic kernel. Unlike zephyr its build system is built around a recursive tree of Makefiles making it easy to add and modify. Its a popular choice for prototyping and academia because of its open and customisable nature. RIOT supports around 280 boards[72] and hundreds of sensors, and due to its modular system, it is quite easy to add support for new boards and sensors.

Conclusion The choice of OS falls between Zephyr and RIOT. Both operating systems are similar in nature, while one study of performance gives the performance advantage to RIOT versus Zephyr OS [73], the choice of OS is much more dependent on familiarity and developer satisfaction, which boils down to personal preference. Given the author's limited experience with both options, RIOT was selected because its build system utilises makefiles. Although these can be quirky, the author is more accustomed to them compared to Zephyr's internal build framework. Additionally, RIOT was the operating system used in the bachelor project responsible for the C port mentioned above, eliminating the need to port the application code. Since RIOT and Zephyr are quite similar, it should not be difficult to port the TERRA application between the two.

Protocol Buffers

For Protocol Buffer support, we use the `nanopb` library [74]. It is a C implementation of the protocol buffer specification, optimised for resource-constrained hardware. In addition to the library being optimised in general for code size instead of performance, it also includes specialised configuration options to guide the generation of protocol buffer C code. In particular, the opportunity to define maximum counts of `repeated` fields, which causes them to be statically allocated. We do this in a separate `.options` file, and the options, including their defaults, can be seen in listing 2[4].

This requires that all structs fit within the RAM of the device, but allows for a per-device optimisation.

[4]Listing shows an extract. Full file can be seen at `https://github.com/FlapKap/Terra/blob/master/app/terra/terraprotocol.options`

```
TerraProtocol.Output.responses max_count:16
TerraProtocol.Expression.instructions max_count:50
TerraProtocol.Query.operations max_count:2
TerraProtocol.Message.queries max_count:1
```

Listing 2: Options for protocol buffer C code generation through `nanopb`

```c
typedef struct _WindowData {
    Number aggregation_value;
    Number start_value;
    Number end_value;
    uint32_t count;
    windowState state;
} WindowData;

typedef struct _TerraConfiguration {
    uint32_t loop_counter;
    uint8_t raw_message_size;
    uint8_t raw_message_buffer[LORAWAN_APP_DATA_MAX_SIZE];
    WindowData window_data[MAX_WINDOW_OPERATORS];
} TerraConfiguration;

bool configuration_save( TerraConfiguration* config, semtech_loramac_t* loramac_config,
↪  bool save_query);
bool configuration_load( TerraConfiguration* config, semtech_loramac_t* loramac_config );
```

Listing 3: Extract of `configuration.h` showing relevant structs and functions

The structs generated by `nanopb` are what we use for the internal query representation. Since everything is statically allocated, this means that no additional decoding is needed after the initial deserialisation of the serial format. It also means the structs generated by `nanopb` accommodate the largest possible messages, which will take up more memory than needed most of the time.

Configuration & State Management

Here, we describe how configuration and state management is implemented in Terra. We need this since Terra is designed to maximise power efficiency, and between executions it will go into the lowest possible power state, which is highly likely to clear RAM or other ephemeral storage. We first describe the configuration and storage handling in Terra and then address the choice of persistent storage media.

In TERRA we define a struct containing all TERRA relevant configuration, seen in listing 3. It contains a loop counter, the serialised query, if any, and any window state.

The `WindowData` struct contains the information required by the design described in section 4.4. We save the query in its serialised format since this is much smaller. With the large deserialised format, loading and saving could become quite costly depending on the storage medium.

Saving and loading of the configuration is achieved through two functions whose signature can be seen in listing 3. TERRA supports multiple different storage mediums which are exposed through RIOT. The appropriate storage medium is selected through preprocessor macros and this defines the appropriate functions. TERRA supports:

- battery backed RAM for the ESP 32 family of processors.
- EEPROM.
- EEPROM through the EEPREG[5] module if used. This is simply a module to manage multiple configurations stored in EEPROM.
- Internal flash if the device supports the RIOT flashpage interface[6].

We will not go through the implementation here, but we simply point to the `configuration.c`[7] file for details.

In addition to the state required by TERRA, RIOT itself needs to store the network state in deep sleeps. This is done in the `semtech-loramac` RIOT module, which we will discuss in the network section below, but this is why a reference to this struct is included in the function calls.

As a small optimisation we only save the query if marked, to ensure that it is only written to persistent storage once, and then when it changes. This is both to save energy, but also to preserve flash devices since they have limited number of writes over their life-time.

We note that some small contributions to the RIOT codebase were made by the author on the path to implementing the above, mainly concerning bugs in the RIOT codebase or adding EEPREG support for the `semtech-loramac` module[8].

[5]RIOT [75, modules/system/eeprom registration]

[6]RIOT [75, modules/drivers/Storage Device Drivers/MTD wrapper for Flashpage devices]

[7]`https://github.com/FlapKap/Terra/blob/master/app/terra/configuration.c`

[8]This pull request is awaiting approval but is utilised by TERRA on supported boards

```
SENSOR_NAMES ?= hts221 hts221
SENSOR_TYPES ?= SAUL_SENSE_TEMP SAUL_SENSE_HUM
```

Listing 4: Example of sensor configuration in the Terra `makefile`

Sensing

For the configuring of sensors, we use the RIOT SAUL system [70]. With a little makefile magic[9], we can then configure the sensors in our makefile by specifying their name and type, as done in listing 4, and RIOT will automatically initialise and configure them. Terra will then use these sensors, in that order, to get data readings and copy them into the environment.

Tensorflow Lite Model

As written in section 4.4, the TFLite model will exist as a pseudosensor dependent on the sensor values being available, but executed before queries. We update and use the Tensorflow Lite for Microcontrollers library[76] in RIOT to load and execute models. Since this library uses C++, we need to include this as a feature and then include the `tflite-micro` package. To separate the TFLite model from the rest of Terra, this is implemented as a module we import. This module contains a C/C++ header declaring an initialisation and execution function for the model, while the corresponding `.cpp` file contains the implementation that defines the TFLite operators used and, when running, copies the environment into the model, and the results out of the model into the environment. In the module `Makefile` we utilise the RIOT `utils_blob` module to import the TFLite binary as a C `char` array in a C header.

In Terra, the inclusion of a TFLite model is defined by whether or not this module is included in the Terra `Makefile`, via e.g. `USEMODULE += tflite_model`. If it is, then the module is built together with Terra and code is included that executes the model after sensors are read but before the query is executed. We provide an example model and code for the `tflite_model` module. The model used in this example is provided by the EU ELEGANT project[10] and is a simple fully connected neural network

[9]Converting the Makefile definitions into the sensor handling logic and the makefile driver definitions. We refer to the makefile for the full implementation https://github.com/FlapKap/Terra/blob/master/app/terra/Makefile

[10]https://www.elegant-h2020.eu/

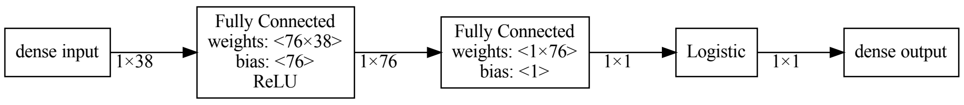

Figure 4.6: Tensorflow Lite network used as an example in TERRA

designed to detect water leaks. The purpose is not important here, as the goal is just to illustrate how to implement a TFLite model in TERRA. An illustration of the network of the model can be seen in fig. 4.6

Networking

RIOT offers several options for managing networking. In particular, it offers the Generic Network Stack (GNRC) [77] as a common clean API to multiple network back-ends. Enabling or disabling support for different network stacks is done by the simple inclusion of modules in the Makefile and its use would make it very easy to de-couple the network implementation from the rest of TERRA. However, the GNRC LoRaWAN module is still in development and does not support LoRa regions outside EU, or adaptive data rates.

That is why we chose to go with the alternative: the `semtech-loramac` module[11] in RIOT, which is an adaptation of the reference implementation. This does support the features mentioned and has built-in support for saving network configuration and state to EEPROM if available. However, it does not offer the same decoupling of network interface and implementation. To decouple TERRA from `semtech-loramac`, we implement a network layer in between. It simply contains functions to initialise, receive, and send messages, and send heartbeats. We then separately declare and define the LoRaWAN specific logic in its own header and implementation files. Changing the network implementation is then simply a question of implementing the same network header, to a new network stack - Possibly even GNRC.

However, the current solution has some drawbacks. In the main loop of TERRA we use heartbeats to make sure that a receive window opens to receive messages. We do this when no query is currently running, and dependent on the value set in the `FORCED_LISTEN_EVERY_N_LOOP` option.

[11]RIOT [75, modules/packages/Semtech LoRaMAC implementation]

This option is there to keep TERRA responsive to new queries when running long windows or very selective filters. This is LoRaWAN specific behaviour, which should be decoupled. The `semtech-loramac` module handles saving to persistent storage - but only if EEPROM is available. So for other storage backends, we have to implement this ourselves. The `semtech-loramac` module is also quite outdated in RIOT, which means that even though latest LoRaWAN standard is 1.1[78] and 1.0.4[36], we can only support 1.0.2[40] in TERRA.

Ideally, GNRC would have been in a state where LoRaWAN would be well supported, so this system could be used to choose network stack without modification of TERRA code. However, heartbeats would still have to be part of TERRA logic, and GNRC does not handle persisting configuration and network state, leaving this for TERRA to manage.

Some information need to be provided to TERRA to support networking:

`LORAMAC_DATA_RATE`: Which baseline data rate to use.

`CONFIG_LORAMAC_DEFAULT_ADR`: If this flag is present Adaptive Data Rate is enabled.

`LORA_REGION`: Which region we operate in.

`APPEUI`: The Application Extended Unique Identifier.

`DEVEUI`: The Device Extended Unique Identifier.

`APPKEY`: The Application key.

In addition to these, we will in chapter 5 modify the following options compared to their defaults in `semtech-loramac`. Mainly to support a specific LoRaWAN provider and to improve reception. We mention them here for completion.

`LORAMAC_DEFAULT_RX1_DELAY`: The default delay after sending until the first receive window.

`LORAMAC_DEFAULT_RX2_DR`: The default data rate for the second receive window.

`LORAMAC_DEFAULT_SYSTEM_MAX_RX_ERROR`: To increase the receive window widths to account for timing errors.

We refer to section 2.2 for more details on all of these parameters.

Numbers

To ease implementation, we define a `Number` struct that contains all types of numbers that TERRA supports, in a union. We support unsigned and signed integers, floats, and doubles. We have made a choice to support these types of numbers, since this provides a nice compromise between the expressibility of numbers and the combinatorial explosion of handling operations between more types.

This allows us to propagate numbers throughout TERRA without worrying about type. Alternatively, we could have represented numbers in their natural type, which would lead to a lot of different environments and tracking, or we could cast each number to the largest type, which would rule out some compiler optimisations.

We implement functions for comparison, truth checking, and arithmetic operations on numbers. These are mainly used for windows and other areas where computation is not done through an expression.

Expressions

An expression in TERRA consists of a stack, a programme counter, and a list of instructions which are of the type Data. The Data type contains either an expression instruction or a number. The stack is simply a stack of the Number struct. To execute an instruction, TERRA simply reads the list from start to finish, and for each expression the instruction executes the relevant function. All of this follows the description in section 4.4.

An implementation detail of note is that since Numbers in TERRA are slightly different from the numbers in the protocol buffer format, we have to copy between these two formats when dealing with `CONST` and `VAR` instructions. These expect the next instruction to be a number that indicates either just a scalar value to be pushed to the stack or the index of the environment they read from.

All instructions are implemented through small functions and a central switch statement, so implementing new instructions only requires adding it to the protocol buffer format and implementing a function and adding it to the switch statement.

The Environment

The environment is the global array in which sensor reads and operator results are stored. Since a result from TERRA comes from queries, we tag any value pushed to the environment with its source; either sensor or query, and any results transmitted contain only values of query source. We do this since we want operators to be able to operate both directly on sensor values and also on earlier operator values. Otherwise, Terra could not filter based on a map operation value. On the other hand, we do not want Terra to submit raw sensor values if they are not specifically requested by an operation. So we need operators to be able to access both sensor and query values while only submitting query values as a result.

Execution of Operators

Operators are executed sequentially, with the stack cleared in between them, and the environment cleared after each query. Map operators execute as described in the design. So does filter operators. If the value left on the stack after a filter operation is false, the whole execution of the query is halted and no result is transmitted.

For window operators, only timing-based tumbling windows are currently supported. This is implemented as described in the design. Notable is the handling of time. We measure the timing in the window based on number of executions times the length of the window. Since we know with which interval Terra executions run and how many times a window has executed, by its `count` property, we can derive the running time.

We do it this way to simplify time management, to be centralised around the epoch time. Currently Terra uses a real-time clock for that, but one could easily imagine that hardware is not available and Terra has to resolve to other means of time keeping.

Terra Main Logic

The Terra `main` function essentially follows fig. 4.2 and orchestrates all different parts of Terra. It times the execution of all these parts to calculate the appropriate sleep duration to maintain an epoch time as defined by the `EXECUTION_EPOCH_S` macro. This macro defines how often a Terra epoch is run. It also prepares the device for sleep and ensures it wakes up properly by disabling peripherals and defining wake-up alarms.

We also allow the user to pre-load a query onto Terra at build time. This is defined as a C formatted serialized protocol buffer `char` array through the option `DEFAULT_QUERY_AS_PB_CHAR_ARRAY`. We also allow the user to provide this in a base 64 format through the option `DEFAULT_QUERY_AS_PB_BASE64` which will then on build set the `DEFAULT_QUERY_AS_PB_CHAR_ARRAY` to the corresponding value.

We also allow for changing the size of the expression stack through the `RUNTIME_STACK_MEMORY` option.

RIOT specific Build Options

In addition to the configuration options listed in the previous sections,

WE also want to mention the below RIOT configuration options, since they are either relevant for TERRA or we modify them in later sections.

DEVELHELP: 1
Enables a wide variety of development helpers, like more informative error and debug messages. Also enables asserts[75, see modules/utilities].

MAIN_STACKSIZE: 5120
The main stack size of RIOT OS in bytes. Normally this is specifically defined for different CPU architectures, but we fix it here to accommodate TERRA.

Overview

Finally as a summary of this section we provide a table of all configuration options discussed in table 4.2. This table is divided into sections according to the sections in which the configuration options were introduced. This also roughly corresponds to how and where the configuration options should be defined. For the **Protocol Buffers** options, they should all be defined in the accompanying `terraprotocol.options` file. The rest are `Makefile` options and as such should be defined in the application Makefile, but can also be overwritten through `make` command-line variables.

4.6 Future work

TERRA in its current state fulfils almost all the requirements listed in section 4.1, with the exception of sliding windows in requirement F4.3. Properly implementing that is an obvious future goal.

In addition to that, there are some possible optimisations. The first one is to implement acquisitional sensing where sensor reads are based on the current running query such that only if a sensor value is used that sensor is

Configuration Option	Description	Default	Options
Protocol Buffers - `terraprotocol.options` All options prefixed with `TerraProtocol.`			
`Output.responses max_count:`	Max # of scalar responses	16	Any positive integer
`Expression.instructions max_count:`	Max # of query instructions	50	Any positive integer
`Query.operations max_count:`	Max # of query operations	2	Any positive integer
`Message.queries max_count:`	Max # of concurrent queries	1	Any positive integer
Configuration & State Management			
`LORAWAN_APP_DATA_MAX_SIZE`	Max size of LoRaWAN packet (bytes). Limits the size of a query	242	Any positive integer
`MAX_WINDOW_OPERATORS`	# of concurrent windows	5	Any positive integer
Sensing			
`SENSOR_NAMES`	SAUL sensor names to load	not set	any SAUL sensor name in RIOT
`SENSOR_TYPES`	SAUL sensor types to load	not set	any SAUL sensor type in RIOT
TFLite model			
`tflite_model`	If module is present, TFLite model will be included	n/a	n/a
Networking			
`LORAMAC_DATA_RATE`	Baseline data rate	not set	DR0-DR6
`CONFIG_LORAMAC_DEFAULT_ADR`	If flag present, enable ADR	not present	not present/present
`LORA_REGION`	LoRaWAN Region	not set	See [39] for entries
`APPEUI`	Application Extended Unique Identifier	not set	n/a
`DEVEUI`	Device Extended Unique Identifier	not set	n/a
`APPKEY`	Application Key	not set	n/a
`LORAMAC_DEFAULT_RX1_DELAY`	Delay to first RX window in ms	1000	Any positive integer
`LORAMAC_DEFAULT_RX2_DR`	Data rate for second RX window	region dependent	`LORAMAC_DR_0-3`
`LORAMAC_DEFAULT_SYSTEM _MAX_RX_ERROR`	Measure of how much timing error to account for	50	Any positive integer
`FORCED_LISTEN_EVERY_N_LOOP`	Send a heartbeat every N epoch	1000	Any positive integer
Terra main logic			
`DEFAULT_QUERY_AS_PB_CHAR _ARRAY`	A protocol buffer serialized default query formatted as a C `char` array	not set	any valid PB query
`DEFAULT_QUERY_AS_PB_BASE64`	A protocol buffer serialized default query formatted in base 64	not set	any valid PB query
`RUNTIME_STACK_MEMORY`	Size of the expression stack	10	Any positive integer
`EXECUTION_EPOCH_S`	Terra execution epoch in seconds	120	Any positive integer
RIOT specific build options			
`DEVELHELP`	Development helpers: 1 to enable, 0 to disable.	1	1 or 0
`MAIN_STACKSIZE`	RIOT OS stack size	5120	Any positive integer

Table 4.2: Summary of configuration options relevant for Terra

read. The next step from that would be to fully embrace acquisitional query processing, where the optimiser chooses not only which sensor is sensed but also when and how often. This requires the support of NebulaStream, which already includes adaptive sampling rates and acquisitional query processing as part of its vision [2].

Improving network support and handling is also a big future goal. At the moment, TERRA is somewhat coupled with LoRaWAN, which should be reduced. For this there are multiple avenues available. Work could be done to utilise the GNRC networking system of RIOT to increase the network agnosticism of TERRA and enable the built-in support for a variety of physical layer/link layer technologies such as IEEE 802.15.4 or Ethernet [77]. This would require a development of the GNRC LoRaWAN support. However, support for network stacks not supported by GNRC, such as Bluetooth LE, would still require custom handling.

Network resiliency is also an area of research that can be investigated. Work is already being done to improve the network resiliency of NebulaStream [79], and some of the techniques used could be ported to TERRA.

