UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN
COMPUTER ENGINEERING

"Communication solution for IoT devices using the Toit programming language"

Relatore: Prof. Andrea Galtarossa

Laureando: Alessandro Fano

ANNO ACCADEMICO 2021 – 2022

Data di laurea 5 Settembre 2022

DTU

DTU Compute

Department of Applied Mathematics and Computer Science

Communication solution for IoT devices using the Toit programming language

Master Thesis

Approval

This thesis has been prepared over six months at the Section for Embedded Systems Engineering, Department of Applied Mathematics and Computer Science, at the Technical University of Denmark, DTU, in partial fulfilment for the degree Master of Science in Engineering, MSc Eng.

Alessandro Fano - s217110

Alessandro Fano

Signature

July the 1st, 2022

Date

Abstract

Toit is a new object-oriented programming language for microcontrollers. The Toit virtual machine enables multiple independent apps to run side-by-side through software-based fault isolation. Toit is being developed as open source by the Danish company Toitware ApS, which collaborates with DTU Compute in the EU project TRANSACT. Although there are a plethora of programming solutions for IoT devices, they typically either involve low-level programming or their high-level programming requires too many resources. The objective of the thesis is to develop a communication solution for IoT devices using the Toit language. The solution proposed in this thesis is a tree-based network that allows devices to exchange data over Bluetooth Low Energy data channels without involving cloud connectivity.

Acknowledgements

Special thanks go to:

Paul Pop, Professor, DTU

for supervising the development of my master thesis;

Gaurav Choudhary, Postdoctoral Researcher, DTU

for helping me with the structuring and review of my thesis;

Kasper Lund, Co-founder and CEO, Toitware ApS

and the whole team behind Toit, for promptly providing me with assistance and software updates.

Contents

List of Figures

List of Tables

1 Introduction

The concept of the *Internet of Things* (IoT) was first predicted by Mark Weiser in 1991 [1] where he describes what he calls *Ubiquitous Computing*: an elevated number of heterogeneous devices, interconnected with each other in a wireless fashion, deeply integrated in the fabric of everyday life and so small to be mostly invisible.
In 2007 [2] it was predicted that by 2024 everything will be connected to the web to the extent that the environment in which we live will be fundamentally indistinguishable from the web itself, and that "Every item, every artefact [...] will have some sliver of connectivity that will be part of the web".
Today the aforementioned concept is a consolidated reality and although there is no universally accepted definition [3] , the IoT paradigm can be described as multitude of heterogeneous *smart objects* (*Things*) capable of exchanging data with each other over a network. Moreover, a *Thing* can be any physical object - or their virtual representation - that is assigned an unique identifier and is able to collect, exchange and process data over a Internet-like structure: in the broad spectrum of IoT application a Thing can be a home surveillance camera, a smart LED lightbulb, smart speakers, smartphones and smart wearable devices, but also RFID tags, a person wearing a heart monitor implant and a farm animal with a biochip transponder [4, 5].
The interest in Internet of Things grew over the year due to its huge potential in industrial, agricultural, healthcare and military application, and the numbers speak for themselves: in 2021 there were more than 10 billions active IoT devices and the number is expected to grow to more than 25 billions by 2030 [6]. If we also consider passive IoT connections - such as RFID tags - then by the end of 2030 there will be over 50 billions IoT devices installed around the world [7]. Although there has been a slow down in the production of smart devices, caused by the COVID pandemic and the war in Ukraine - earlier forecast from 2016 [8] were expecting 46 billions devices by 2021 - it is clear that the Internet of Things is here to stay.

IoT is already a consolidated paradigm in industrial applications - the so called Industry 4.0 - where a huge number of smart connected devices are used to manage fleet of autonomous vehicle and to monitor the production line [9].

In agriculture, a network of wireless sensors and satellites allows a farmer to monitor the crops soil moisture and temperature and to make decision based on rainfall and wind direction reports from a network of weather station [10, 11].
A person can use their smartphone to start the washing machine, modify the temperature of the air conditioning and warm the oven, on his way home from work [12].
Every time we wear a smartwatch, every time we control our thermostat through our smartphone and every time we ask Alexa to play a song, we are taking part in the Internet of Things: it is already part of our everyday life. IoT will be deeply integrated in the cities of the future - smart cities - which will employ a huge network of sensors to thoroughly monitor and manage every aspect of the city life, such as smart transportation, home and office automation, traffic, security and smart energy and water management[13].