4.7 Summary

In this chapter, we introduce TERRA, our system that augments NebulaStream with code offloading to sensor nodes. We exploit the already existing infrastructure of NebulaStream to parse and interpret a query, and then hook into this infrastructure to selectively push down partial queries to TERRA. We analyse and discuss different avenues of opportunities we have for designing and implementing TERRA given the requirements given. We also discuss the nature of the integration of TERRA into NebulaStream. In table 4.3, we provide a summary of the design choices made in TERRA and their alternatives. The chosen option is shown in **bold**.

In the next chapter, we define a cost model for TERRA, which we use to guide our evaluation. Next we will define the experimental framework which we follow to obtain our results, which we then discuss.

Aspect	**Design Choice**
Network Technology	**LoRaWAN** SigFox NB-IoT
Serial Format	**Protocol Buffers** Flat Buffers Apache Thrift
Query Execution	Cross compiled machine code Intermediate representation executed by a Virtual Machine Parametrized functions **Parametrized functions with intermediate representation of expressions**
Integration	Deep integration - Introduce pseudo-worker to NebulaStream **Shallow integration** - Represent as Data Source and hook into Node Engine

Table 4.3: Summary of design choices in Terra

Chapter 5

Cost Model

5.1 Introduction

To answer our hypothesis and research question **R3** we need to evaluate the energy cost of code offloading. More specifically, we need to compare the energy cost of offloading the code to the sensor device versus the energy savings of not transferring raw sensor reads. We do this by developing and deriving an energy cost model through a series of experiments where we evaluate the energy consumption of the various activities that TERRA goes through as part of its execution. That model will allow us to compare the different energy costs of queries under varying conditions and shed light on how code offloading affects the energy consumption of sensor devices. Such a cost model could also enable NebulaStream to intelligently decide whether or not a query push down is worth it, since it describes TERRA's energy consumption when a query is provided, and when one is not provided. This will enable NebulaStream to optimise query placement given a query's characteristics and life-time.

In the rest of this chapter, we will go through the formulation of this cost model, the experimental framework that derives it, and the results from those experiments. We will then show the performance of the model.

5.2 Energy Cost Model

Here we describe the energy cost model we later experimentally derive. We want the energy model to show the energy consumption of TERRA given a query, its response rate, and whether or not it runs a TFLite model.

	Query Length		Response Rate		TFLite	
Activity	Startup	Steady	Startup	Steady	Startup	Steady
0. RiotInit						
1. LoadConfig		•				
2. Deserialize	n/a	•				
3. SensorInit	n/a		n/a		n/a	
4. NetInit						
5. SensorCollect	n/a		n/a		n/a	
6. ExecTFLite	n/a		n/a		n/a	•
7. ExecQuery	n/a	•	n/a		n/a	
8. SendReceive	•			•		
9. SaveConfig	•					

Table 5.1: Activities and their expected dependency on Query Length, Response Rate or TFLite in either the Start-up or Steady State phase. · indicates a dependency. "n/a" indicates the activity is not executed in the given phase.

Terra supports the execution of a series of activities that take place at some given frequency. These activities consume some amount of power. Some activities are dependent on user-dependent factors, while others are influenced by environmental factors. There are also activities that are not dependent on any external factors at all. Terra's behaviour is also dependent on the presence of a query. If there is no query present, most activities are not executed and some activities have different power usages.

We describe this difference by defining whether or not an activity takes place during a **startup** epoch or a **steady-state** epoch, with the startup epoch being the first execution of Terra, as this generally is the execution where no query is present and one will be fetched from the network.

Then, for each activity, we describe its dependency and define a model based on this. Specifically, we look at what is done during each activity and judge if energy consumed would correlate with Query Length (QL) or the presence of a TFLite model (TF). If it does, we model that specific activity/parameter relationship as a linear model. If not, it is simply an experimentally derived constant and we define it as such. We do this for both the start-up epoch and steady-state epochs.

Response Rate (RR) is a special case, as that is neither a linear relation nor a constant. Its costs vary from epoch to epoch. We will discuss this in detail in section 5.2.3.

Each activity and their parameter dependencies can be seen in table 5.1. As an example, we see that the "8. SendReceive" activity has marked dependency with query length in the startup phase and response rate in the steady-state phase. This activity covers network communication, and as such it makes sense that the consumption of it is affected by the length of the query it receives and how often it needs to transfer results.

Table 5.1 gives us a series of constants and models, which we present below. The models are named with the subscript $f_{\text{phase,activity}}$ where phase is 0 for the start-up phase and 1 for the steady-state phase. Activity corresponds to the activity number given in table 5.1. Constants follow the same naming scheme. Cells labelled "n/a" indicate that the activity is not carried out during the specified phase.

We start by defining the constants in section 5.2.1. Then we define the linear models correlated with query length in section 5.2.2. After this we define the model for response rate in section 5.2.3, and then the model for TFlite in section 5.2.4.

Finally in section 5.2.5, we gather all the individual models described in the following in a model that then describes the energy consumption of a given epoch under a given QL, RR, and TF. To aid in comparison, we also define a baseline model without the cost of code offloading.

5.2.1 Constants

As can be seen in table 5.1 there are many activities that do not depend on any of the workload parameters. These are modelled as constants and are derived from taking the average of the measured values. The constants are $C_{0,0}$, $C_{1,0}$, $C_{0,1}$, $C_{1,3}$, $C_{0,4}$, $C_{1,4}$, $C_{1,5}$, $C_{1,9}$. Note that there are no $C_{0,2}$, $C_{0,3}$, $C_{0,5}$, $C_{0,6}$, $C_{0,7}$ as a query is never deserialised and executed in the start-up phase.

5.2.2 Query Length

Query Length is the parameter that affects most activities in Terra, from loading the query from memory to executing it and saving it. The query length is its serialised size in bytes, as this directly affects storing, loading, and deserialising. It also serves as an indirect measure of its complexity. It goes from a minimum query size of around 16 B up to the maximum supported by Terra, which ranges from 50 B to 222 B dependent on data rate.

The length of the query will have an impact on the startup cost of the `sendReceive` activity, as this is where the query is received. It will also affect the `save` activity, and the steady state cost of loading, deserialising and executing:

$$f_{0,8}(\mathrm{QL}) = \alpha_{0,8} + \beta_{0,8}\mathrm{QL} \tag{5.1}$$
$$f_{0,9}(\mathrm{QL}) = \alpha_{0,9} + \beta_{0,9}\mathrm{QL} \tag{5.2}$$
$$f_{1,1}(\mathrm{QL}) = \alpha_{1,1} + \beta_{1,1}\mathrm{QL} \tag{5.3}$$
$$f_{1,2}(\mathrm{QL}) = \alpha_{1,2} + \beta_{1,2}\mathrm{QL} \tag{5.4}$$
$$f_{1,7}(\mathrm{QL}) = \alpha_{1,7} + \beta_{1,7}\mathrm{QL} \tag{5.5}$$

5.2.3 Response Rate

An epoch will only use energy to transmit a response if it generates a response. This behaviour does not fit a linear relationship between response rate and energy usage, as the energy usage comes in spikes. Instead, we amortise the cost of the transmission over the epochs. A response will have some cost $C_{1,8}$ when sent, but that will only occur after m epochs. We thus define our response rate as the reciprocal of m:

$$\mathrm{RR} = \frac{1}{m} \tag{5.6}$$

This allows us to multiply the fixed transmission cost $C_{1,8}$ with RR to produce the amortised cost of an epoch for a given response rate:

$$f_{1,8}(\mathrm{RR}) = C_{1,8}\mathrm{RR} \tag{5.7}$$

This means that, if a response is only sent for every 4^{th} epoch, the response rate would be $1/4$ and the cost of transmission would for each epoch be $f_{1,8}(1/4) = 1/4 C_{1,8}$.

5.2.4 Tensorflow Lite

TERRA includes the option to run a TFLite model on the sensor input before query execution as a sort of pseudo-sensor. This entails a significant increase in compute time. However, since TFLite models are included on build-time and not provided during run-time like a query, the consumption of the model is constant for any given deployment of TERRA. That is why we model it here as as simple binary dependency.

$$f_{1,6}(\mathrm{TF}) = C_{1,6}\mathrm{TF} \tag{5.8}$$

Where TF $= 1$ if there is a TFLite model present and TF $= 0$ if there is none.

5.2.5 Final Model

Now we produce the final model. First, we gather all constants into two variables, C_0, C_1 to simplify.

$$C_0 = C_{0,0} + C_{0,1} + C_{0,4} \tag{5.9}$$
$$C_1 = C_{1,0} + C_{1,3} + C_{1,4} + C_{1,5} + C_{1,9} \tag{5.10}$$

Then we produce two functions: one for the startup epochs f_0 and one for the steady-state epochs f_1.

$$\begin{aligned} f_0(\mathrm{QL}) &= C_0 + f_{0,8}(\mathrm{QL}) + f_{0,9}(\mathrm{QL}) \\ f_1(\mathrm{QL}, \mathrm{RR}, \mathrm{TF}) &= C_1 + f_{1,1}(\mathrm{QL}) + f_{1,2}(\mathrm{QL}) + f_{1,6}(\mathrm{TF}) \\ &\quad + f_{1,7}(\mathrm{QL}) + f_{1,8}(\mathrm{RR}) \end{aligned} \tag{5.11}$$

We can then describe the energy consumption of an epoch using the piecewise function f:

$$f(\mathrm{QL}, \mathrm{RR}, \mathrm{TF}, n) = \begin{cases} f_0(\mathrm{QL}) & \text{if } n = 0 \\ f_1(\mathrm{QL}, \mathrm{RR}, \mathrm{TF}) & \text{if } n > 0 \end{cases} \tag{5.12}$$

Where n is epoch number.

To get the total energy consumption of some query over n epochs, we sum over f from 0 to n.

$$f_{total}(\mathrm{QL}, \mathrm{RR}, \mathrm{TF}) = \sum_{i=0}^{n} f(\mathrm{QL}, \mathrm{RR}, \mathrm{TF}, i) \tag{5.13}$$

Baseline

We also produce a baseline model that represents TERRA's energy cost when no query is provided, and it just transfers the sensor measurements directly. This model is almost identical to f_{total}, except that it does not contain $f_{0,8}$ in f_0. There is no transfer of query done in the startup phase. Specifically:

$$f_{0_{baseline}}(\mathrm{QL}) = C_0 + f_{0,9}(\mathrm{QL}) \tag{5.14}$$

$$f_{baseline}(\mathrm{QL}, \mathrm{RR}, \mathrm{TF}, n) = \begin{cases} f0_{baseline}(\mathrm{QL}) & \text{if } n = 0 \\ f_1(\mathrm{QL}, \mathrm{RR}, \mathrm{TF}) & \text{if } n > 0 \end{cases} \tag{5.15}$$

$$f_{baseline\_total}(\mathrm{QL}, \mathrm{RR}, \mathrm{TF}) = \sum_{i=0}^{n} f_{baseline}(\mathrm{QL}, \mathrm{RR}, \mathrm{TF}, i) \tag{5.16}$$

5.2.6 Assumptions and Trade-offs

Here we briefly discuss the assumptions and trade-offs made with this energy model. We note that the variables of the model are all based on the energy consumption of TERRA. However, these are naturally dependent on the device TERRA runs on, and as such the model is only applicable in absolute terms on the hardware chosen. That is, as a tool for NebulaStream to estimate the absolute power consumption, the model will only work on the chosen hardware. This includes not only the choice of board, but also sensor and peripherals. Ultimately, that is a challenge for all energy cost optimisation. In TinyDB they solved this by hardcoding the consumption of its activities for each sensor node [9]. However, the model we develop is not tied to any specific hardware features or peculiarities in its design and makes very few implicit assumptions about the system it is modelling. In general, we assume that the consumption of power within activities follows just the relations we set up. For example, there is no modelling of power states and utilisation of peripherals. Instead, all of this is contained within the variables.

We also assume that sensors do not vary the amount of data they produce over time. Data volume is constant, which means that the consumption per epoch is constant. An optimisation in TERRA could be to only sense data that are used in queries but that is not currently done in TERRA, see implementation details in section 4.5. Then energy consumption would depend on the amount of sensors used in the given query. Additionally, the expression language reads at most one sensor value per instruction. This means that a query working with many sensor values would also be very long which would be captured in the QL parameter. The consequence of this is that the volume of data is not a factor in the model, as it is either constant or covered by the QL parameter.

We also do not model sleep at all, but only the consumption during TERRA execution. The reasons for this are two-fold:

- Although power consumption during sleep can be assumed to be constant, sleep duration is influenced by both epoch and execution times, creating a complex interaction with other model variables. Consequently, epoch time would need to be an integral part of the model, adding to its complexity.
- TERRA cannot affect the power consumption of sleep at all, as that is purely dictated by the lowest available power state on the microcontroller as exposed through RIOT.

The outcome of this is that the model is very general, and just need to have its variables derived for other hardware to be applicable. It does come with the trade-off that the model is not that granular and might not reflect the true energy cost of the chosen microcontrollers.

5.3 Experimental Framework

In this section we describe the setup of the experiments to determine the values of the variables in the model described in the previous section. Our framework follows the procedure described in Jain [41, ch. 2]. We do that by describing:

1. The goal of the experiments and the system under test, including its parameters.
2. The metrics we will use.
3. The workload we will execute on the chosen test bed.
4. The experiments we will execute.

For accuracy, these experiments are run on real-world devices in a set-up not dissimilar to how TERRA would be used. The code defining and running experiments is publicly available[1].

Setting up large-scale experiments with power consumption measurements at the required sampling rate is not easy. Luckily, as described in section 2.3, there is a test bed for embedded device testing in Europe which allows us to conduct our experiments on multiple devices.

We use IoT-LAB and describe it briefly in the next section 5.3.1. After that we go through the many parameters of the system that we keep constant

[1] `https://github.com/FlapKap/Terra-experiments`

in section 5.3.2, followed by the metrics that we vary in section 5.3.3. Then we cover the workload we execute in section 5.3.4 and finally the actual experiments in section 5.3.5.

5.3.1 Testbed

Testing the energy consumption of embedded devices can be challenging as it requires specific equipment and infrastructure attached to the system under test. Fortunately, The Future Internet Facility in France provides the IoT-LAB described in section 2.3 that supports a wide variety of nodes and network technologies. Through their stack, we can set up experiments with tens of nodes connected through LoRaWAN where it is possible to schedule, run and collect data from firmware build for their supported hardware. IoT-LAB connects the board under test (called ”Open node” in their terminology) to a gateway board (not to be confused with a LoRaWAN gateway) that manages the programming of the board while a control node monitors radio, traffic, and power consumption [43, getting-started/design].

5.3.2 System

The purpose of the experiments is to model the energy consumption of Terra according to the cost model defined in section 5.2. Queries are sent to Terra from a client computer through the LoRaWAN network, and as such we include the network to be part of the system. The influence of external elements on the system will be reduced. An illustration of the system can be seen in fig. 5.1. Note that we do not include NebulaStream to be part of the system. While Terra is integrating with NebulaStream, it is not part of the system under test, and the user in this case refers to the communication done using the protocol described in section 4.5.1.

Next, we review the system parameters. These are all the variables that could affect the performance of the system under test, but notably *not* variables we vary. That includes almost all the options listed in table 4.2, but not options concerning TFLite and default queries. The system parameters also include the hardware chosen for all parts of the system. We cover the system parameters in the following order:

1. Terra hardware Parameters.
2. Network Stack Parameters.
3. Implementation parameters as shown in table 4.2.

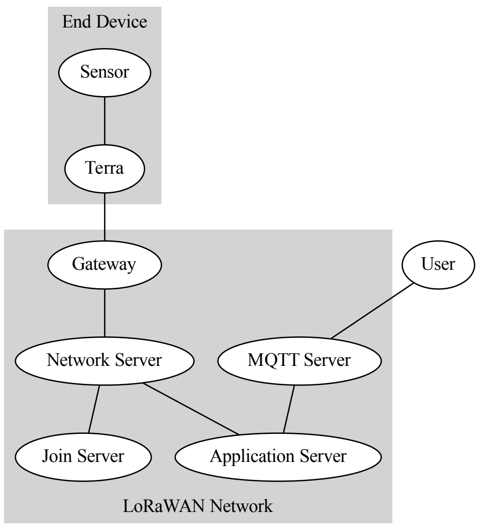

Figure 5.1: System under test

We follow this structure since the network stack and the hardware parameters are limited by the testbed, and the implementation parameters are limited by the chosen hardware.

Terra Hardware

Here we go through the hardware chosen for the experiments. Since we chose to use IoT-LAB to perform power measurement tests, we are limited by the hardware available on their testbed. They provide four boards with LoRa connectivity, where only three of them support uploading your own firmware. We list them below together with the number of boards available, and any attached sensors:

Microchip SAMR34 Xplained Pro IoT-LAB provides two boards at their Grenoble site. There are no sensors attached, so any sensor reads would have to be provided by built-in sensors.

Nucleo-WL55JC IoT-LAB provides five boards at the Grenoble site. These are equipped with the `ST X-NUCLEO-IKS01A2` sensor shield that contains four sensors:

HTS221 a temperature and humidity sensor.

LPS22HB an atmospheric pressure sensor.

LSM6DSL an accelerometer and gyroscope sensor.

LSM303AGR an accelerometer and magnetometer sensor.

ST B-L072Z-LRWAN1 IoT-LAB provides twenty boards at their Saclay site and five boards at their Lille site. Most of these nodes, all the Saclay nodes, and two of the Lille nodes, are also provided with the `ST X-NUCLEO-IKS01A2` mentioned above. The three other Lille nodes are attached to a `ST X-NUCLEO-IKS01A3` sensor shield which contains the following sensors:

HTS221 a temperature and humidity sensor.

STTS751 a temperature sensor.

LPS22HH an atmospheric pressure sensor.

LSM6DSO a 3-axis accelerometer and 3-axis gyroscope sensor.

LIS2MDL a 3-axis magnetometer sensor.

LIS2DW12 a 3-axis accelerometer sensor.

To support easy activity tracking and consistent power measurements, we choose the same board for all devices during an experiment, and so to enable experiments with more than five devices we choose the `ST B-L072Z-LRWAN1`. Riot contains drivers for `HTS221`, `LPS22HB` and `LSM303AGR`, so we choose boards that have the `ST X-NUCLEO-IKS01A2` sensor shield attached. For simplicity and ease of integration into Terra, we only use `HTS221` which has two outputs: humidity and temperature.

Network Stack Parameters

Above we limited ourselves to the Saclay and Lille sites of IoT-LAB. At the Saclay site, IoT-LAB runs a LoRaWAN gateway connected to The Things Network. Based on information from TTNMapper[80] this gateway uses the ”The Things Gateway” Gateway hardware. Telecom Paris runs a nearby gateway labelled `gw-tetech-test`, with unspecified hardware. We note that there are many options for gateway hardware, ranging from off-the-shelf gateways to network interface hardware, called concentrators, that can be connected to compatible computers. During development, we worked with a `RAK2287` concentrator connected to a Raspberry Pi 4 as a gateway or an

IMST `iC880A` gateway concentrator connected to a Raspberry Pi 3 running ChirpStack. However, we cannot use our own hardware on the testbed and, as such, are limited to what they provide. This is why Saclay will be our site for experiments, as there are no public The Things Network gateways at the Lille site. This also means that we are limited to using The Things Network as our LoRaWAN network provider and their software stack.

Moving our attention to the software parameters, all network parameters defined in the networking section in section 4.5.4 need to be defined both on the network stack side and also in TERRA itself. In the network stack, we need to create an application, and then within this application, we create an entry for each device used. So, here we register all the devices we want to use during the experiments. Since in TERRA we use Over-the-air Activation (OTAA) we need to provide the following information for each registered device:

- Frequency Plan
- LoRaWAN Version
- Regional Parameters Version
- AppEUI
- DevEUI
- AppKey
- End device ID

The last being a simple identifier used to address each device within The Things Network.

Some of these settings are fixed for all devices, while others are per device. For our testing scenario, the priority is stable and repeatable results. As written in section 2.2, LoRaWAN contains features to optimise the choice of data rate depending on environmental conditions. In our experiments, we disable or modify these features to ensure that we always use the same data rate. Concretely we:

- Disable Adaptive Data Rate (ADR).
- Fix data rate to DR0. With disabled ADR network rules, we must use DR2 at the minimum[81]. Due to problems on the testbed, however, this data rate prevents our devices from joining, which is why we run with a lower default. In addition, the lowest data rate will have the longest transmission times resulting in the highest energy cost.
- Fix the data rate of the second receive window to DR0 as recommended by the LoRaWAN specification[39, p. 11]. However, The Things

Network Community network recommends using DR3 for the second receive window[38, Additional Information/Frequency Plans]. We comply with the LoRaWAN specification to ensure the same data rate is used for both receive windows as this makes power consumption of a downlink consistent no matter which window it hits.

We also note that, while the recommended delay to the first receive window is 1 second[39, p. 12], The Things Network by default uses a five second wait before the start of the first receive window[82]. We comply with this as this accommodates for any latency within the network servers, even though this will increase the cost of transmission.

In summary, the options and our chosen values for all our devices in the network stack can be seen in table 5.2.

Setting	**Value**
Frequency Plan	Europe 863-870 MHz (SF12 for RX2)
LoRaWAN Version	LoRaWAN Specification 1.0.2
Regional Parameters Version	RP001 Regional Parameters 1.0.2
AppEUI	`0000000000000000`[1]
DevEUI	Per device: `70b3d57ed005ea59` to `70b3d57ed005ea69`.
AppKey	Per app/device, we use a single key for all.
End device ID	Per device: `eui-70b3d57ed005ea59` to `eui-70b3d57ed005ea69`
Adaptive Data Rate	Disabled
Data rate	`DR0`
RX2 data rate	`DR0`
RX1 delay	5 seconds

[1] This is a typical default for development or user-configured EUIs but it slightly reduces encryption entropy, so in production, random AppEUIs are preferred.

Table 5.2: LoRaWAN configuration parameters

Protocol Buffers

For the protocol buffer sizes, we use the defaults shown in table 4.2. We do this since they fit within the RAM of the chosen hardware, and they are large enough to accommodate the queries we want to run on Terra. The options and their chosen values can be seen in table 5.3.

Configuration Option	**Value**
`TerraProtocol.Output.responses max_count:`	16
`TerraProtocol.Expression.instructions max_count:`	50
`TerraProtocol.Query.operations max_count:`	2
`TerraProtocol.Message.queries max_count:`	1

Table 5.3: Protocol Buffers configuration options

Configuration & State Management

Here we also use the default values shown in table 4.2. We note that the maximum LoRaWAN data size here is the default in RIOT, but that the actual highest packet size depends on region and data rate. This is set high enough to contain the packet sizes in the regions supported by TERRA.

The number of window operators supported is also just the default, as this covers the queries we need to run.

The settings chosen can be seen in table 5.4.

Configuration Option	**Value**
`LORAWAN_APP_DATA_MAX_SIZE`	242
`MAX_WINDOW_OPERATORS`	5

Table 5.4: Configuration & state management configuration options

Sensing

For sensing, we simply define the name and type of the chosen sensors. As written in the hardware section above, we utilise the `HTS221` sensor that reads two phenonema: Temperature and humidity. We therefore include the same sensor twice in `SENSOR_NAMES` but with two different types. The chosen parameters can be seen in table 5.5.

Configuration Option	**Value**
`SENSOR_NAMES`	`hts221 hts221`
`SENSOR_TYPES`	`SAUL_SENSE_TEMP SAUL_SENSE_HUM`

Table 5.5: Sensing configuration options

Networking

On the TERRA side of networking the values have to match what they are on the network side as described in section 5.3.2. Additionally, throughout our testing, we regularly experienced dropped packets. This was, at least partially, fixed when we increased the maximum timing error allowed in `semtech-loramac`, by increasing `LORAMAC_DEFAULT_SYSTEM_MAX_RX_ERROR` from 50 to 150. We also keep the default value of 1000 on `FORCED_LISTEN_EVERY_N_LOOP` to ensure no heartbeats are sent during the execution of an experiment. This is to ensure that any transmission from TERRA is triggered by queries. In summary, the chosen parameters can be seen in table 5.6.

Configuration Option	**Value**
`LORAMAC_DATA_RATE`	DR0
`CONFIG_LORAMAC_DEFAULT_ADR`	not present
`LORA_REGION`	EU868
`APPEUI`	00 00 00 00 00 00 00 00
`DEVEUI`	70b3d57ed005ea59-69
`APPKEY`	a fixed secret key
`LORAMAC_DEFAULT_RX1_DELAY`	5000
`LORAMAC_DEFAULT_RX2_DR`	DR0
`LORAMAC_DEFAULT_SYSTEM_MAX_RX_ERROR`	150
`FORCED_LISTEN_EVERY_N_LOOP`	1000

Table 5.6: Networking configuration options

TERRA Main Logic

Here we list the parameters concerning the runtime of TERRA. These concern the execution epoch and the stack memory of the expression execution runtime. We set the execution epoch to 120 seconds, which means that TERRA will wake up and execute every 2 minutes, giving us 30 executions an hour. We set the runtime stack memory to 10, meaning that up to 10 numbers can be stored on the stack. This covers our use case. In summary, the chosen settings can be seen in table 5.7.

Configuration Option	Value
`RUNTIME_STACK_MEMORY`	10
`EXECUTION_EPOCH_S`	120

Table 5.7: TERRA main logic configuration options

RIOT specific build options

RIOTs parameters are mostly on choice of features included in the build and whether or not the development modes are enabled. Note also that, depending on the hardware chosen, RIOT also enables different drivers or other modules to run and interact with that hardware. There is a tight coupling between the hardware chosen and the performance of RIOT.

We enable development helpers when running TERRA to enable stack overflow protection, more informative log messages and enabling asserts [75, see modules/utilities]. We also modify the default stack size of RIOT to accommodate TERRA. An overview of the parameters can be seen in table 5.8.

Configuration Option	Value
`DEVELHELP`	1
`MAIN_STACKSIZE`	5120

Table 5.8: RIOT specific build options

Besides these parameters the Makefile also includes Riot modules relevant for the OS features TERRA uses, and the driver modules needed for interacting with sensors. We also note that while we use the `GCC` compiler we do not apply any `GCC` optimisations. We do this for multiple reasons; primarily, this is not supposed to be a test of the effectiveness on a specific compiler's optimisations but of TERRA itself. And depending on the hardware used, it might be better to optimise for size rather than speed. Running without optimisations gives a more correct view of the performance of Terra and facilitates easier comparison between compilations and, therefore, reproducibility of results. We also use the default log level of Riot OS, `LOG_INFO`. This affects Riot to a lesser degree, but since Terra also utilises the logging framework, it does output additional information needed for the experiments and debugging.

5.3.3 Metrics

We will assume TERRA only receives valid queries and failure states will therefore not be studied. The default behaviour of failure states in TERRA is to ignore it and continue, so the energy consumption of a failure state should not increase power consumption. During normal execution of TERRA it goes through several activities, as seen in fig. 4.2. The metrics measured will be:

- Power consumption in Watts and their associated timestamps.
- Timing data from TERRA on how long each activity took which is outputted over serial.

We will not measure timing data for sleep. We will primarily use the two metrics to convert into Joule for energy consumption analysis.

Power Consumption

As described in section 2.3 the data is collected on the IoT-LAB testbed using the `INA226`[44] Current and Power Monitor. The configuration used for the experiments can be seen in table 5.9.

Configuration Option	**Value**
Conversion Time/Period	140 µs
Averaging Mode	4

Table 5.9: IoT-LAB Power monitoring configuration options

This, according to eq. (2.1) in section 2.3 leaves the experiments with a final sampling rate of 1120 µs. This value was chosen as a nice trade-off between noise reduction and frequency. The sampling rate needs to accommodate LoRaWAN packet transmissions, which can be in the order of seconds, and execution of queries on embedded devices, which is in milliseconds. In addition, this also provides measurements around the same resolution as the internal timings measured in TERRA, which we will discuss next.

```c
conf_load_time_ms = ztimer_stopwatch_reset(&stopwatch);

// DESERIALIZE MESSAGE
LOG_INFO("Deserialize message if any...\n");
if (config.raw_message_size > 0)
{
  LOG_INFO("Message there! Deserializing...\n");
  bool res = serialization_deserialize_message(config.raw_message_buffer,
  ↪  config.raw_message_size, &msg);
  print_terraprotocol_message(&msg);
  if (!res)
  {
    LOG_ERROR("Failed to deserialize\n");
  }
}

deserialize_msg_ms = ztimer_stopwatch_reset(&stopwatch);
```

Listing 5: Timing code and deserialisation call from `main.c` in Terra

```
TIMINGS> Loop: 1, Sync: 805 ms, Load: 126 ms, deserialize: 26 ms, sensor init: 13 ms, net
↪  init: 53 ms, Collect: 14 ms, Exec tflite: 0 ms, Exec query: 2 ms, Send: 4039 ms, save
↪  config: 1463 ms, Sleep: 113 s
```

Listing 6: Example of timing measurement output

Measuring Timings

To measure the duration of the different activities in Terra, we modify Terra by incorporating timing measurement code. For this, we use the stopwatch capabilities of the `ztimer` module [75, modules/system/ztimer high level abstraction layer]. To minimise the impact of the timing code, a single stopwatch is used and reused for all tracking. All values are saved during an epoch and only at the end of an epoch is the values outputted as serial over `UART`. To ensure the impact of timing the code is consistent across all experiments, and to ease parsing of the timing output, all activities are timed no matter if they are used for a particular execution or not. That is, in the start up phase there is no deserialisation, but this is still timed to ensure the impact of timing is similar. This can be seen in listing 5. At the end of each epoch all timing information is output through serial together. An example of this can be seen in listing 6.

As we are measuring power consumption at the same time and timings on embedded devices are often provided by dedicated hardware, we take care to only use timing peripherals that Terra would use in normal operation. Choosing ones with higher resolution will provide better timings, but will also skew the power measurements. As Terra already requires a ms precision clock to calculate the required sleep time, we utilise the same for timings. This does give us ± 1 ms inaccuracy, but we judge that it is a worthwhile trade-off for more accurate power measurements. It also depends on the accuracy of the underlying time-keeping hardware.

Although the timings for activities within each epoch, as shown in listing 6, are accurate within the epoch, we do not know when the epoch starts and as such cannot use it to timestamp our power measurements. This is made more difficult by the IoT-LAB infrastructure, as there are no guarantees of when power measurements start in relation to when Terra starts.

To correlate the power measurements with the timing measurements, we incorporate a signal into our power measurements that we know triggers on every epoch start. Similar to Rice and Hay [83], we flash all available LEDs on the board twice with known timings, and then we can look for similar rises and falls in the power measurements to synchronise the two series of measurements. This simple approach turned out to be adequate for our experiments, since the LED's provide a big enough power draw to stand out from the noise of the signal. We incorporate a known idle time at the start and end of each synchronisation signal to make sure the power draw stabilises before continuing. Again, the flashing of these LEDs depends on the time-keeping accuracy of the `ztimer` library.

5.3.4 Workload

Here we look at the workload parameters. These parameters are the ones we vary during our experiments (factors in Jain [41]). The workload parameters are:

Query Terra is built to execute queries provided by an end user. Normally, these are delivered to Terra via a network, as described in section 4.5, but we also utilise the option to preload them in Terra, which we discuss below.

TFLite Model included Terra can include a TFLite model that executes after sensor data collection, but before query execution. By default, we include a representative TFLite model provided by ELEGANT, as described in section 4.5.4

Queries

Terra's behaviour, and therefore its energy consumption, is highly dependent on the queries it is provided. Both in terms of their content and their size. Terra can receive queries consisting of one of three types of operations:

Map An operation that contains an expression to be evaluated and an attribute to save the result to. The saved result will always be sent back, unless a later filter cancels it.

Filter Mostly used together with a map, a filter can cancel execution of a query if its provided expression evaluates to false. As such a Map-Filter combo can be used to set up a query that e.g. only send responses if a value is above some threshold.

Window An operation that often requires multiple executions before providing an answer. This operation contains an aggregate instruction that it runs once every execution and then reports the result after a specific amount of time.

Each Terra query will be run repeatedly indefinitely until a new query is provided. Terra will after each execution either send a result back to the user over the network or not if, e.g. the query was cancelled by a filter, is an in-progress window or it contains an illegal expression. When producing the queries, we assume that each instruction in the expression has the same effect on the metrics and that they are independent of each other. In other words, the content of the expressions does not matter, only the length in bytes in relation to network activities and the number of instructions with relation to the execution activity. Given this, we produce two sets of queries to execute in our experiments:

1. A set varying the length of the queries in bytes.
2. A set varying the response rate of the query.

These will be described in more detail below.

1. Query Length

When varying the query length, we have to be aware of the limitations of LoRaWAN network transfer and, under the experimental parameters, we set in section 5.3.2. To test outside these limitations, we can utilise the option of pre-baking the query, using the configuration options described in section 4.5.4 and sidestepping the network transmission completely. However, this will not allow us to test the effect of query length on network-related activities.

We therefore choose a two-legged approach. For the network-related activities we keep within our practical network limitation, but for the query length-dependent activities that are not dependent on actual network transfer of the query, we pre-bake larger queries until we near the hard-coded size limit of LoRaWAN to get a more accurate result. Note that in real-world usage this is the limit that counts, since the max packet size will depend on the data rate, which in turn depends on environmental factors such as distance to gateway, noise, and any material blocking line of sight. We note that since we fix our data rate to the lowest available, we are limited to ≈ 10 instructions in our experiments, while in the real world 5 times as many are often possible.

Consumption-wise, transferring at higher data rates is always cheaper as each increase in data rate roughly halves the time on air. Our results will therefore be a conservative measure of the energy cost of network transfer.

The network related queries consist of a single map operation of increasing expression size, until we near the maximum payload size of 50 B.

The pre-baked queries follow the same exponential increase in size until we come near the 242 B limit. Including pre-baked queries slightly changes Terras behaviour, as normally queries are not executed during it's startup phase, since there is no query provided over the network yet. With the inclusion of pre-baked queries, TERRA can and will execute them. For the purpose of the model, we assume that queries are always delivered over the network and will disregard the startup cost of query execution and its related activities.

The expression of the map is a combination of arbitrary instructions defined in such a way that we do not exceed the maximum stack size defined in section 5.3.2. Using both constant values and sensor values.

The queries we run in our experiments are shown below. These are produced only to conform to the specified size, and their computation is as such meaningless. The queries are as follows:

16 Byte: A map operation with

> **Expression:** `VAR, 0`
>
> **Result stored in Environment index:** 0

32 Byte: A map operation with

> **Expression:** `VAR, 0, CONST, 8, MUL, FLOOR`
>
> **Result stored in Environment index:** 0

46 Byte: This is close to the maximum LoRaWAN packet size for DR0, which is 50 B. The query is a map operation with:

> **Expression:** `VAR, 0, CONST, 256, MUL, CONST, 256, DIV, FLOOR`
>
> **Result stored in Environment index:** 0

64 Byte: A map operation with:

> **Expression:** `VAR, 0, CONST, 8, MUL, FLOOR, VAR, 1, DIV, ABS, CONST, 8, MUL, FLOOR`
>
> **Result stored in Environment index:** 0

128 Byte: A map operation with:

> **Expression:** `VAR, 0, CONST, 8, MUL, VAR, 1, DIV, CONST, 8, MUL, VAR, 0, CONST, 8, MUL, VAR, 1, DIV, CONST, 8, MUL, VAR, 0, CONST 8, MUL, MUL, MUL, FLOOR`
>
> **Result stored in Environment index:** 0

214 Byte: This is around the maximum allowed packet size. This query is a map operation with:

> **Expression:** `VAR, 0, CONST, 8, MUL, CONST, 512, MUL, VAR, 0, CONST, 8, MUL, VAR, 1, MUL, CONST, 8, MUL, VAR, 0, CONST, 8, MUL, VAR, 1, MUL, CONST 8, MUL, VAR, 0, CONST, 512, MUL, VAR, 1, CONST, 8, MUL, VAR, 0, CONST, 8, MUL, MUL, MUL, MUL, MUL`
>
> **Result stored in Environment index:** 0

2. Response Rate

For testing the response rate, we have the option of either producing queries with a map followed by a filter, or utilising windows. Since filters conditions on values in the environment or stack and those values are either provided by sensors or hard coded in the filter itself, it is impossible to reliably use these to fix the response rate to, for example, only transfer results on every

3rd execution. Windows, on the other hand, are designed for this, and with careful selection of the window size in relation to epoch time we can produce queries that consistently produce result at the rate we want; every 1st, 2nd, 4th and 8th epoch.