The wide variety of applications and scenarios makes it impossible to find a one-fit-all solution when it comes to communication technologies, but rather multiple possible choices depending on the requirements of the system. Other than that, the biggest challenge that the academia and the industry are facing when it comes to IoT development is the lack of standardization: the multitude of protocols employed by these devices, the platform they run on and the lack of a universally accepted architecture is seriously slowing down the technological advancement of IoT [14]. This greatly affect interoperability [15] - *i.e.*, the capacity for multiple components within an IoT deployment to effectively communicate, share data and perform together to achieve a shared outcome - and poses a security threat as well [16]. The absence of a standard approach to IoT often leads IoT developers to employ their own proprietary solutions, further deepening the issue. In some cases, major participants in the IoT scene might want to defer the standardization debate for as long as possible in order to preserve some proprietary technology that controls the access to the market.
On the other hand, there are some serious concerns whether the imposition of a universal standard might be used by powerful actors in an attempt to seize a decisive technological advantage at the expenses of their competitors [17].

The other great issue that IoT is facing is the incapability of traditional cloud models to keep up with the evergrowing amount of data produced on site. The huge volume of data produced tirelessly by the billions smart objects that make up the Internet of Things needs

to be transferred to the cloud to be processed, however, the bandwidth availability and latency are restrictive bottlenecks when it comes to time-critical scenarios or application that crank out large quantities of data.
To address this issue, there is a tendency in providing the smart devices with more computing power, so that part of the data processing can be performed on the edge of the network. The importance of this new model of computation, called *Edge* or *Fog* computing, in the take over of the Internet of Things is discussed in section 2.1.

In most of the mentioned cases the smart objects - which are the building blocks of the Internet of Things - are small microcontrollers programmed to implement some communication protocol in order to connect with other devices. Developing and deploying the code for these microcontroller is a crucial part in the building of an IoT systems.
Toitware ApS, a Danish company founded in 2018, developed the Toit platform, along with a high level language with a syntax similar to Python's, to ease the task of programming microcontrollers in IoT contexts. The code runs on a virtual machine which allows for over-the-air updates and multiple programs running at the same time, independently [18].
The scope of this thesis is to explore the capabilities of a Toit-powered edge device in the Internet of Things.
The thesis is structured as follows:

- In chapter chapter 2 will be given an overview of IoT systems and a survey of the most popular communications protocol employed in IoT applications, their advantages, their practical issues and their typical use cases;
- In chapter chapter 3 will be given a detailed description of the Bluetooth Low Energy (BLE) protocol;
- In chapter chapter 4 will be given a description of the Toit platform;
- In chapter chapter 5 will be described the implementation of a communication solution using the BLE protocol in the Toit programming language;
- In chapter chapter 6 the proposed implementation will be evaluated.

1 Introduzione

L'idea di *Internet of Things* (IoT) o *Internet delle Cose* è stata immaginata inizialmente da Mark Weiser nel 1991[1] con il nome di *Ubiquitous Computing* (Computazione Onnipresente): un elevata quantità di dispositivi eterogenei, interconnessi tra di loro in modalità wireless, profondamente integrati nel tessuto della vita di tutti i giorni e di dimensioni così ridotte da essere praticamente invisibili.
Nel 2007 [2] è stato previsto che entro il 2024 l'ambiente che ci circonda sarà sostanzialmente indistinguibile dal web stesso, e che "ogni oggetto, ogni manufatto [...] sarà almeno in parte connesso al web".
Oggi i concetti appena riportati sono una realtà affermata e nonostante non ne esista una definizione universalmente accettata [3], il paradigma IoT può essere descritto come una moltitudine di *smart object* (*Things, Cose*) in grado di scambiare dati tra di loro tramite una rete. In particolare, una *Cosa* può essere ogni oggetto reale - o la sua rappresentazione virtuale - a cui viene assegnato un identificatore univoco, in grado di raccogliere, trasmettere ed elaborare dati tramite una rete come l'internet: nell'ampio spettro delle applicazioni IoT, una Cosa può essere una videocamera di sorveglianza domestica, una lampadina LED intelligente, altoparlanti smart, smartphone e dispositivi smart indossabili, ma anche etichette RFID, una persona che indossa un impianto per il monitoraggio cardiaco o un animale da fattoria equipaggiato con un biochip transponder [4, 5].
L'interesse per l'Internet of Things è cresciuto negli anni grazie al suo enorme potenziale in applicazioni industriali, agricole, sanitarie e militari, e i numeri parlano da sé: nel 2021 erano attivi più di 10 miliardi di dispositivi IoT, e si aspetta che entro il 2030 saranno più di 25 miliardi [6]. Se si considerano anche le connessioni IoT passive - come le tag RFID - allora entro la fine del 2030 ci saranno più di 50 miliardi di dispositivi IoT operativi nel mondo [7]. Nonostante ci sia stato un rallentamento della produzione di dispositivi smart causata dalla pandemia COVID e dalla guerra in Ucraina - nel 2016 le previsioni si aspettavano 46 miliardi di dispositivi entro il 2021 [8] - è chiaro che l'Internet of Things è qui per rimanere.

L'IoT è un paradigma già affermato nelle applicazioni industriali - la cosiddetta Industria 4.0 - dove un grande numero di dispositivi smart connessi vengono usati per gestire flotte

di veicoli autonomi e per monitorare la linea di produzione[9].
Nel settore agricolo, una rete di sensori wireless e satelliti permette all'agricoltore di monitorare l'umidità e la temperatura del terreno, e di prendere decisioni basate sui report atmosferici provenienti da un network di stanzoni meteo [10, 11].
Un utente può usare il proprio smartphone per avviare la lavatrice, modificare la temperatura del condizionatore e preriscaldare il forno, durante il tragitto di ritorno dal lavoro [12].
Ogni volta che indossiamo uno smartwatch, ogni volta che controlliamo il termostato della nostra casa con il cellulare e ogni volta che chiediamo ad Alexa di riprodurre un brano, stiamo prendendo parte all'Internet delle Cose: è già parte della nostra vita quotidiana.
L'IoT sarà profondamente integrato nelle città del futuro - smart cities - che impiegheranno un ampio network di sensori per monitorare e gestire scrupolosamente ogni aspetto della vita cittadina, come i trasporti pubblici smart, automazione domotica, traffico, sicurezza e gestione smart dell'acqua e dell'energia[13].

La grande varietà di applicazioni e scenari rende impossibile una soluzione universale, parlando di tecnologie di comunicazione: esistono invece diverse alternative possibili a seconda dei requisiti di sistema. Oltre a questo, la grossa sfida che l'industria e la ricerca stanno affrontando nel campo sello sviluppo IoT è la mancanza di standardizzazione: la moltitudine di protocolli impiegati da questi dispositivi e la mancanza di un'architettura standard universalmente riconosciuta costituiscono un serio rallentamento all'avanzamento tecnologico dell'IoT [14]. Questo ha un impatto negativo sull'interoperabilità [15] - *i.e.*, la capacità di diversi componenti all'interno di un sistema IoT di comunicare, scambiare dati e collaborare efficacemente al fine di perseguire un traguardo comune - e costituisce inoltre un rischio per la sicurezza [16]. L'assenza di un approccio standard spinge molti sviluppatori IoT a impiegare le loro soluzioni proprietarie, aggravando di conseguenza il problema. In alcuni casi, grandi nomi del settore IoT hanno tutto l'interesse nel rallentare e posticipare il dibattito sulla standardizzazione, con lo scopo di preservare alcune tecnologie proprietarie che controllano l'accesso al mercato.
D'altro canto, ci sono serie preoccupazioni riguardo al fatto che l'imposizione di standard universali possa essere sfruttata da attori potenti per impadronirsi di vantaggi tecnologici decisivi alle spalle dei loro concorrenti [17].

L'altro grande problema che l'IoT si ritrova ad affrontare è l'incapacità dei modelli cloud

tradizionali di stare al passo con la enorme quantità di dati, in costante crescita, che viene prodotta sul posto. L'enorme volume di dati prodotto incessantemente dai miliardi di dispositivi che formano l'Internet of Things necessita di essere trasferito sul cloud per venire elaborato, tuttavia, la disponibilità di banda e la latenza sono stretti colli di bottiglia, specialmente in applicazioni time-sensitive o che sfornano grosse quantità di dati.
Per affrontare questo problema, c'è la tendenza a fornire sempre più potenza di calcolo ai dispositivi smart, così che parte dell'elaborazione dati possa svolgersi in loco. L'importanza di questo nuovo modello di elaborazione dati, chiamato *Edge* o *Fog* computing, nell'avvento dell'Internet delle Cose, è discussa insection 2.1.