Since the epoch duration is 2 minutes (see section 5.3.2) the queries are going to consist of a single window operation with sizes of 2 min, 4 min, 8 min and 16 min. The query details can be seen below:

Aggregation: `COUNT`

On environment index: 0

Save in environment index: 1

Save start value in environment index: 2

Save end value in environment index: 3

Window type: Tumbling with window sizes of 2 min, 4 min, 8 min and 16 min

TensorFlow Lite

In addition to this, Terra also supports TFLite models, as described in section 4.5.4. These run after sensor values have been collected but before queries are executed, and from a query's point of view, they exist as a sort of pseudosensor whose value depends on the sensor values. Since a TFLite model can be constructed in almost infinitely many ways, it is outside the scope of this thesis to perform a detailed analysis of its effect on power consumption. Instead, we chose a representative TFLite model, the one provided by Terra by default, and model it as a binary cost. The goal is to increase the computation time in Terra in order to gauge its overall effect on energy consumption. Of course, this is highly dependent on the model being run.

5.3.5 Experiment

The metrics collected in the experiments will be externally measured power consumption and internally tracked timing data for the end devices. No metrics will be measured for any other parts of the system. All experiments will be carried out on 11 devices. All experiments will run for 60 minutes, which with an epoch time of 120 seconds will give at most 30 epochs/executions. We will do 1 experiment per workload.

For each experiment, the power consumption for the end device will be monitored at a sufficient sampling rate to discern the power consumption of individual activities.

So in summary, the following series of experiments will be conducted:

Query Length: 6 experiments: 3 with map queries of length 16 B, 32 B, 46 B and 3 with pre-baked queries of 64 B, 128 B and 214 B.

Response Rate: 4 experiments with window queries of size 2 min, 4 min, 8 min and 16 min.

TFLite: 1 experiment with TFLite enabled and a map query that transfers the TFlite output.

Experiment Assumptions and Trade-offs

Here we briefly discuss the assumptions and trade-offs made in designing the experiments. There are 2 main points: First, we assume that identical devices have identical power consumption. Second, in our experiment design, we implicitly assume that there are no interactions between workload parameters. We address these concerns in this order.

At first, the assumption that identical software running on identical hardware should reveal identical, or at least very similar, power consumption is not unreasonable. After all, they are all covered by the same datasheet.

Regarding workloads, there are multiple ways to design our experiments. We utilise One-Factor-at-a-Time, but other options exist, such as full factorial or fractional factorial experiments [41, ch. 16]. This is done out of simplicity, since it is easier to execute and reason about, and it isolates the workload/factor being changed. However, it runs the risk of not uncovering the interaction between the workload parameters. If, say, power consumption during one activity increased exponentially if both response rate and query length varied. However, as can be seen in table 5.1, the model is designed in such a way that no workloads affect the same activity at the same time. `SendReceive` is both affected by QL and RR, but not in the same phase. So we can conclude that there is no interaction between workload parameters, and the main drawback of One-Factor-at-a-Time experiments is not an issue in our case. However, another approach with fractional factor experiments could have given similar results with fewer experiments.

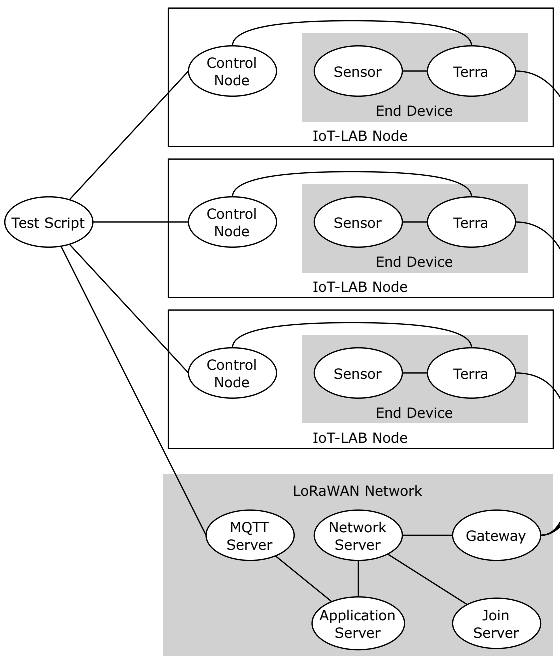

Figure 5.2: System under test with the test infrastructure shown

Experiment Execution

We now take a look into the work done to execute the actual experiments.

As can be seen in fig. 5.2, running an experiment involves many moving parts. What is not shown in fig. 5.2 is the fact that, since every device needs its own set of network keys, it also needs to be individually compiled. Also, since TERRA makes use of persistent storage, it is required to clear storage before experiments can start, in case previous users of the nodes left data that can be parsed mistakenly by TERRA, or TERRA was the firmware executed previously. For consistency and reproducibility, all experiments are defined by an experiment `.json` file, and keys and passwords are stored in a corresponding `secrets.json` file. This experiment file contains the following.

- Information needed to build TERRA for each device.
- Information needed to provision devices at IoT-LAB.
- Information needed to push the built firmware to the devices on IoT-LAB.

- Duration of experiment.
- TERRA Epoch duration.
- Login information for the ssh server on the testbed - excluding the password.
- Login information for the MQTT server on the Network Stack - excluding the password.
- The query (`PAYLOAD`) to be sent to the devices.

An example of such a file can be seen in the repository[2].

The test script goes through the following high-level steps in sequence when running an experiment:

1. Parse the provided experiment configuration file and `secrets.json` file.
2. Build persistent storage reset firmware for each device.
3. Build TERRA for each device.
4. Registering an experiment on IoT-LAB and start script that collect all serial output in one file on the IoT-LAB ssh front end.
5. Fetching assigned nodes from IoT-LAB.
6. Initialise the SQLite database used for storing experiment data.
7. Upload and execute firmware to clear persistent storage on IoT-LAB nodes concurrently.
8. Clear down link queries on the Network stack through MQTT.
9. Start parallel job of listening to MQTT to record events to SQLite database and push payload when nodes join the network.
10. Wait for experiment to finish.
11. Download serial output file, and all power consumption data.
12. Parse and add serial output traces to SQLite database.

[2] `https://github.com/FlapKap/Terra-experiments/blob/main/experiment_10_32byte_map.json`

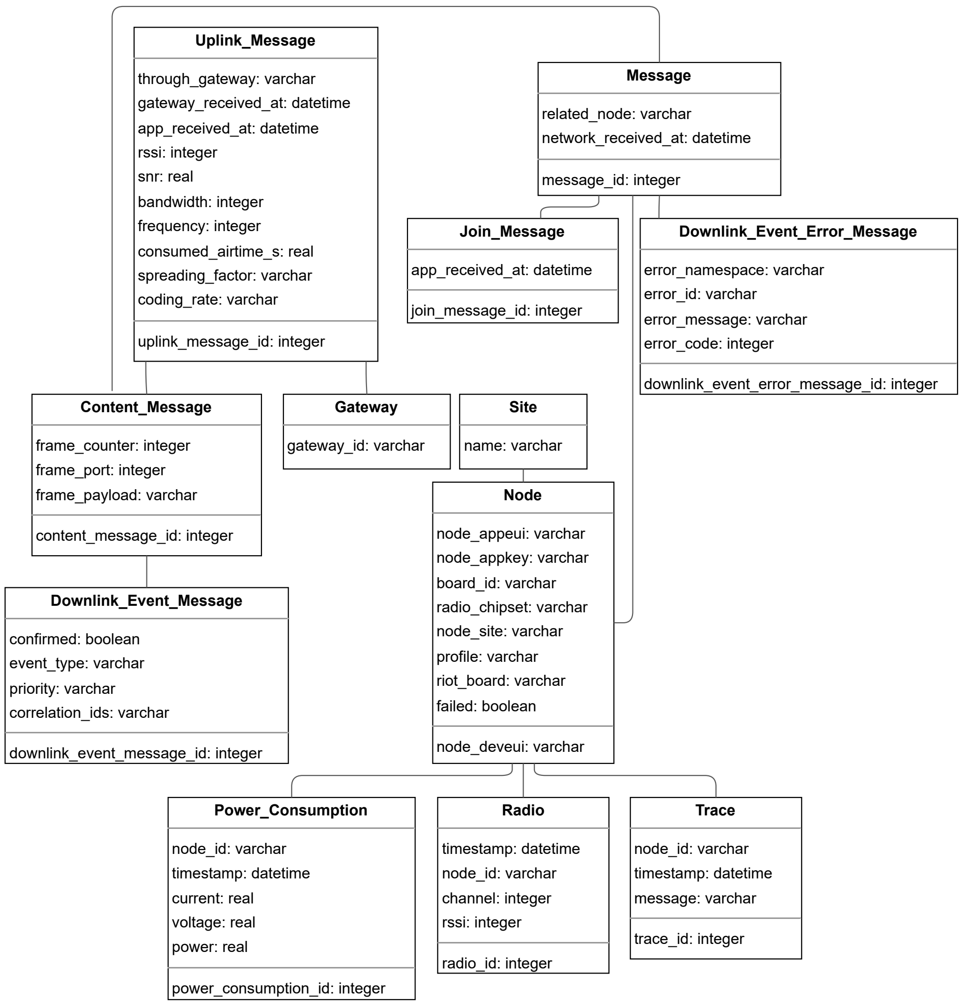

Figure 5.3: SQLite Schema used for experiment data storage

13. Parse and add power consumption data to SQLite database.

The complete test script can be seen in the repository[3].

SQLite[84] is used as the default data storage format for the experiments. The complete data definition language file of the experiment database can be found in the repository[4]. Figure 5.3 describes the layout of the database.

[3]`https://github.com/FlapKap/Terra-experiments/blob/main/experiment_runner/__main__.py`

[4]`https://github.com/FlapKap/Terra-experiments/blob/main/experiment_runner/resources/experiment.db.sql`

The tables of note are `Trace` and `Power_Consumption` since these contain the information needed to discern how much energy is used in each activity.

All the code to execute the experiments can be found in the repository[5].

5.4 Experimental Results

Based on the experiment framework described in the previous chapter we, in this chapter, execute the experiments and interpret the results, to finally develop the energy model. First, we go through the processing of the data to get it to a state where we can do linear regressions to derive the α's and β's in the models. After this, we go through each of the results and then finally produce the model.

5.4.1 Data Processing

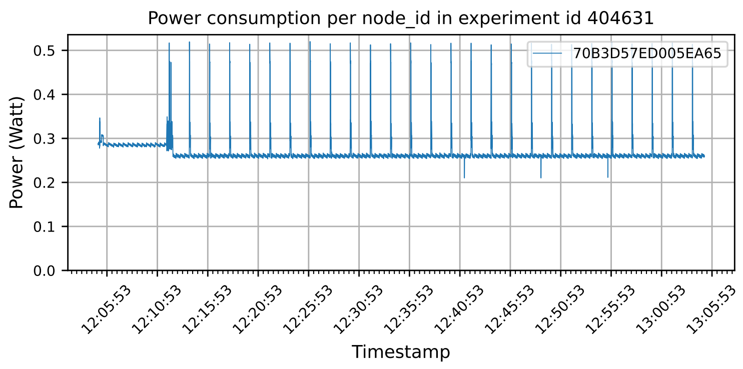

Figure 5.4: Raw power consumption measurements

To get to the point where the models described in section 5.2 can be produced, the data collected during the experiments need to be pre-processed. The collected data is a series of power measurements and timings. These have to get converted into energy consumption per activity per epoch. This is done in a Jupyter Notebook named `experiment_analysis.ipynb`[6], using Pandas[85], DuckDB[86], matplotlib[87] and NumPy[88]. At a high level the `Power_Consumption` data is:

[5] https://github.com/FlapKap/Terra-experiments

[6] https://github.com/FlapKap/Terra-experiments/blob/main/experiment/experiment_analysis.ipynb

1. Filtered to only make use of nodes that received and run queries.
2. Edge detected to find epoch starts.
3. Joined with timings from `Trace` to assign each row in `Power_Consumption` an activity.

We expand on this process below. The data is stored in the database format described by fig. 5.3. We look through the `Trace` table to find nodes that received and are executing a query:

```
SELECT node_id, count(*) FROM expdb.Trace WHERE Trace.message like 'Execute Queries%'
↪  GROUP BY Trace.node_id
```

Then only `Power_Consumption` data containing these `node_id`'s are selected. An illustration of the data at this point can be seen in fig. 5.4. Note for this experiment only a single node got an executed a query. After this edge detection is done to find the synchronisation signals described in section 5.3.3. This detection is very rudimentary. First, we define the length of the different stages of the synchronisation signal. Then we calculate the differences in power consumption between subsequent measurements. Using that, we can find sudden rises or falls in power consumption that corresponds to the change in power consumption demanded by the toggle of LEDs. The value of this change is found through experiments. This produces a large amount of false positives, but by knowing the length of the different phases of the synchronisation signal we can filter out any edge that do not conform. The first two synchronisation signals detected in fig. 5.4 can be seen in fig. 5.5. Here we are zoomed in enough to see the individual detected edges and the whole synchronisation signal period.

Each epoch starts with two synchronisation signals, so the next thing we do is to find synchronisation signals that are very close together, $< 10\,\mathrm{ms}$, but not overlapping, and filter out any that aren't. Then we mark the start of epochs (`loops` in code) on every two subsequent synchronisation signals.

The synchronisation signal detection code can be seen in its totality in the repository [7].

With synchronisation signals found, we can combine the data with the timings provided by Terra through the serial output. These are stored as a single row per line of serial out in the `Trace` table.

[7] https://github.com/FlapKap/Terra-experiments/blob/main/experiment/experiment_analysis.ipynb

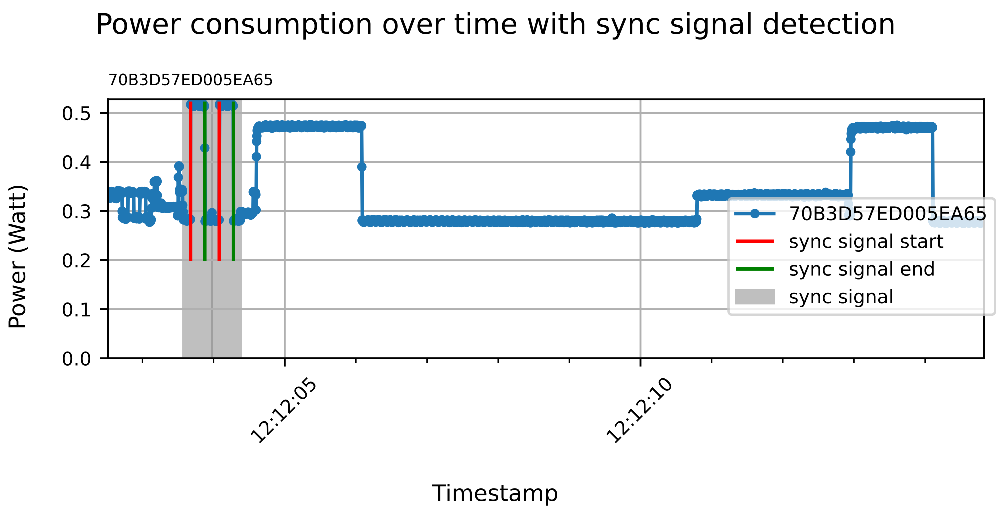

Figure 5.5: First synchronisation signal detected in fig. 5.4

We extract the concrete timings from the SQL Table using DuckDB `regexp_extract` function. This gives us a table with the columns:

- `node_id`
- `loop_signal_end`
- `loop_num`
- `timestamp`
- `sync_time_ms`
- `load_time_ms`
- `deserialize_time_ms`
- `sensor_init_time_ms`
- `net_init_time_ms`
- `collect_time_ms`
- `exec_tflite_time_ms`
- `exec_query_time_ms`
- `send_time_ms`
- `save_config_time_ms`
- `sleep_time_s`

We now augment our epochs with these timings. This is done by adding e.g. a `sync_end` time thats equivalent to the known `start_time` + `sync_time_ms`. `load_start` is then `sync_end` + `load_time_ms` and so on.

Then we utilise these timings to categorise each measurement in `Power_Consumption` with its corresponding activity, per node id. Note that the `Riot Initialisation` activity is a special case. Figure 5.6 shows a section of power measurements for a single device with activity periods marked. Note the unmarked area between `sleep_time` and `sync_time`.

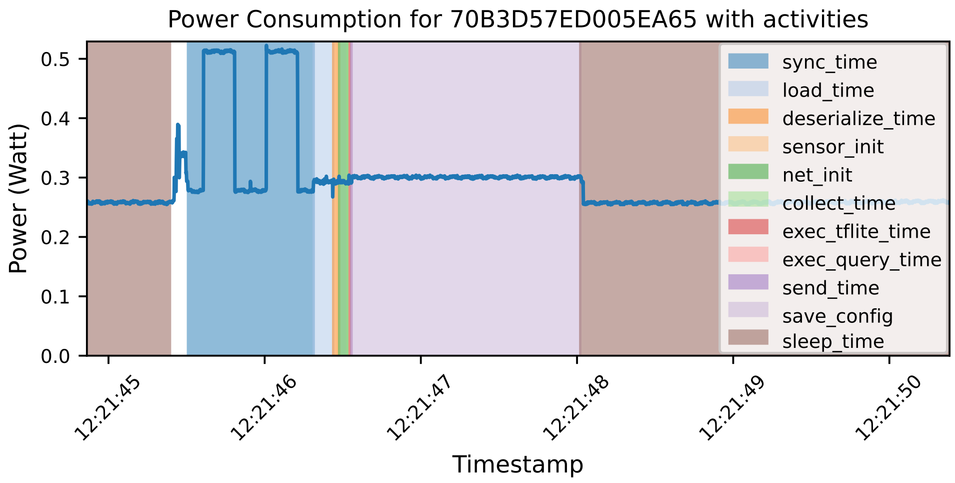

Figure 5.6: Power measurements of a single node with activities marked

Since the OS is not initialised, we cannot use the OS itself to measure its initialization duration. We therefore mark the still uncategorised measurements between the sleep activity ending of the previous epoch and the synchronisation signal beginning of the next as `Riot Initialisation`. We cannot do this for the startup epoch since there is no previous sleep, and the power measurements start at the start of the experiment before the firmware is even uploaded. We then assign epoch numbers to each epoch. For the startup epoch it starts from the beginning of the synchronisation signal. For the rest, it starts from riot initialisation. At this point the table of power consumption data contains the columns of timestamp, node id, power, current, voltage, activity and epoch number and this data is saved as a parquet file, ready for the next step of the analysis: the fitting of the models.

Since the models are working in units of energy, joule, we need to convert our measurements of power, Watts, into joules. Power is defined as energy over time, so to get energy from power we multiply it by the period of time the power is exerted. We assume that the power between measurements is constant. So we can calculate the energy for some period from n to $n+1$, $E_{n,n+1}$, by finding the length of the time between measurements and multiplying it by the measured power at n P_n.

$$E_{n,n+1} = P_n \cdot (t_{n+1} - t_n) \tag{5.17}$$

In terms of units, Joule is defined as Watts times seconds, so our units work out as long as the time difference is represented in seconds.