Nella maggior parte dei casi menzionati, gli smart objects - che sono i mattoni costituenti dell'Internet of Things - sono piccoli microcontrollori programmati per implementare qualche protocollo di comunicazione, con lo scopo di connettersi ad altri dispositivi. Sviluppare il codice per questi microcontrollori è una parte cruciale nella costruzione di un sistema IoT. L'azienda Danese Toitware ApS, fondata nel 2018, ha sviluppato la piattaforma Toit, assieme a un omonimo linguaggio di alto livello simile a Python, per facilitare il compito di programmare microcontrollori in un contesto IoT. Il codice sviluppato viene eseguito su una macchina virtuale che consente aggiornamenti over-the-air e esecuzione simultanea ed indipendente di diversi programmi [18].
Lo scopo di questa tesi è di sondare le capacità di dispositivi Toit come edge device nel contesto dell'Internet of Things.
La tesi è strutturata in questo modo:

- Nel capitolo chapter 2 verrà data una panoramica dei sistemi IoT e un elenco dei protocolli di comunicazione maggiormente impiegati in applicazioni IoT;
- Nel capitolo chapter 3 verrà data una descrizione dettagliata del protocollo Bluetooth Low Energy (BLE);
- Nel capitolo chapter 4 verrà data una descrizione della piattaforma Toit;
- Nel capitolo chapter 5 verrà descritta l'implementazione di una soluzione di comunicazione basata sul protocollo BLE usando il linguaggio di programmazione Toit;
- Nel capitolo chapter 6 l'implementazione proposta verrà valutata.

2 Literature Review

2.1 Cloud Computing and Edge Computing

Cloud computing is a paradigm characterized by the transfer of data to and from a client over the internet. Cloud services such as Dropbox, Google Drive and iCloud are extremely popular and most of us uses them on a daily basis, but it's important to remark that cloud computation is much more than just online data storage. Cloud computation involves data synchronization between multiple distributed devices, data processing on server side and data transfer from the server back to the client over the internet.

The focus of cloud computing, also called on-demand computing, is to exploit the maximum potential of computing resources, distributed all over the globe, shared between multiple clients and dynamically reallocated [19, Chapter 2].

Given the premises, cloud computing has worked - and works - well when it comes to non-time-driven data processing that require huge amount of computing power. However, the enormous abundance of data produced tirelessly by the billions of smart things connected to the network is incompatible with traditional cloud models. Not only the amount of bandwidth required to transfer all the data to the cloud is out of reach, but the latency - *i.e.*, the time between the data being produced by the device and its processing - makes cloud computing unsuitable for time-sensitive applications.

The unprecedented volume of data generated by the Internet of Things that needs to be processed and analysed on the spot require for a new model of computation, where the processing power is physically located in proximity of the data producers, that is, on the *Edge* of the network [19, Chapter 2].

Edge computing, also called *fog computing*, is a new paradigm that involves the placement of *fog nodes* - any device with processing, storage and network connectivity capabilities - to extend the capabilities of the cloud allowing for the processing of time-sensitive data shortly after they are produced. The potential of edge computing in the contest of IoT is well understood and estimates shows that in 2015 the amount of data analysed on the edge is 40% [20]. This paradigm shift is inevitable and will likely become a standard in IoT applications. The cloud alone cannot fulfill the necessities of the Internet of Things, and the relevance of edge computing will grow to the point of surpass the cloud's [21].

2.2 IoT Architecture

Due to the lack of standardization mentioned in the Introduction, there is no universally accepted architecture for IoT systems, but rather multiple possible layered representations. One widely used when designing an IoT infrastructure is the four-stage architecture (fig. 2.1) [22][23][24]:

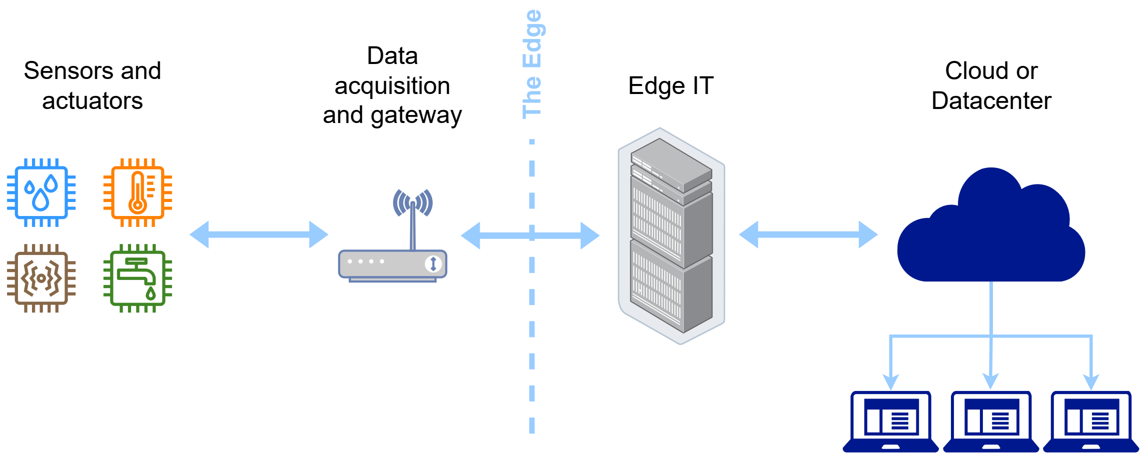

Figure 2.1: Layout of the four-stage architecture.

(1) *Sensors and Actuators*: this is the stage of the *Things*. In this stage are present devices capable of converting the information from the environment into data. Such devices are, for example, temperature sensors, water-level detectors, pressure sensors, accelerometers and so on. We use the term *sensor* in a broad sense: everything is counted as a sensor as long as it provides data about its current state.
Actuators are devices capable of intervene on the environment in order to change the physical conditions that generate the data. An actuator can regulate a water valve, shut off the power supply or adjust the speed of a cooling fan.
In this stage there is usually no processing - although in a IoT architecture some data processing may occur in each of the four stages - and the data is forwarded to the next stage;

(2) *Data Acquisition and Gateway*: in this stage the raw information from the sensors is collected, converted into digital data and pre-processed by the Data Acquisition System (DAS). The Internet Gateway takes the digital data and forwards it to Stage 3 of Stage 4 systems via Wi-Fi or wired LAN.
The Stage 2 systems often are physically located in proximity of the Stage 1 systems: imagine an industrial machinery in a production line that mounts several sen-

sors and actuators (Stage 1) connected - for instance via Bluetooth - to the Electronic Control Unit (Stage 2) of the machinery.

This stage is of vital importance because the several sensors in the previous stage will produce a large volume of data in short times which exceed the available bandwidth if directly forwarded to the next stages. To carry on the previous example, imagine a facility employing hundreds of industrial machinery, each of them with dozens of sensors constantly producing data. This data will quickly overcome the capacity of the infrastructure if directly forwarded to the next stage, therefore every machine is equipped with a Stage 2 system that takes care of collecting, converting and pre-processing the sensor's data, significantly reducing its volume, before submitting it to the next stage.

(3) *Edge IT*: this stage refers to processing systems that offer enhanced analytics, machine learning and visualization technologies. This stage of data processing allows to have quick access to meaningful information and scan for anomalies in the data. In time critical scenarios this is particularly useful rather than just send the data to the next Stage. Edge systems are often physically close to the previous stage: in our example, the facility employs a highly integrated compute system which process the data from all the machines in the production line and provides an easy to read overview of the status of each machine to the human operators, while also displaying production statistics and scanning for anomalies.

    In some application, where the amount of data produced in the previous stages is contained and there aren't time critical requirements, there is no need for a Stage 3 system, and the data is directly forwarded to the Cloud - *e.g.*, in a smart home context.

    In other cases it may be more convenient to have powerful smart devices, capable of processing the data on their own before sending it to the cloud, therefore incorporating Stage 2 and 3 in the same device;

(4) *Cloud or Data Center*: this is the last stage, where the data is stored for in-depth processing and analysis. This stage allows to extract insights, trends and patterns from the gathered information, in order to make crucial business decisions. Cloud-based systems provide the processing power to perform thorough examination of the data sent by several facilities. At this stage, the data is accessible for every device with an internet connection.

The architecture just described is by any means no standard in IoT, but rather an attempt to categorize all the parts in an IoT system; in some scenarios there might not be a clear distinction between the stages.
Other architectures proposed in the literature are the three- and five-layer architecture (fig. 2.2). The three-layer architecture is the most basic layered representation that defines

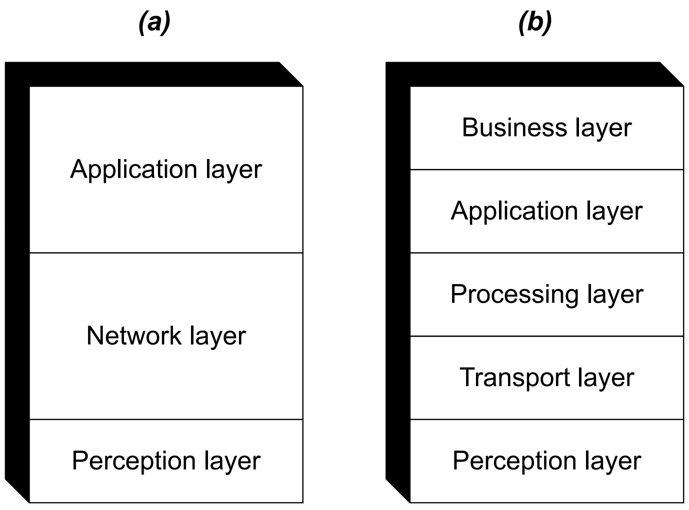

Figure 2.2: ***(a)*** three-layer and ***(b)*** five-layer architecture.

the main idea of IoT [25][26]. The three layers are:

(1) The *Perception layer* is a physical layer composed of sensors and smart objects. It is responsible for sensing and gathering information about the environment and identifying other smart devices;

(2) The *Network layer* is responsible for transmitting and processing information obtained from the perception layer, and connecting to other smart objects;

(3) The *Application layer* is responsible for delivering application specific services to the end user such as data visualization and advanced analytics. The function of this layer is providing all kinds of applications for each industry.

The five-layer architecture proposed by the literature allows for a finer distinction between the functionalities of IoT systems [25][26]. The layers are:

(1) The *Perception layer*, as before;

(2) The *Transport layer* is responsible for transmitting the data received from the Perception Layer to the processing center through various network;

(3) The *Processing layer* store, analyse and process the large volume of information

received from the transport layer. The reason behind this layer being severed from the 3-layer architecture's network layer is that the huge amount of information to be processed represents one of the main challenges of IoT systems. The main technologies employed are databases, cloud computing and big data processing

(4) The *Application layer*, as before;

(5) The *Business layer* manages the whole IoT system, including applications, business and profit models, and users' privacy.

2.3 Communication protocols

As explained in the introduction, the variety of IoT applications goes hand in hand with the multitude of the protocols employed. They differ for range, data rate, power consumption, network topology, security and frequency, making each protocol valid for some specific scenario. The protocols can be categorized in several way, depending on weather they are for short or long range data transfer; which layer of the OSI model they work on and so on.

Bluetooth

Bluetooth is a low power radio that streams data over 79 channels in the 2.4GHz unlicensed industrial, scientific, and medical (ISM) frequency band [27][19, Chapter 3]. It was introduced in 1994 as a wireless communication standard for data exchange between computer and mobile phones, but gained its initial popularity thanks to wireless headsets that allowed to make phone calls without holding the phone. Today it's one of the most used IoT protocols in domestic application and in handheld devices - every smartphone has Bluetooth capabilities - and estimates forecast 7 billions of Bluetooth enabled devices shipped annually by 2026 [27].
The Bluetooth standard is completely controlled by the Bluetooth Special Interest Group (SIG) and it includes application profile to describe the data exchange for a particular task, like audio streaming or remote control of a television. The Bluetooth SIG also offers qualification process to ensure that every product that utilize the technology comply with the standard specification. This allows for an incredible flexibility and interoperability

between certified devices [28]. Bluetooth is designed for medium-short range - up to 100 meters for industrial grade radios - low power data transmission at up to 3 Mbit/s. This, along with the cost effectiveness of the implementation, makes it first choice technology for wearable smart devices and domestic application in small battery powered devices. Bluetooth offers point-to-point and star type (piconet) network topology.
Bluetooth is typically employed in smart watches, smart home appliances and home automation, medical devices, car and home entertainment systems and all those application where a device is connected to a smartphone or a tablet.
The main disadvantage of classic Bluetooth technology is it's reach, especially the fact that the strength of the connection drops quickly when obstacles such as walls are between its path, even within the 10 meter range. The topology offered by this technology does not allows to work around this problem. Another drawback is the limit of active connections - one master can have only seven active connection to other slave devices - and the security issues found is some popular chipsets that could lead to denial of service and arbitrary code execution [29].