Activity	Phase	Mean	Standard Deviation
0. RiotInit	steady state	0.035 30	0.000 49
1. Load	startup	0.036 13	0.000 17
2. Deserialize	startup	0.000 71	0.000 24
3. SensorInit	startup	NaN	NaN
	steady state	0.003 79	0.000 16
4. NetInit	startup	2.753 97	0.111 02
	steady state	0.015 27	0.000 24
5. SensorCollect	startup	0.002 45	0.000 16
	steady state	0.003 78	0.000 26
6. ExecTFlite	startup	NaN	NaN
7. ExecQuery	startup	NaN	NaN
9. SaveConfig	steady state	0.434 84	0.004 60

Table 5.10: Average consumption in Joule and their standard deviation for the activities that are not dependent on QL, RR or TF

5.4.2 Experiments

In this section, we go through each set of experiments, derive the values for the model, and discuss the expected and actual results.

Unless otherwise noted, the data points are joules spent for an epoch on one device. All α, β and r^2 have been rounded to 4 decimal places.

Constants

First, we look at the constants. These are the activities that are not dependent on QL, RR or TF. These are simply calculated averages of joules per epoch for all devices of all network-bound experiments. Since experiments with pre-baked values represent a slightly different workflow and we in our model presume that there are no pre-baked queries (see section 5.3.4) we do not use them here. Additionally, we want to verify that the activities marked "n/a" in table 5.1 (Activities 2, 3, 5, 6 and 7 in the startup phase) are not applicable and that the averages we calculate are valid, i.e. they follow a normal distribution. The averages can be seen in table 5.10.

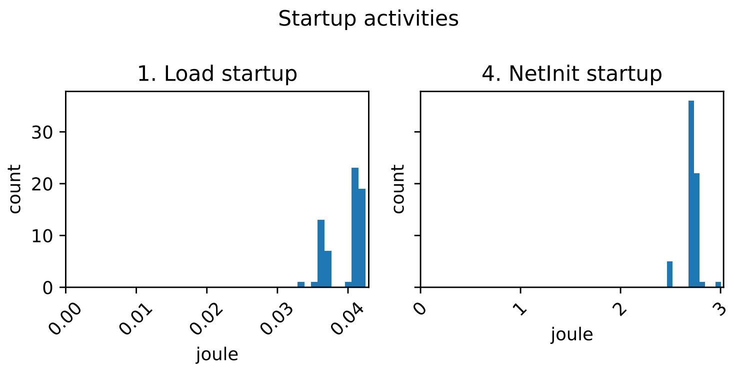

Figure 5.7: 10 bin histogram of consumption in of startup activities

Here we see that activities 3, 6, and 7 have no energy consumption in the startup phase. The average consumption is `NaN`, since there is no recorded activity duration for these. However, `SensorCollect` and `Deserialize` has an average cost that is not expected in the startup phase. This consumption is due to unrelated logs done by TERRA within that activity. This takes enough time to affect our measurements, but is not inherent in the activity itself.

Also note that, due to the way that `RiotInit` is measured, we cannot measure the cost of this activity in the startup phase. We reasonably assume that the cost is independent of the phase and conclude that $C_{0,0} = C_{1,0}$.

We now check that the distribution of the constants is normal. First, we look at the startup phases in fig. 5.7. We see that `netInit` activity follows a normal distribution, while the `load` startup activity does not. However, the values are relatively close, so an average is still a reasonable estimate of `load` startup consumption.

Looking at the steady-state histograms in fig. 5.8 we notice that `RiotInit` and `SaveConfig` follow normal distributions, while `SensorCollect` and `NetInit` seem more bimodal, and `SensorInit` is more spread out. We also notice that in general their energy consumption is very low. Since `SensorInit` and `SensorCollect` are dependent on sensor hardware, communication overhead and timings may affect the consistency of measurements. However, since their standard deviation is still quite low, as seen in table 5.10, we judge it valid to use the mean in the model.

Based on this, we get the constants:

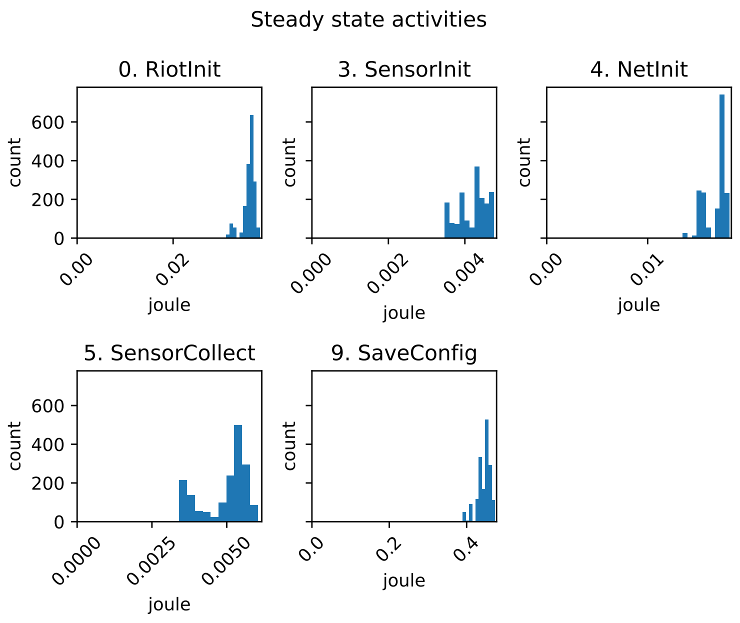

Figure 5.8: 10 bin histogram of consumption in of steady state activities

$$C_{0,0} = C_{1,0} = 0.03530$$
$$C_{0,1} = 0.03613$$
$$C_{0,4} = 2.75397$$
$$C_{1,3} = 0.00379$$
$$C_{1,4} = 0.01527$$
$$C_{1,5} = 0.00378$$
$$C_{1,9} = 0.43484$$

Query length

Now to look at the query length experiment data. As written in section 5.3.4, we use a two-legged approach for all models that are not network-bound. For the network-bound models we only utilise the 3 experiments where queries are transferred over the network.

Startup Phase

Here we go through the fitting of eqs. (5.1) and (5.2).

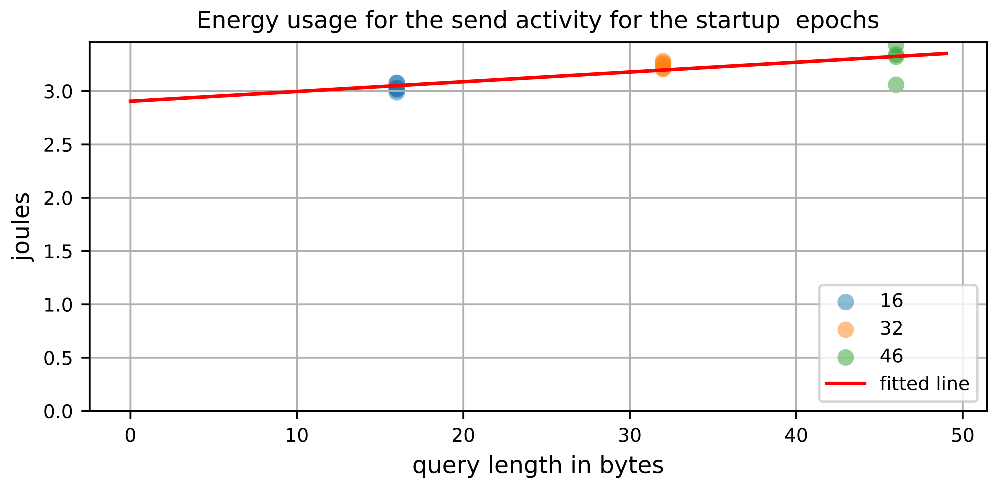

Figure 5.9: Fitting model eq. (5.1) to data

Startup Send-Receive Activity

This is a network-bound model, so we only utilise the 16 B, 32 B and 46 B experiments. Looking at fig. 5.9 and eq. (5.18), we see that there is a correlation between the length of the query and the energy consumption of the sending activity. This is expected as the larger the packet, the longer airtime is needed to transfer it. We also see that there is a high base cost of 2.9 J per send activity. LoRaWAN dictates a send before a receive, and due to the increased delay in the first receive window (see section 5.3.2) the radio is on for a significant amount of time for every transmission, which will explain the high base cost.

The fitted values are:

$$\begin{aligned}\alpha_{0,8} &\approx 2.9052\\ \beta_{0,8} &\approx 0.0091\\ r^2 &\approx 0.6496\end{aligned}\tag{5.18}$$

Startup Save Configuration Activity

This is not a network-bound model, so we look at both the normal and pre-baked experiments. The data and linear model can be seen in fig. 5.10. Here we see a strong linear relationship between Joule and Query Length. The longer the query, the more energy is needed to save that query to persistent storage. This is what we expect, for the board used for the experiments, as here is the persistent storage technology EEPROM as described in section 5.3.2. Note that other boards might use different

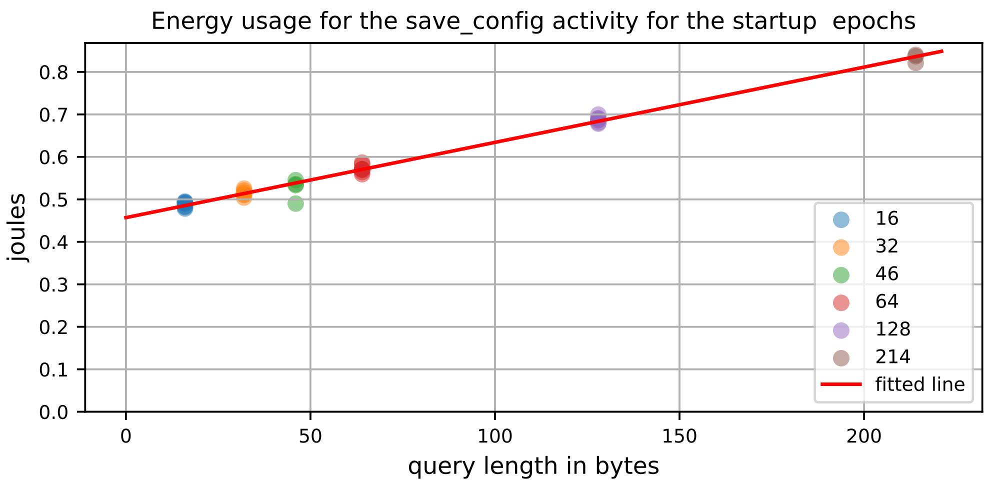

Figure 5.10: Fitting model eq. (5.2) to data

storage technologies such as battery-backed RAM, in which any correlation between length and energy will be much diminished due to faster save times. The fitted values are:

$$\begin{aligned} \alpha_{0,9} &\approx 0.4570 \\ \beta_{0,9} &\approx 0.0018 \\ r^2 &\approx 0.9887 \end{aligned} \tag{5.19}$$

Steady-State Phase

Here we go through the fitting of eqs. (5.3) to (5.5).

Steady-State Load Configuration Activity

This is not a network-bound model, so we look at both the normal and prebaked experiments. The data and linear model can be seen in fig. 5.11. Here, we see no strong correlation between load times and query length. The load activity covers moving state from persistent storage to RAM. This includes the saved query, but additionally the network state, the window state, and others. The lack of correlation is somewhat surprising, since the query constitutes a significant chunk of the state to load. Answering the question of why this correlation is lacking would require a deeper investigation of the behaviour of the load configuration code and its interaction with the EEPROM. One theory is that loading continuous chunks of memory from EEPROM is fast, so the majority of the overhead is not the actual data transfer but the overhead of initialising the interaction with EEPROM. In

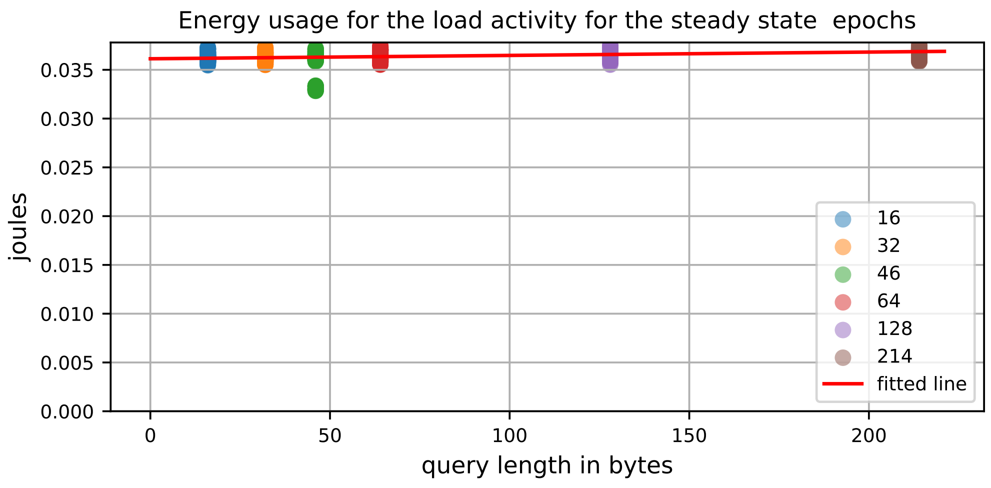

Figure 5.11: Fitting model eq. (5.3) to data

any case, based on the analysis above, we can conclude that the amount of energy used in the steady-state load activity is constant. We conclude it to be the average of all measured values which is 0.0363 J.

The fitted values for the linear model are:

$$\begin{aligned} \alpha_{1,1} &\approx 0.0361 \\ \beta_{1,1} &\approx 3.4337 \times 10^{-6} \\ r^2 &\approx 0.0762 \end{aligned} \tag{5.20}$$

But given our analysis, we replace $f_{1,1}(\mathrm{QL})$ defined in eq. (5.3) with a constant $C_{1,1} = 0.0363\,\mathrm{J}$

Steady-State Deserialise Activity

This is not a network-bound model, so we look at both the normal and prebaked experiments. The data and the fitted line can be seen in fig. 5.12. Here we see a very nice linear relationship between the amount of joules spent deserialising the query and the length of the query. This is what we expect. The fitted values are:

$$\begin{aligned} \alpha_{1,2} &\approx 0.0052 \\ \beta_{1,2} &\approx 0.000\,166 \\ r^2 &\approx 0.9974 \end{aligned} \tag{5.21}$$

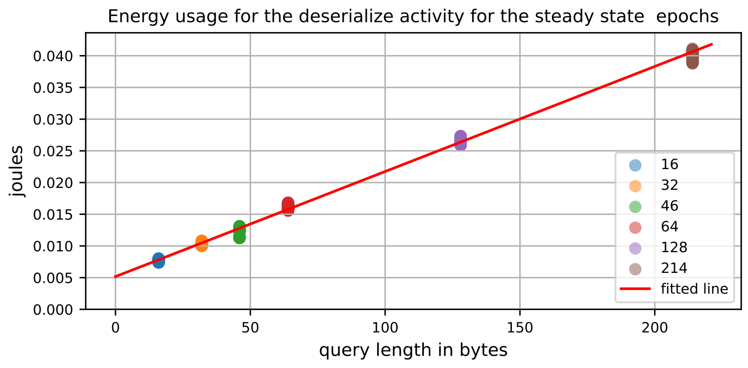

Figure 5.12: Fitting model eq. (5.4) to data

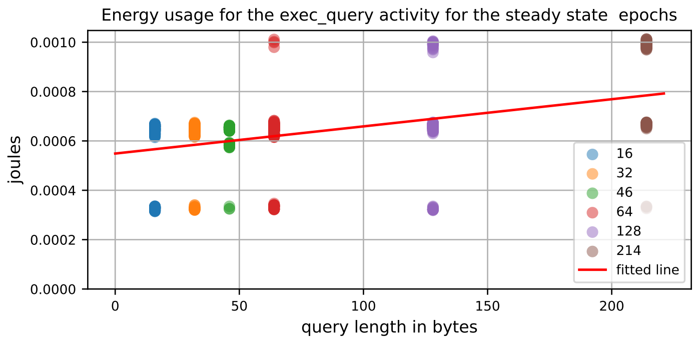

Figure 5.13: Fitting model eq. (5.5) to data

Steady-State Exec Query

The data and fitted line can be seen in fig. 5.13. First thing to notice is the discontinuous nature of the data in fig. 5.13. It almost seems trimodal. However, this is due to the execution of queries being so fast that we are close to the measurement frequency employed in the experiments. This becomes especially clear if we look at the average duration of this activity for each query length in table 5.11. Since joule = Watt · seconds and our duration is within one or two ms this will naturally produce distinct data points based on whether we have 1,2 or 3 measurements covering the `exec_query` activity.

The fitted values are:

$$\begin{aligned} \alpha_{1,7} &\approx 0.000\,549 \\ \beta_{1,7} &\approx 1.1006 \times 10^{-6} \\ r^2 &\approx 0.1889 \end{aligned} \tag{5.22}$$

Query Length	16	32	46	64	128	214
Avg. duration (ms)	1.909	2.051	2.204	2.118	2.286	2.74

Table 5.11: Average duration of the exec query activity per Query Length

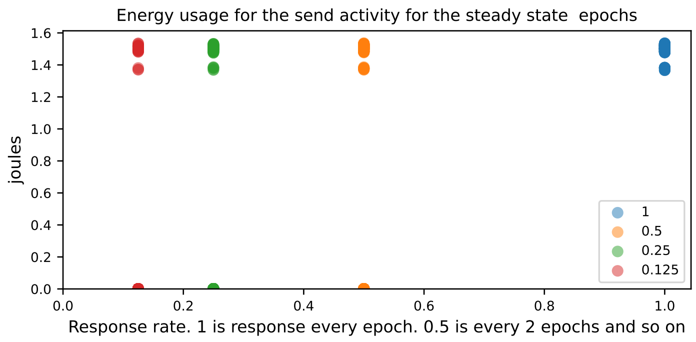

Figure 5.14: Fitting model eq. (5.7) to data

We choose to use linear model as an approximation despite its low coefficient of determination.

Response Rate

Here we go through the fitting of eq. (5.7). A plot of the data can be seen in fig. 5.14. It is clear from the figure that the cost of sending, when it happens, is constant and independent of the response rate. This also makes sense since the value transferred is scalar, and thus also it's size independent from the number of epochs. There could be edge cases with `SUM`s producing a larger and larger number that requires an increasing amount of energy to transfer, but those are not representative of general use.

Looking closer at the data, in fig. 5.15, we see the source of the bimodal nature of the nonzero data in fig. 5.14. For each experiment, a single node seems to have a lower energy consumption than the others.