MQTT

Message Queuing Telemetry Transport is a machine-to-machine, open source network protocol. It was introduced in 1999 by IBM as a way to monitor oil pipelines and since then has been widely employed in industrial scenarios. In 2013, the OASIS MQTT technical committee was founded in an effort to standardized the protocol [30], in order to make it a more viable option in the IoT scene. The light weight of the protocol and it's ability to deliver data messages over unreliable network make MQTT the ideal choice when it comes to condition monitoring in logistic and industrial application, such as transportation status monitoring [31] and petrolchemical plant and powerplant monitoring [32][33], but it is also used in smart home scenarios for energy and water consumption monitoring and smart home appliances [34].
MQTT is designed to work on low resources and optimize the network bandwidth and allows for bidirectional communication device-to-cloud and cloud-to-device. It is built on top of TCP/IP and is suitable to 2G, 3G and 4G networks [35]. MQTT employs a publish/subscribe architecture (fig. 2.3) by defining two entities: clients - publishers and/or subscribers - and brokers - or servers. Clients can connect to a broker and publish messages under

a certain topic. Whenever a message is received by the broker, it is forwarded to all the clients subscribed to the message topic [36]. This architecture allows for millions of devices to exchange data in an efficient way, ensuring great scalability.
Another great feature is the reliability of the message delivery, with three defined level of

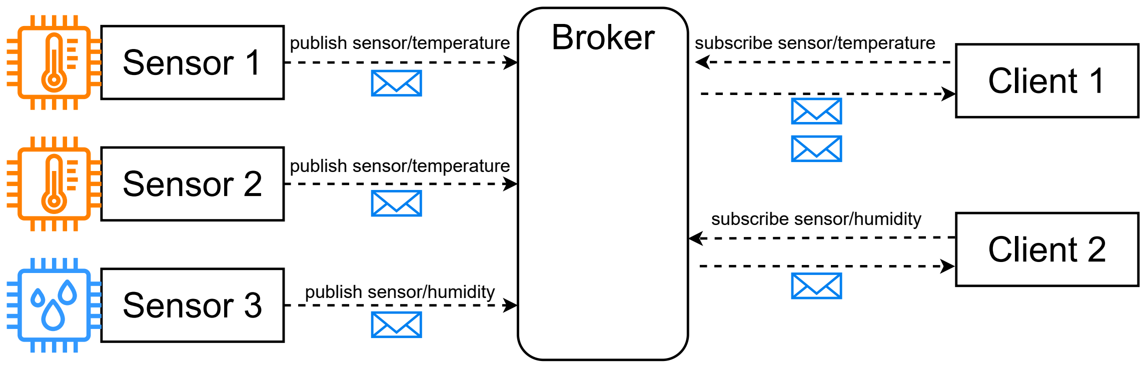
Sensor 1
publish sensor/temperature
Sensor 2
publish sensor/temperature
Sensor 3
publish sensor/humidity
Broker
subscribe sensor/temperature
Client 1
subscribe sensor/humidity
Client 2

Figure 2.3: MQTT client-broker architecture.

quality of service (QoS), in increasing order of overhead [37]:

(1) *At most once* (QoS 0): there is no guarantee of delivery. The recipient does not acknowledge receipt of the message and the message is not stored and re-transmitted by the sender;

(2) *At least once* (QoS 1): guarantees that a message is delivered at least one time to the receiver. The message is stored by the sender and retransmitted periodically until the receiver responds with an acknowledgement;

(3) *Exactly once* (QoS 2): a four-part handshake between the sender and the receiver guarantees that the message is received exactly one time. This QoS has greater latency and overhead.

The messages are composed of a 1-byte control header, that defines the message type and the flags; a variable header of length 1 to 4 bytes, to carry additional control information; and the payload, for a maximum packet size of 256 MB [37].
The main drawback of this protocol is that, due to the asynchronous nature of the communication, one publisher have no way to know if the message has reached the desired client; there are also some open discussions about the feasibility of the quality of service of level 3 in practice [38]. Another critical point, inherent to the architecture, is the centrality of the broker, which, in case of failure, will interrupt the connection between all the devices involved.

ZigBee

ZigBee is wireless networking protocol defined in the IEEE 802.15.4 standard as a Low Rate Wireless Personal Area Network (LR-WPAN) [39]. It is designed as an open global standard to address the needs of low-cost, low-power wireless IoT networks.
It offers support for multiple network topologies such as point-to-point, star and mesh networking (fig. 2.4), the latter being a key point in it's usefulness in IoT scenarios. The ZigBee architecture consist of three types of nodes:

(1) *The coordinator*: there is one in every network and is responsible for handling and storing the information while receiving and transmitting data operations;

(2) *The routers*: allow the data to hop through them;

(3) *The end devices*: produce and consume the data.