To get a more conservative average we exclude not only the non-zero energy consumption, but also the single device with lower consumption. Our average then becomes:

$$C_{1,8} = 1.5069\,\text{J} \tag{5.23}$$

Which then leaves us with the simple model:

$$f_{1,8}(\text{RR}) = 1.5069\text{RR} \tag{5.24}$$

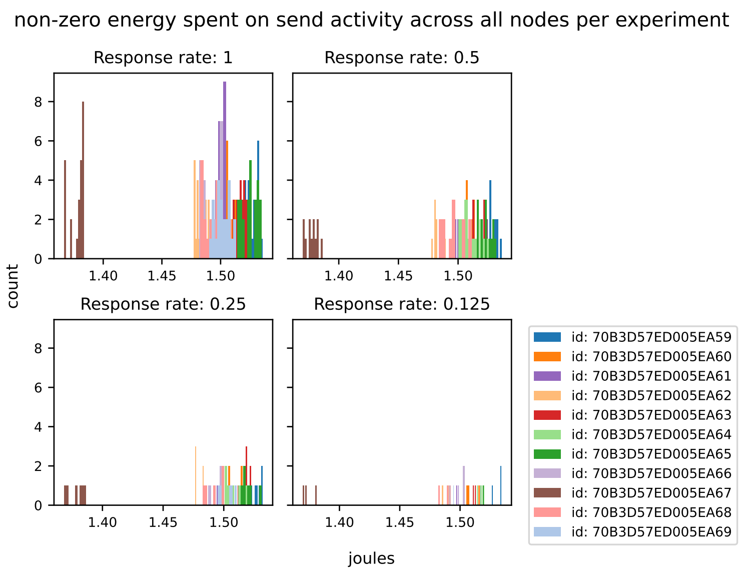

Figure 5.15: 4 histograms of non-zero energy consumption for the send activity coloured per device. Note the X-axis does not start at 0

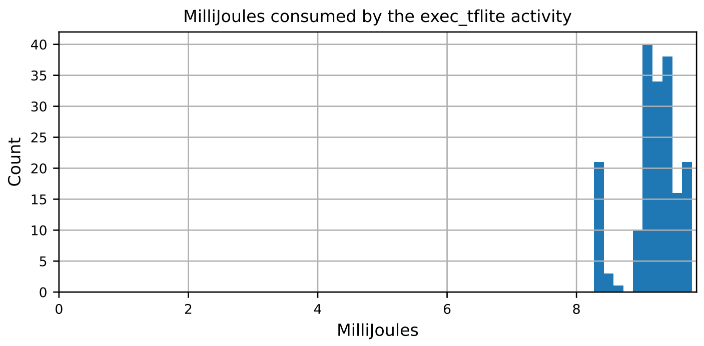

Figure 5.16: Energy consumption of individual TFLite execution activities

TensorFlow Lite Model Presence

Here we go through the fitting of eq. (5.8). We model the inclusion of a tflite model by setting TF = 1 and the exclusion of one by TF = 0. When TF = 0 no TFLite code is included in the TERRA firmware at all, and as such it's energy consumption is zero. The energy consumption data of the TFLite execution activity does not seem to follow a normal distribution. If we look at the energy consumption per device we see that the individual device data is normally distributed, but there is a device-specific centre. We see that in fig. 5.17 and also by looking at the raw data in table 5.12.

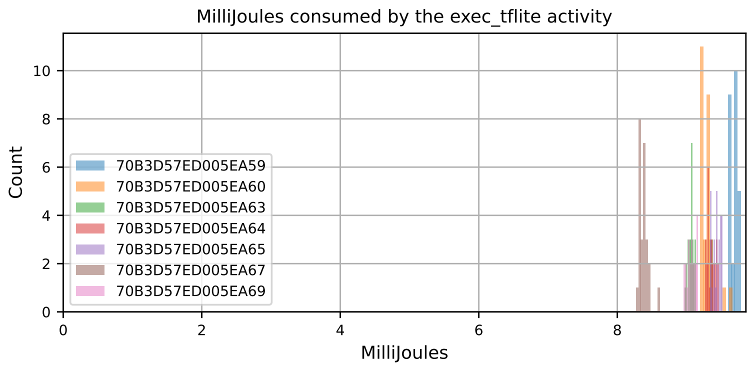

Figure 5.17: Energy consumption of individual TFLite execution activities per device

Node ID	Mean	Standard Deviation
70B3D57ED005EA59	0.009654	0.000118
70B3D57ED005EA60	0.009294	0.000099
70B3D57ED005EA63	0.009064	0.000040
70B3D57ED005EA64	0.009358	0.000060
70B3D57ED005EA65	0.009434	0.000059
70B3D57ED005EA67	0.008377	0.000070
70B3D57ED005EA69	0.009073	0.000064

Table 5.12: Mean energy consumption for TFLite activity per Node ID

We conclude that while there is a per device specificity, the energy consumption is overall quite low, and close to each other. As such an average of all 184 measured TFLite execution activities gives us:

$$C_{1,6} = 0.009194 \tag{5.25}$$

5.4.3 Model Results

To gauge the performance of the model (in eq. (5.12) defined in section 5.2.5), we evaluate it given a variety of different parameters in table 5.13. Here, the startup and steady-state cost can be seen given with varied QL, RR and TF.

Using eq. (5.13) f_{total}, we can also look at the model over multiple epochs. In fig. 5.18 we have visualised f_{total} over 25 epochs, and it shows what we expect. QL affects the starting cost, while RR affects the running cost.

We will now discuss these results.

Parameters			Values in J	
QL	RR	TF	Startup Phase	Steady-State Phase
16	1	0	6.362	2.032
64			6.886	2.040
128			7.584	2.051
256			8.980	2.072
16	1	0	6.362	2.032
	0.75		6.362	1.659
	0.5		6.362	1.285
	0.25		6.362	0.911
16	1	0	6.362	2.032
		1	6.362	2.042

Table 5.13: Values for Different QL, RR, and TF Configurations. Values have been rounded to 3 decimals for readability

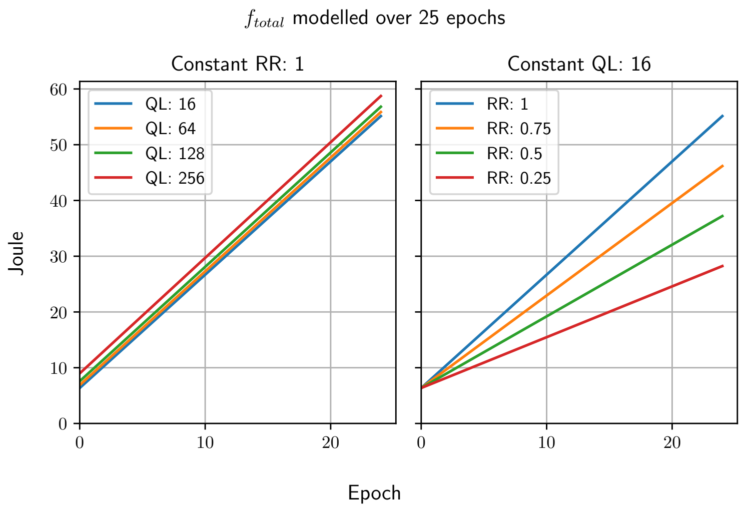

Figure 5.18: f_{total} modelled over 25 epochs with varying QL (left) or RR (right)

5.4.4 Model Discussion

We start by looking at the values in table 5.13.

For query length, we see a sharp increase in consumption for the startup phase and a very slight increase in the steady-state phase. This is what we expect, as the startup phase concerns itself with the actual network transfer and the steady-state phase with storing and execution, which are much less energy-costly.

For response rate, the startup phases are, as expected, equivalent, but the steady-state phase varies wildly, with the 0.25 response rate consuming less than half of the power of a response rate of 1. We see that halving the response rate leads to a $\approx 37\%$ decrease in the energy consumed per steady-state epoch.

For the TFLite model, we see no change in startup phase, while the steady-state phase sees a slight increase. This is also expected, but will, of course, differ with other models.

Compared to Baseline

Here we compare f_{total} with $f_{baseline\_total}$. The goal is to show the overhead cost of Terra, compared to the baseline that does not do query transfer at all. In fig. 5.19 we see this comparison. It is a relative comparison with $f_{baseline\_total}$ as a baseline of 1 and variations of f_{total} with different values of RR compared to it.

From fig. 5.19, it is evident that although the overhead incurred by receiving a query is substantial, nearly doubling the cost per epoch, the expense is recovered within fewer than 10 epochs, even with a relatively high response rate, resulting in net savings thereafter. We also see that with a response rate of 1 there are no savings at all. This makes sense as it consumes the same amount of energy per epoch as the baseline. So, if there is no aggregation, Terra does not provide any energy savings compared to a baseline and only incurs an additional cost. However, even with a relatively high RR we quickly recoup the overhead of Terra, and see only savings thereafter.

This answers research question **R3**, as when we see a decrease in the response rate, we see a comparable decrease in energy consumed per steady-state epoch due to the lack of network transmission. Although code offloading close to doubles the energy consumption of the startup epoch, as long as that code is allowed to execute for more than just 10 epochs in the pessimistic case, we see net savings of energy.

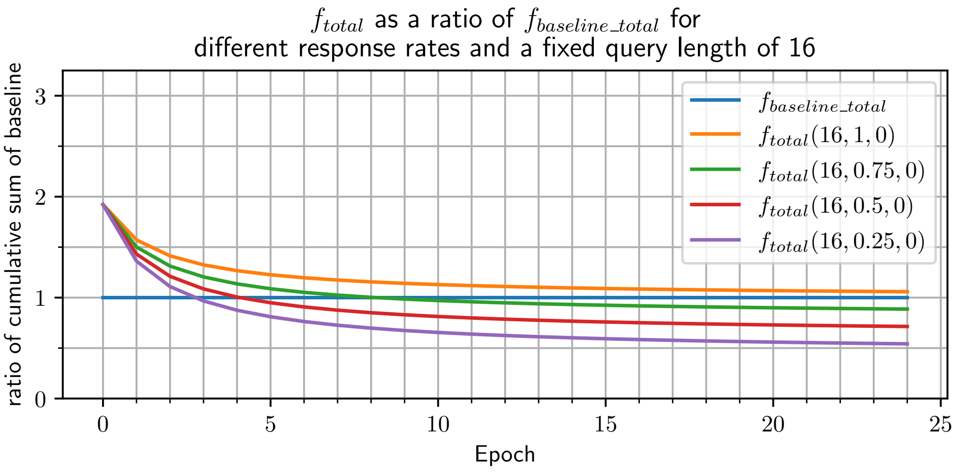

Figure 5.19: Comparison between $f_{baseline\_total}(16, 1, 0)$ and $f_{total}(16, 1, 0)$, $f_{total}(16, 0.75, 0)$, $f_{total}(16, 0.5, 0)$ and $f_{total}(16, 0.25, 0)$ relative to $f_{baseline\_total}(16, 1, 0)$.

5.4.5 Model Evaluation

Ideally we would now verify the model experimentally by executing similar experiments at a different site, or locally, and comparing the models performance to it. IoT-LAB does offer a different site with a few identical devices, in the French city of Lille, but as TTNMapper[80] shows there is no LoRaWAN gateway in the area, at least at the time of writing, so any comparison would have to be without network. Alternatively, the experiments could be carried out at the IT University of Copenhagen, which does have a couple of the same devices and attached sensor boards. However, they do not have the measurement hardware and test-setup of IoT-LAB again, making experimentation impossible.

So, in a lack of better methods, we here present two figures that compare the model performance to two RR experiments in figs. 5.20 and 5.21. These experiments were chosen because they reflect the most interesting behaviour of the model: The amortisation of the network transfer. The figures show both the cumulative energy cost and the individual devices in the experiment. For both experiments, we see the inherent variation in the energy consumption of individual devices and how the model follows the energy consumption. The difference between the model and the individual devices adds up over time, making the model more and more inaccurate for

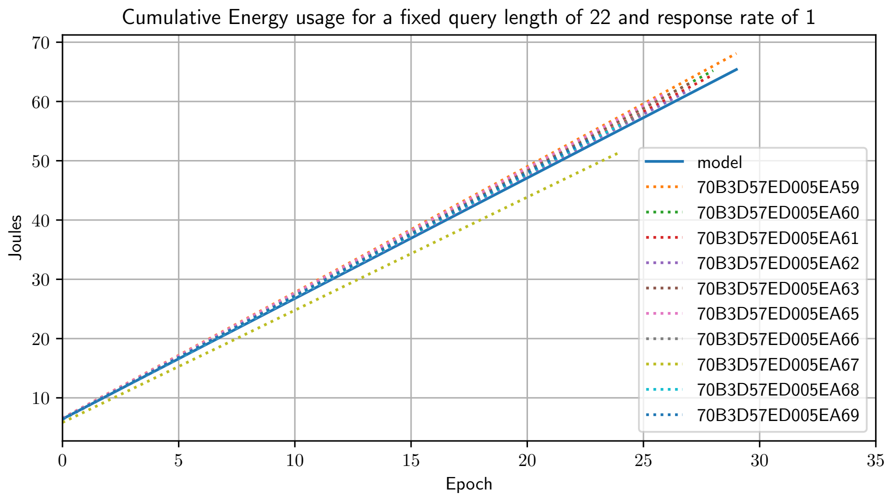

Figure 5.20: comparison of $f_{total}(22, 1, 0)$ with RR $= 1$ experiment

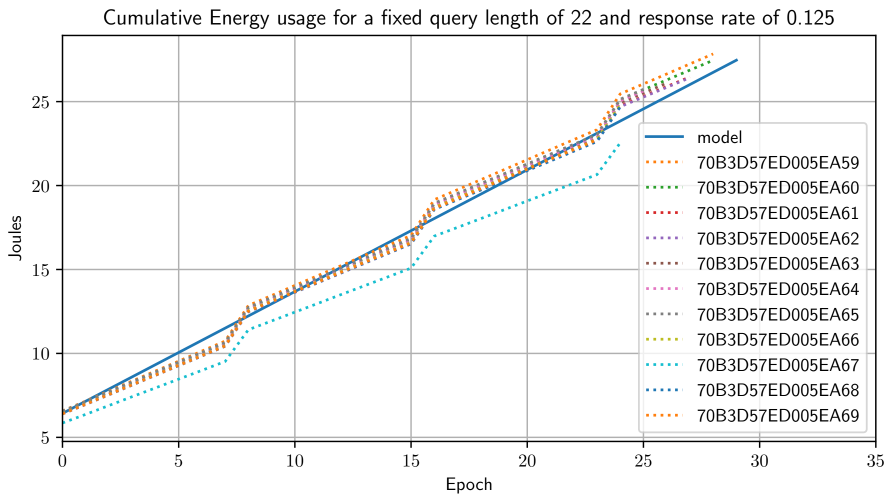

Figure 5.21: comparison of $f_{total}(22, 0.125, 0)$ with RR $= 0.125$ experiment

predictions for more epochs. However, even within the experiment this error is not more than 5 Joule, which is about the size of the startup epoch.

In summary: The model produced seems to follow the consumption of Terra accurately; however, device-specific variations make it hard to predict consumption over longer periods of time.

5.5 Future Work

The above work answers research question **R3** and therefore fulfils its purpose. However, we were not able to satisfactorily verify the model outside the IoT-LAB testbed which would be an immediate next step. An option here would be to verify the model locally using an oscilloscope. With a higher sample rate it is possible to more accurately measure the consumption of execution, which would improve our model.

One of the motivations for developing this cost model was also to enable NebulaStream to predict the energy cost of running a given query. This is useful as part of a query optimiser to be able to decide on effective placement of operators throughout the network given some optimisation strategy. For example, in TinyDB they optimise for power consumption [9].

A big future work is therefore improving this energy model and the use of it by integrating it as part of an operator placement strategy in NebulaStream. This would require integrating the measured energy usage as part of NebulaStream. Since these are device-specific, the calibration process should be streamlined to reduce the work required to derive a model for a specific device.

5.6 Summary

In this chapter, we introduce the cost model with which we evaluate TERRA as a series of constant and linear models with respect to the length of the query received, the rate of response transmitted, and whether or not a TFLite model was present and executed. The model splits epochs into a series of activities performed in either a startup or steady-state phase.

Through a series of experiments, each variable in the model is experimentally derived and then the model is analysed.

Our findings show that query length has an impact on several activities on both startup and steady-state energy consumption. This mostly concerns transmission in the startup phase and loading and storing the serialised query in the steady-state phase. The execution of the query was generally so fast that it is hard to conclude a correlation. This also means that the energy consumption of the execution was very low compared to other activities.

The response rate saw a significant decrease in amortised energy consumption when the response rate decreased.

The presence of a TFLite model slightly increased power consumption in the steady-state phase, but drawing wider conclusions from this is hard to do without testing other models.

Generally, the model answers research question **R3**. Code offloading does lead to a significant decrease in energy consumption if the offloaded query provides a decrease in response rate. If not, the added cost of transferring the query will never be recouped, and code offloading is only an added expense.

Throughout this whole testing process we used the IoT-LAB testbed, and before we end with a conclusion, we will go through some of the lessons learnt using a public testbed extensively. This is what we do in the next chapter.

Chapter 6

Lessons Learnt with IoT-LAB

To obtain the results of this thesis, a lot of work went into the use of the FIT IoT-LAB testbed. The existence of which made this thesis possible. It would not have been possible to perform the experiments needed and obtain these results without the hardware provided by IoT-LAB and the software stack to manage it. IoT-LAB provided boards with the appropriate hardware and network technology. In addition, they provide key monitoring hardware and infrastructure to facilitate the data collection that is the foundation of this thesis. They provide an easy-to-use web interface, with accompanying command-line tools to easily configure, provision, and start experiments. This was heavily used to write scripts and configurations that utilised those command-line tools to fully automate the experiment flow from configuration and compilation of Terra, through provisioning and uploading of firmware, to monitoring and fetching results.

With all that praise, there was also a series of annoyances or things to be aware of when using IoT-LAB. That is why we include this chapter to dwell on the experiences made and lessons learnt during this project, to serve as a guide for others.

We categorise our experience into three sections and cover them individually below. In section 6.1, we discuss considerations in testbed networking. In section 6.2, we explore hardware and boards, and in section 6.3, we focus on power monitoring.

6.1 Networking and IoT-LAB

On the front page of IoT-LAB[1] they write:

[1] `https://www.iot-lab.info/`

> IoT-LAB provides a facility suitable for testing networking with small wireless sensor devices and heterogeneous communicating objects.

Thus, their fundamental purpose is network research in an IoT context. This purpose they fulfil by providing access to boards with support for various network technologies. Access in numbers that individual users rarely have. However, when using a remote testbed, a user loses physical access to the hardware and the space in which they exist. The user cannot, outside of the tools of the testbed, see the physical layout of the hardware and the distances, or obstacles, between them. And the user also cannot change position or hardware.

IoT-LAB offered at one point access to mobile nodes that allowed you to control and change the position of some specific hardware, but this is not available today [42, 43].

The consequence of this is that network connectivity issues are difficult to deal with. As a user, you cannot visually see or move obstructions in the environment. You can gauge obstacles and layout of nodes in a room in the IoT-LAB documentation through descriptions or possibly floor plans, and you can query each node for their position in a 3D coordinate system and then determine their relative positions, but this method is cumbersome and lacks the intuition of physical presence in the room. Moreover, it does not alter the fundamental issue: nodes cannot be repositioned to handle possible dead zones; you must use another node.

All of this adds difficulty when you are dealing with network protocols that could be already unreliable.

These problems also apply to LoRaWAN gateways. Since the experiments done use LoRaWAN they need a gateway to be present and active near the nodes. IoT-LAB is kind enough to provide such a gateway for some of their sites with LoRa-capable hardware, but it is not part of their testbed infrastructure. This means that you cannot access it, be it to view logs or reboot. You also do not know the positions of it, with the same problems described as above. During experiments, we frequently encounter nodes that neither receives nor transmits messages, and not having access to the gateway logs makes it difficult to gauge if the problem exists between node and gateway, or between gateway and core network.

Lack of access to reboot the gateway also hinders experiments as the gateway seems to be quite unstable and often goes down or becomes unresponsive. This is addressed often within a week by contacting IoT-LAB through their mailing list and waiting for personnel to bring the gateway back up, but in periods of vacation or for other reasons we have experienced downtimes of months where no experimentation was possible.

For the reasons mentioned above, we recommend having at least a couple of identical devices and a small gateway locally at hand to develop and test your experiment setup and then use the testbed, when available, to do larger experiments.

6.2 Devices

Programmatic and remote access to upload firmware and communicate over serial with devices is generally a breeze on IoT-LAB, and it is quite impressive how simple and accessible they have done a process that is often fraught with annoying toolchains and hardware support. However, this also comes with a cost. When problems do arise, there is no access to the raw error logs and it is difficult to debug or restart. We saw cases of devices that would repeatedly fail to flash, but there was no indication as to why they failed.

Often devices can also fail silently in the sense that it seems like firmware got uploaded and the device is running it, but it is not as revealed by a lack of expected serial output. IoT-LAB does have a device status that could indicate whether or not devices are failing or otherwise unavailable, but this is not always properly set, which can cause you to be repeatedly assigned devices that silently fail.

Speaking of availability of devices, IoT-LAB as a resource provided by universities is often used by those universities for teaching or research, and this, of course, has priority. However, there is no central place where these planned reservations are announced, and instead they are communicated through the same general mailing list that users use to ask questions or report problems. While a minor issue, this does make these windows harder to track as they easily disappear in one's inbox.

6.3 Power Monitoring

The IoT-LAB monitoring infrastructure is what allows us to collect these detailed power measurements that are the basis of the experiments we run. However, the instrumentation library and file format used by IoT-LAB is unmaintained, which is problematic. The website of the library does not exist any more[2] so web archives had to be used to find proper documentation, and even then it is mostly on the instrumentation library itself and not on the file format. Fortunately, it is mostly formatted as a CSV with modifications, so reverse engineering the format is easy. IoT-LAB does provide some facilities for reading and visualising the data files, and these are written in Python[3] and can also be easily reverse-engineered[43, tools/Consumption Monitoring]. However, there is no central description of the data format used by IoT-LAB.

Depending on your monitoring configuration, these data files could quickly take up a lot of space on the remote machine that IoT-LAB uses to control the testbed. There is no warning, which caused an unlucky episode in which part of the testbed was down due to a full hard drive. Luckily, this was quickly resolved, but did cause issues afterwards with symbolic links due to the way it was cleared. So, we recommend moving the data from the testbed to local storage right after the experiments.

6.4 Summary

We aim to highlight the exceptional asset that IoT-LAB represents for research. Its extensive range of features, diverse configuration possibilities, and the simplicity associated with its free access are unparalleled in the academic world. However, as frequent users of this testbed, we wish to discuss certain challenges that arise and the precautions that should be taken before and during its use. One important insight is that dealing with networking and embedded hardware can be challenging due to elusive bugs or connectivity issues; having direct access to the hardware is crucial in these situations. Consequently, we recommend acquiring some of the boards and hardware intended for use at IoT-LAB to conduct small-scale local experiments to resolve any issues. Once you are prepared, you can scale up on the testbed. This approach guarantees a more seamless and effective experience.

We provide a summary of the above points in table 6.1.

In the next chapter, we will conclude the thesis.

[2]`https://oml.mytestbed.net/` is down, but versions can be found using `https://web.archive.org/`

[3]Source code can be found here: `https://github.com/iot-lab/oml-plot-tools`

Category	**Description**
Networking	No control over, or access to, physical environment leads to hard to debug network issues No control over gateway leads to increased downtime No access to gateway prevents efficient debugging
Devices	Devices can fail silently, and just not be flashable without any errors Devices can give false or hard to read error messages
Monitoring	Monitoring is performed using an unmaintained library that is quite difficult to find documentation for
General	Site-wide communication is done via mailing lists also in use by users There is no limit or warnings on data storage, which can lead to platform failure and users inadvertently crippling their own access

Table 6.1: Summary of key issues

Chapter 7

Conclusion

In the beginning of this thesis, we defined the following hypothesis and research questions:

> *Code offloading to sensor devices in modern sensor network databases leads to a significant reduction in energy consumption.*

R1 What is a modern sensor network database?

R2 How do we support code offloading in such a database?

R3 How does code offloading lead to a significant reduction in energy consumption?

In chapter 2 we answer **R1** by defining a modern sensor database as a database that takes advantage of computing resources at the edge, fog, and compute layers to reduce network congestion, energy usage, and increase data privacy.

In chapter 4 we introduce Terra to answer **R2**. Our system enables another sensor network database, in this case NebulaStream, to offload computation on to sensor devices, thereby making it modern. We do this to exploit the ability of NebulaStream to use fog and cloud resources and to parse and optimise queries. What NebulaStream lacks is the ability to push computation to the very edge. We adapt NebulaStream to send applicable partial queries to Terra in a transparent manner, by capturing queries early in the NebulaStream pipeline and eagerly extracting operators to offload to Terra. Upon receiving these operators, Terra then executes them and transmits the results back. In particular, when aggregation or filtering queries are executed and no results are present, nothing is reported, leading to substantial energy savings.

We verify this claim experimentally in chapter 5, where we also answer **R3**. We do this by defining an energy cost model that describes the energy consumption of the activities Terra undertakes for each epoch. The energy cost model is a series of constant and linear models that describe the energy cost of each activity. We then define and execute 11 experiments on the IoT-LAB testbed with unique queries. We record detailed power consumption measurements for each experiment at a sampling rate high enough to extract the energy consumption for each activity. With these measurements in the energy cost model, we conclude that while energy consumption is correlated with query length and the presence of a TFLite model, the biggest impact is the response rate. Receiving a query almost doubles the cost of a Terra epoch, while halving the response rate reduces the consumption by $\approx 37\%$. These savings come purely from the energy that is conserved by avoiding unnecessary network transmissions. The consequence of this is that even with a high response rate, the added overhead of code offloading is recouped within approximately 10 epochs.

Here we also note that the use of LoRaWAN and The Things Network as a provider does increase consumption of network transfer compared to other LoRaWAN networks due to its higher receive window delays. This affects both code offloading and result transmission. As such, we can answer **R3** and the hypothesis:

If code offloading leads to a decrease in response rate by containing aggregating or filtering operators, it can lead to a significant reduction in energy consumption by avoiding unnecessary network transfers.

Bibliography

[1] Juan José López Escobar, Rebeca P. Díaz Redondo, and Felipe Gil-Castiñeira. “In-depth analysis and open challenges of Mist Computing”. In: *Journal of Cloud Computing* 11.1 (Nov. 19, 2022), p. 81. ISSN: 2192-113X. DOI: 10.1186/s13677-022-00354-x. URL: https://doi.org/10.1186/s13677-022-00354-x (visited on 11/27/2024) (cit. on pp. 1, 13).

[2] Steffen Zeuch et al. “The NebulaStream Platform: Data and Application Management for the Internet of Things”. In: CIDR 2020. Jan. 2020, p. 11 (cit. on pp. 2, 15, 27, 37, 39, 69).

[3] Peter J. Denning. “ACM president’s letter: performance analysis: experimental computer science as its best”. In: *Commun. ACM* 24.11 (Nov. 1, 1981), pp. 725–727. ISSN: 0001-0782. DOI: 10.1145/358790.358791. URL: https://dl.acm.org/doi/10.1145/358790.358791 (visited on 11/27/2024) (cit. on p. 2).

[4] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. “Towards Sensor Database Systems”. In: *Mobile Data Management.* Ed. by Kian-Lee Tan, Michael J. Franklin, and John Chi-Shing Lui. Red. by Gerhard Goos, Juris Hartmanis, and Jan Van Leeuwen. Vol. 1987. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 3–14. DOI: 10.1007/3-540-44498-X_1. URL: http://link.springer.com/10.1007/3-540-44498-X_1 (visited on 10/10/2024) (cit. on pp. 5, 6, 9).

[5] R. Govindan et al. “The Sensor Network as a Database”. In: 2002. URL: https://www.semanticscholar.org/paper/The-Sensor-Network-as-a-Database-Govindan-Hellerstein/6bd5a40da1a5bdb83d085e9e4c9bf14aa5336130 (visited on 10/11/2024) (cit. on pp. 5, 6).

[6] Jinbao Li, Zhipeng Cai, and Jianzhong Li. “Data Management in Sensor Networks”. In: *Wireless Sensor Networks and Applications*. Ed. by Yingshu Li, My T. Thai, and Weili Wu. Series Title: Signals and Communication Technology. Boston, MA: Springer US, 2008, pp. 287–330. ISBN: 978-0-387-49592-7. DOI: 10.1007/978-0-387-49592-7_12. URL: http://link.springer.com/10.1007/978-0-387-49592-7_12 (visited on 10/10/2024) (cit. on pp. 5, 6).

[7] Abderrahmen Belfkih, Claude Duvallet, and Bruno Sadeg. “A survey on wireless sensor network databases”. In: *Wireless Networks* 25.8 (Nov. 2019), pp. 4921–4946. ISSN: 1022-0038, 1572-8196. DOI: 10.1007/s11276-019-02070-y. URL: http://link.springer.com/10.1007/s11276-019-02070-y (visited on 10/10/2024) (cit. on pp. 6, 7, 9, 11).

[8] Yong Yao and Johannes Gehrke. “The cougar approach to in-network query processing in sensor networks”. In: *SIGMOD Rec.* 31.3 (Sept. 1, 2002), pp. 9–18. ISSN: 0163-5808. DOI: 10.1145/601858.601861. URL: https://doi.org/10.1145/601858.601861 (visited on 10/12/2024) (cit. on p. 9).

[9] Samuel R. Madden et al. “TinyDB: an acquisitional query processing system for sensor networks”. In: *ACM Transactions on Database Systems* 30.1 (Mar. 2005), pp. 122–173. ISSN: 0362-5915, 1557-4644. DOI: 10.1145/1061318.1061322. URL: https://dl.acm.org/doi/10.1145/1061318.1061322 (visited on 05/01/2023) (cit. on pp. 9, 10, 26, 76, 115).

[10] Giuseppe Amato, Stefano Chessa, and Claudio Vairo. “MaD-WiSe: a distributed stream management system for wireless sensor networks”. In: *Software: Practice and Experience* 40.5 (Apr. 25, 2010), pp. 431–451. ISSN: 0038-0644, 1097-024X. DOI: 10.1002/spe.965. URL: https://onlinelibrary.wiley.com/doi/10.1002/spe.965 (visited on 10/15/2024) (cit. on pp. 9, 11).

[11] Philippe Bonnet et al. *Query Processing in a Device Database System*. Cornell University, Oct. 1999. URL: https://hdl.handle.net/1813/7429 (visited on 10/12/2024) (cit. on p. 9).

[12] P. Levis et al. "TinyOS: An Operating System for Sensor Networks". In: *Ambient Intelligence*. Ed. by Werner Weber, Jan M. Rabaey, and Emile Aarts. Berlin, Heidelberg: Springer, 2005, pp. 115–148. ISBN: 978-3-540-27139-0. DOI: 10.1007/3-540-27139-2_7. URL: https://doi.org/10.1007/3-540-27139-2_7 (visited on 10/12/2024) (cit. on pp. 9, 10, 26, 58).

[13] David Gay et al. "The nesC language: A holistic approach to networked embedded systems". In: *SIGPLAN Not.* 38.5 (May 9, 2003), pp. 1–11. ISSN: 0362-1340. DOI: 10.1145/780822.781133. URL: https://doi.org/10.1145/780822.781133 (visited on 10/14/2024) (cit. on p. 10).

[14] Philip Levis and David Culler. "Maté: A tiny virtual machine for sensor networks". In: *ACM Sigplan Notices* 37.10 (2002). Publisher: ACM New York, NY, USA, pp. 85–95 (cit. on p. 10).

[15] G. Amato, P. Baronti, and S. Chessa. "MaD-WiSe: Programming and Accessing Data in a Wireless Sensor Networks". In: *EUROCON 2005 - The International Conference on "Computer as a Tool"*. EUROCON 2005 - The International Conference on "Computer as a Tool". Belgrade, Serbia and Montenegro: IEEE, 2005, pp. 1846–1849. ISBN: 978-1-4244-0049-2. DOI: 10.1109/EURCON.2005.1630339. URL: http://ieeexplore.ieee.org/document/1630339/ (visited on 10/15/2024) (cit. on p. 11).

[16] Daniel J. Abadi et al. "Aurora: a new model and architecture for data stream management". In: *The VLDB Journal The International Journal on Very Large Data Bases* 12.2 (Aug. 1, 2003), pp. 120–139. ISSN: 1066-8888, 0949-877X. DOI: 10.1007/s00778-003-0095-z. URL: http://link.springer.com/10.1007/s00778-003-0095-z (visited on 10/09/2024) (cit. on p. 12).

[17] Matthias J. Sax et al. "Streams and Tables: Two Sides of the Same Coin". In: *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*. BIRTE '18. New York, NY, USA: Association for Computing Machinery, Aug. 27, 2018, pp. 1–10. ISBN: 978-1-4503-6607-6. DOI: 10.1145/3242153.3242155. URL: https://dl.acm.org/doi/10.1145/3242153.3242155 (visited on 10/17/2024) (cit. on p. 12).

[18] *Apache Storm*. URL: https://storm.apache.org/ (visited on 10/17/2024) (cit. on p. 12).

[19] Paris Carbone et al. “Apache Flink™: Stream and Batch Processing in a Single Engine”. In: () (cit. on p. 12).

[20] Michael Armbrust et al. “Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark”. In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD ’18. New York, NY, USA: Association for Computing Machinery, May 27, 2018, pp. 601–613. ISBN: 978-1-4503-4703-7. DOI: 10.1145/3183713.3190664. URL: https://doi.org/10.1145/3183713.3190664 (visited on 10/17/2024) (cit. on p. 12).

[21] Alessio Botta et al. “Integration of Cloud computing and Internet of Things: A survey”. In: *Future Generation Computer Systems* 56 (Mar. 2016), pp. 684–700. ISSN: 0167739X. DOI: 10.1016/j.future.2015.09.021. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167739X15003015 (visited on 10/15/2024) (cit. on p. 13).

[22] Abhishek Hazra et al. “Fog computing for next-generation Internet of Things: Fundamental, state-of-the-art and research challenges”. In: *Computer Science Review* 48 (May 1, 2023), p. 100549. ISSN: 1574-0137. DOI: 10.1016/j.cosrev.2023.100549. URL: https://www.sciencedirect.com/science/article/pii/S1574013723000163 (visited on 10/17/2024) (cit. on p. 13).

[23] Evangelos K. Markakis et al. “EXEGESIS: Extreme Edge Resource Harvesting for a Virtualized Fog Environment”. In: *IEEE Communications Magazine* 55.7 (July 2017). Conference Name: IEEE Communications Magazine, pp. 173–179. ISSN: 1558-1896. DOI: 10.1109/MCOM.2017.1600730. URL: https://ieeexplore.ieee.org/document/7981547 (visited on 11/27/2024) (cit. on p. 14).

[24] Ivan Zyrianoff et al. “Architecting and Deploying IoT Smart Applications: A Performance–Oriented Approach”. In: *Sensors* 20.1 (Jan. 2020). Number: 1 Publisher: Multidisciplinary Digital Publishing Institute, p. 84. ISSN: 1424-8220. DOI: 10.3390/s20010084. URL: https://www.mdpi.com/1424-8220/20/1/84 (visited on 11/27/2024) (cit. on p. 14).

[25] Yugo Nakamura et al. “Design and Implementation of Middleware for IoT Devices toward Real-Time Flow Processing”. In: *2016 IEEE 36th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. 2016 IEEE 36th International Conference on Distributed Computing Systems Workshops (ICDCSW). ISSN: 2332-5666. June 2016, pp. 162–167. DOI: 10.1109/ICDCSW.2016.37. URL:

https://ieeexplore.ieee.org/document/7756225 (visited on 11/27/2024) (cit. on p. 14).

[26] Zhitao Shen et al. “CSA: Streaming Engine for Internet of Things”. In: () (cit. on p. 14).

[27] Nicolas Tsiftes and Adam Dunkels. “A database in every sensor”. In: *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*. SenSys ’11: The 9th ACM Conference on Embedded Network Sensor Systems. Seattle Washington: ACM, Nov. 2011, pp. 316–332. ISBN: 978-1-4503-0718-5. DOI: 10.1145/2070942.2070974. URL: https://dl.acm.org/doi/10.1145/2070942.2070974 (visited on 10/09/2024) (cit. on pp. 14, 41).

[28] *NebulaStream*. Nebula Stream. Nov. 12, 2020. URL: https://docs.nebula.stream/docs/nebulastream/generaloverview/ (visited on 10/18/2024) (cit. on pp. 16, 34).

[29] Free Software Foundation. *GCC, the GNU Compiler Collection - GNU Project*. GCC, the GNU Compiler Collection. URL: https://gcc.gnu.org/ (visited on 10/08/2024) (cit. on p. 16).

[30] LLVM Developer Group. *Clang C Language Family Frontend for LLVM*. Clang: a C language family frontend for LLVM. URL: https://clang.llvm.org/ (visited on 10/08/2024) (cit. on p. 16).

[31] Bishal Ranjan Swain et al. “Rise of Fluid Computing A Collective Effort Of Mist, Fog and Cloud”. In: *International Journal of Computer Sciences and Engineering* 7.4 (Apr. 30, 2019), pp. 62–69. ISSN: 23472693. DOI: 10.26438/ijcse/v7i4.6269. URL: http://www.ijcseonline.org/full_paper_view.php?paper_id=3996 (visited on 12/13/2024) (cit. on p. 16).

[32] Vasilios A. Orfanos et al. “A Comprehensive Review of IoT Networking Technologies for Smart Home Automation Applications”. In: *Journal of Sensor and Actuator Networks* 12.2 (Apr. 2023). Number: 2 Publisher: Multidisciplinary Digital Publishing Institute, p. 30. ISSN: 2224-2708. DOI: 10.3390/jsan12020030. URL: https://www.mdpi.com/2224-2708/12/2/30 (visited on 10/21/2024) (cit. on p. 18).

[33] M. Mroue et al. “LPWAN Technologies in Smart Cities: A Comparative Analysis of LoRa, Sigfox, and LTE-M”. In: *Information System Design: Communication Networks and IoT*. Ed. by Vikrant Bhateja et al. Singapore: Springer Nature, 2024, pp. 219–231. ISBN: 978-981-9748-95-2. DOI: 10.1007/978-981-97-4895-2_18 (cit. on p. 18).

[34] Kais Mekki et al. “Overview of Cellular LPWAN Technologies for IoT Deployment: Sigfox, LoRaWAN, and NB-IoT”. In: *2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. 2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops). Mar. 2018, pp. 197–202. DOI: 10.1109/PERCOMW.2018.8480255. URL: https://ieeexplore.ieee.org/document/8480255 (visited on 11/30/2024) (cit. on p. 18).

[35] Ritesh Kumar Singh et al. “Energy Consumption Analysis of LPWAN Technologies and Lifetime Estimation for IoT Application”. In: *Sensors* 20.17 (Jan. 2020). Number: 17 Publisher: Multidisciplinary Digital Publishing Institute, p. 4794. ISSN: 1424-8220. DOI: 10.3390/s20174794. URL: https://www.mdpi.com/1424-8220/20/17/4794 (visited on 11/30/2024) (cit. on p. 18).

[36] LoRa Alliance. *LoRaWAN® L2 1.0.4 specification*. manual. 2020. URL: https://resources.lora-alliance.org/document/ts001-1-0-4-lorawan-l2-1-0-4-specification (cit. on pp. 18, 64).

[37] Stefano Milani et al. *Edge2lora: Enabling Edge Computing on Long-Range Wide-Area Internet of Things*. 2024. DOI: 10.2139/ssrn.4821982. URL: https://www.ssrn.com/abstract=4821982 (visited on 06/25/2024) (cit. on p. 20).

[38] The Things Network. *The Things Network LoRaWAN® Documentation*. The Things Network. URL: https://www.thethingsnetwork.org/docs/lorawan/ (visited on 10/01/2024) (cit. on pp. 21, 54, 82).

[39] LoRa Alliance. *LoRaWAN regional parameters v1.0.2*. manual. Feb. 2017. URL: https://lora-alliance.org/resource_hub/lorawan-regional-parameters-v1-0-2rb/ (cit. on pp. 21, 68, 81, 82).

[40] LoRa Alliance. *LoRaWAN® specification v1.0.2*. manual. 2016. URL: https://resources.lora-alliance.org/document/lorawan-specification-v1-0-2 (cit. on pp. 21, 64).

[41] Raj Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. Nachdr. Wiley professional computing. New York: Wiley, 1991. 685 pp. ISBN: 978-0-471-50336-1 (cit. on pp. 22, 77, 88, 93).

[42] Cédric Adjih et al. “FIT IoT-LAB: A Large Scale Open Experimental IoT Testbed”. In: IEEE World Forum on Internet of Things (IEEE WF-IoT). Dec. 14, 2015. URL: https://inria.hal.science/hal-01213938 (visited on 09/27/2024) (cit. on pp. 23, 118).

[43] FIT IoT-LAB. *IoT-LAB documentation*. FIT IoT-LAB. URL: https://iot-lab.github.io/docs/getting-started/introduction/ (visited on 10/01/2024) (cit. on pp. 23, 78, 118, 120).

[44] Texas Instruments. *INA226 36V, 16-bit, ultra-precise I2C output current, voltage, and power monitor with alert*. 2024. URL: https://www.ti.com/lit/ds/symlink/ina226.pdf (cit. on pp. 23, 86).

[45] Rafael Lajara, José Pelegrí-Sebastiá, and Juan J. Perez Solano. “Power Consumption Analysis of Operating Systems for Wireless Sensor Networks”. In: *Sensors* 10.6 (June 2010). Number: 6 Publisher: Molecular Diversity Preservation International, pp. 5809–5826. ISSN: 1424-8220. DOI: 10.3390/s100605809. URL: https://www.mdpi.com/1424-8220/10/6/5809 (visited on 11/28/2024) (cit. on p. 26).

[46] A. Dunkels, B. Gronvall, and T. Voigt. “Contiki - a lightweight and flexible operating system for tiny networked sensors”. In: *29th Annual IEEE International Conference on Local Computer Networks*. 29th Annual IEEE International Conference on Local Computer Networks. ISSN: 0742-1303. Nov. 2004, pp. 455–462. DOI: 10.1109/LCN.2004.38. URL: https://ieeexplore.ieee.org/document/1367266/?arnumber=1367266 (visited on 11/28/2024) (cit. on p. 26).

[47] Adam Dunkels et al. “Software-based on-line energy estimation for sensor nodes”. In: *Proceedings of the 4th workshop on Embedded networked sensors*. EmNets ’07. New York, NY, USA: Association for Computing Machinery, June 25, 2007, pp. 28–32. ISBN: 978-1-59593-694-3. DOI: 10.1145/1278972.1278979. URL: https://dl.acm.org/doi/10.1145/1278972.1278979 (visited on 11/28/2024) (cit. on p. 26).

[48] Xiaofan Jiang et al. “Micro Power Meter for Energy Monitoring of Wireless Sensor Networks at Scale”. In: *2007 6th International Symposium on Information Processing in Sensor Networks*. 2007 6th International Symposium on Information Processing in Sensor Networks. Apr. 2007, pp. 186–195. DOI: 10.1109/IPSN.2007.4379678. URL: https://ieeexplore.ieee.org/document/4379678 (visited on 11/28/2024) (cit. on p. 26).

[49] Robert Hartung, Ulf Kulau, and Lars Wolf. "Distributed Energy Measurement in WSNs for Outdoor Applications". In: *2016 13th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*. 2016 13th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON). June 2016, pp. 1–9. DOI: 10.1109/SAHCN.2016.7732983. URL: https://ieeexplore.ieee.org/document/7732983 (visited on 11/28/2024) (cit. on p. 26).

[50] Dan O'Keeffe, Theodoros Salonidis, and Peter Pietzuch. "Frontier: resilient edge processing for the internet of things". In: *Proceedings of the VLDB Endowment* 11.10 (June 2018), pp. 1178–1191. ISSN: 2150-8097. DOI: 10.14778/3231751.3231767. URL: https://dl.acm.org/doi/10.14778/3231751.3231767 (visited on 10/17/2024) (cit. on p. 27).

[51] Donald Kossmann. "The state of the art in distributed query processing". In: *ACM Comput. Surv.* 32.4 (Dec. 1, 2000), pp. 422–469. ISSN: 0360-0300. DOI: 10.1145/371578.371598. URL: https://dl.acm.org/doi/10.1145/371578.371598 (visited on 10/15/2024) (cit. on pp. 27, 29).

[52] Feilong Liu et al. *To Ship or Not to (Function) Ship (Extended version)*. arXiv:1807.11149. version: 1 type: article. arXiv, July 29, 2018. DOI: 10.48550/arXiv.1807.11149. arXiv: 1807.11149[cs]. URL: http://arxiv.org/abs/1807.11149 (visited on 10/15/2024) (cit. on pp. 28, 29).

[53] *nebulastream/nebulastream: NebulaStream - Data Management for the Internet of Things*. In collab. with BIFOLD, DIMA, and DFKI IAM. Note: Not made public yet, but will be publicized at a later date. URL: https://github.com/nebulastream/nebulastream (visited on 10/23/2024) (cit. on p. 35).

[54] Google. *Protocol Buffers*. Protocol Buffers Documentation. URL: https://protobuf.dev/ (visited on 10/22/2024) (cit. on p. 38).

[55] Juan Cruz Viotti and Mital Kinderkhedia. *A Benchmark of JSON-compatible Binary Serialization Specifications*. Jan. 9, 2022. DOI: 10.48550/arXiv.2201.03051. arXiv: 2201.03051. URL: http://arxiv.org/abs/2201.03051 (visited on 10/22/2024) (cit. on p. 38).

[56] AMRIT KUMAR BISWAL and OBADA AL MALLAH. "Analytical assessment of binary data serialization techniques in IoT context (evaluating protocol buffers, flat buffers, message pack, and BSON for sensor nodes)". In: (Dec. 17, 2019). Accepted: 2020-01-07T08:40:39Z Publisher: Italy. URL: https://www.politesi.polimi.it/handle/10589/150617 (visited on 10/22/2024) (cit. on p. 38).

[57] Amandeep Kaur et al. "A Literature Review on Device-to-Device Data Exchange Formats for IoT Applications". In: *JOURNAL OF INTELLIGENT SYSTEMS AND COMPUTING* 1.1 (Dec. 31, 2020). Number: 1, pp. 1–10. ISSN: 2976-8098. DOI: https://n2t.net/ark:/47543/JISCOM2020.v1i1.a4. URL: https://scienceandtech.co.uk/journals/index.php/jiscom/article/view/4 (visited on 10/22/2024) (cit. on p. 38).

[58] *micropython/micropython*. original-date: 2013-12-20T11:47:07Z. Nov. 1, 2024. URL: https://github.com/micropython/micropython (visited on 11/01/2024) (cit. on pp. 40, 57).

[59] *bytecodealliance/wasm-micro-runtime*. original-date: 2019-05-02T21:32:09Z. Nov. 29, 2024. URL: https://github.com/bytecodealliance/wasm-micro-runtime (visited on 11/30/2024) (cit. on p. 40).

[60] Volodymyr Shymanskyy. *wasm3/wasm3*. original-date: 2019-10-01T17:06:03Z. Nov. 30, 2024. URL: https://github.com/wasm3/wasm3 (visited on 11/30/2024) (cit. on p. 40).

[61] Emmanuel Baccelli et al. "Scripting Over-The-Air: Towards Containers on Low-end Devices in the Internet of Things". In: *2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. 2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops). Mar. 2018, pp. 504–507. DOI: 10.1109/PERCOMW.2018.8480277. URL: https://ieeexplore.ieee.org/document/8480277/?arnumber=8480277 (visited on 11/30/2024) (cit. on p. 40).

[62] Koen Zandberg et al. "Femto-containers: lightweight virtualization and fault isolation for small software functions on low-power IoT microcontrollers". In: *Proceedings of the 23rd ACM/IFIP International Middleware Conference*. Middleware '22: 23rd International Middleware Conference. Quebec QC Canada: ACM, Nov. 7, 2022, pp. 161–173. ISBN: 978-1-4503-9340-9. DOI: 10.1145/3528535.3565242. URL:

https://dl.acm.org/doi/10.1145/3528535.3565242 (visited on 11/30/2024) (cit. on p. 40).

[63] Ana Cristina Franco da Silva and Pascal Hirmer. “Models for Internet of Things Environments—A Survey”. In: *Information* 11.10 (Oct. 2020). Number: 10 Publisher: Multidisciplinary Digital Publishing Institute, p. 487. ISSN: 2078-2489. DOI: 10.3390/info11100487. URL: https://www.mdpi.com/2078-2489/11/10/487 (visited on 10/24/2024) (cit. on p. 41).

[64] Google. *LiteRT for Microcontrollers | Google AI Edge*. Google AI for Developers. URL: https://ai.google.dev/edge/litert/microcontrollers/overview (visited on 10/25/2024) (cit. on p. 42).

[65] Laurits Bonde Henriksen and Markus Kildebæk Raun Johansen. “Exploring the Space of Energy Constrained Devices Using NebulaStream”. Bachelors Thesis. IT University of Copenhagen: IT University of Copenhagen, May 2023. 40 pp. (cit. on p. 53).

[66] ChirpStack. *ChirpStack open-source LoRaWAN Network Server*. ChirpStack, open-source LoRaWAN Network Server. URL: https://www.chirpstack.io/ (visited on 11/12/2024) (cit. on p. 54).

[67] Mahmoud H. Qutqut et al. “Comprehensive survey of the IoT open-source OSs”. In: *IET Wireless Sensor Systems* 8.6 (2018). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1049/iet-wss.2018.5033, pp. 323–339. ISSN: 2043-6394. DOI: 10.1049/iet-wss.2018.5033. URL: https://onlinelibrary.wiley.com/doi/abs/10.1049/iet-wss.2018.5033 (visited on 11/01/2024) (cit. on p. 58).

[68] Farhana Javed et al. “Internet of Things (IoT) Operating Systems Support, Networking Technologies, Applications, and Challenges: A Comparative Review”. In: *IEEE Communications Surveys & Tutorials* 20.3 (2018). Conference Name: IEEE Communications Surveys & Tutorials, pp. 2062–2100. ISSN: 1553-877X. DOI: 10.1109/COMST.2018.2817685. URL: https://ieeexplore.ieee.org/abstract/document/8320780 (visited on 11/01/2024) (cit. on p. 58).

[69] FreeRTOS. *FreeRTOS*. URL: https://freertos.org (visited on 11/05/2024) (cit. on p. 58).

[70] Emmanuel Baccelli et al. “RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT”. In: *IEEE Internet of Things Journal* 5.6 (Dec. 2018), pp. 4428–4440. ISSN: 2327-4662, 2372-2541. DOI: 10.1109/JIOT.2018.2815038. URL: https://ieeexplore.ieee.org/document/8315125/ (visited on 05/04/2023) (cit. on pp. 58, 59, 62).

[71] Zephyr Project. *Zephyr Project Overview*. Oct. 17, 2024. URL: https://www.zephyrproject.org/wp-content/uploads/2024/10/Zephyr-Overview-20241017.pdf (visited on 11/04/2024) (cit. on p. 58).

[72] RIOT. *RIOT - Boards*. URL: https://www.riot-os.org/boards.html (visited on 11/05/2024) (cit. on p. 59).

[73] Rafael Raymundo Belleza and Edison de Freitas Pignaton. “Performance study of real-time operating systems for internet of things devices”. In: *IET Software* 12.3 (2018). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1049/iet-sen.2017.0048, pp. 176–182. ISSN: 1751-8814. DOI: 10.1049/iet-sen.2017.0048. URL: https://onlinelibrary.wiley.com/doi/abs/10.1049/iet-sen.2017.0048 (visited on 11/05/2024) (cit. on p. 59).

[74] Petteri Aimonen. *nanopb*. original-date: 2015-04-29T17:21:37Z. Nov. 5, 2024. URL: https://github.com/nanopb/nanopb (visited on 11/05/2024) (cit. on p. 59).

[75] RIOT. *RIOT Documentation*. URL: https://doc.riot-os.org/ (visited on 02/13/2024) (cit. on pp. 61, 63, 67, 85, 87).

[76] *TensorFlow Lite for Microcontrollers*. In collab. with Google LLC, Yuan Tang, and Arm Ltd. Version 2024.05.21. original-date: 2021-04-08T21:40:50Z. Nov. 6, 2024. URL: https://github.com/tensorflow/tflite-micro (visited on 11/06/2024) (cit. on p. 62).

[77] Martine Lenders et al. *Connecting the World of Embedded Mobiles: The RIOT Approach to Ubiquitous Networking for the Internet of Things*. Jan. 9, 2018. arXiv: 1801.02833[cs]. URL: http://arxiv.org/abs/1801.02833 (visited on 11/01/2024) (cit. on pp. 63, 69).

[78] LoRa Alliance. *LoRaWAN® specification v1.1*. manual. 2017. URL: https://resources.lora-alliance.org/document/lorawan-specification-v1-1 (cit. on p. 64).

[79] Eleni Tzirita Zacharatou, Volker Markl, and Elena Paz. *Towards Resilient Data Management for the Internet of Moving Things*. Jan. 1, 2021 (cit. on p. 69).

[80] *TTN mapper*. URL: https://ttnmapper.org (cit. on pp. 80, 113).

[81] LoRa Alliance. *LoRa alliance member NetID policy and terms*. manual. July 2024. URL: https://lora-alliance.org/wp-content/uploads/2024/08/LoRa-Alliance-Member-NetID-Policy-and-Terms-July-2024.pdf (cit. on p. 81).

[82] The Things Industries. *Major Changes In The Things Stack*. URL: https://www.thethingsindustries.com/docs/the-things-stack/migrating/migrating-from-v2/major-changes/ (visited on 10/01/2024) (cit. on p. 82).

[83] Andrew Rice and Simon Hay. "Decomposing power measurements for mobile devices". In: *2010 IEEE International Conference on Pervasive Computing and Communications (PerCom)*. 2010 IEEE International Conference on Pervasive Computing and Communications (PerCom). Mannheim, Germany: IEEE, Mar. 2010, pp. 70–78. ISBN: 978-1-4244-5329-0. DOI: 10.1109/PERCOM.2010.5466991. URL: http://ieeexplore.ieee.org/document/5466991/ (visited on 08/01/2023) (cit. on p. 88).

[84] Richard D Hipp. *SQLite*. tex.version: 3.31.1. 2024. URL: https://www.sqlite.org/index.html (cit. on p. 96).

[85] The pandas development team. *pandas-dev/pandas: Pandas*. Sept. 20, 2024. DOI: 10.5281/zenodo.13819579. URL: https://zenodo.org/records/13819579 (visited on 09/30/2024) (cit. on p. 97).

[86] Mark Raasveldt and Hannes Muehleisen. *DuckDB*. URL: https://github.com/duckdb/duckdb (cit. on p. 97).

[87] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007). Publisher: IEEE COMPUTER SOC, pp. 90–95. DOI: 10.1109/MCSE.2007.55 (cit. on p. 97).

[88] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020). Publisher: Springer Science and Business Media LLC, pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2 (cit. on p. 97).