The small dimensions of ZigBee chips - 5x5 mm - makes it an ideal technology to adopt in small battery-powered device.
ZigBee operates in the 2.4 GHz (global), 915 MHz (America) and 868 MHz (Europe) frequencies, and has a coverage from 10 to 100 meters in line-of-sight, depending on the power output. The ZigBee Alliance takes care of maintaining and updating the specifications, ensuring a good level of interoperability between products from different vendors [40] but still providing the possibility of creating specific variation to manufacturers.
Due to its characteristic, ZigBee is placed in direct competition with Bluetooth as a low power, high interoperability, low range technology for small IoT contest such as home automation and healthcare applications like patient condition monitoring [41].
With respect to Bluetooth, ZigBee offer more possibilities in terms of network topologies, which in turns means a greater reach of the network. ZigBee mesh network is self-forming and self-healing, meaning that it configures itself automatically and dynamically to repair itself if some nodes are removed; they also allows for a much greater number of connected devices to the network - 65536 devices in a ZigBee network against 8 devices in a classic Bluetooth piconet.
On the downside, the larger the network, the greater the latency between two distant devices, as the message need to hop through the routers to reach its target. A great number of devices also means that each one of these devices must be powered, although ZigBee offers two operating modes, beacon and non-beacon. In beacon mode, the coordinator

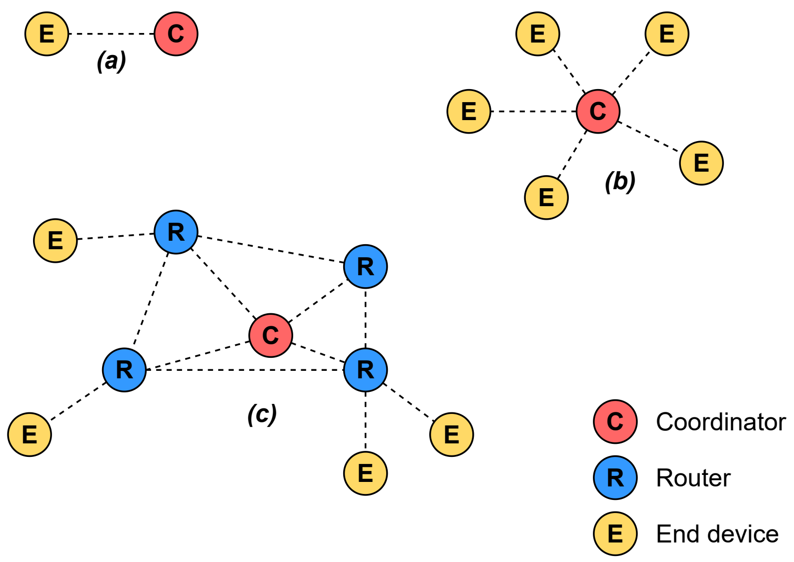

Figure 2.4: ZigBee network topologies: ***(a)*** point-to-point, ***(b)*** star and ***(c)*** mesh

.

periodically transmit the beacon, which is used to dictate a schedule and synchronize the communication in the network: this way, all the devices know when to communicate to each other, allowing them to sleep in between beacons. This mode works best when the coordinator and the routers are battery powered, as long as the timing circuits in the devices are accurate. In non-beacon mode instead, the end devices are sleeping almost all the time and periodically wake up to confirm their presence to the coordinator, which is always awake. The end devices only start communicating on detection of activity.
The great advantages of ZigBee are consrained by the low data rate of 250 kbit/s on 2.4 GHz band. This is because ZigBee is designed for Wireless Sensonr Network (WSN), expecially for condition monitoring context, where the volume of data is little [39].

Z-Wave

Z-Wave is proprietary wireless communication protocols developed the Danish company Zensys in 1999 for residential and lightweight commercial environments, and is now regulated by the Z-Wave Alliance.
Z-Wave operates in the sub-1 GHz band - from 865 Mhz to 926 MHz depending on the country - which ensure low power transmission at 100 kbit/s up to 30 meters [42]. By operating in the sub-1 GHz band, Z-Wave signal is able to penetrate obstacles such as walls while avoiding possible collisions with Wi-Fi and Bluetooth signals - but may interfere with cordless phones and other wireless devices.

Z-Wave network operates using controllers and slaves. A controller may send a message to a slave which act as a monitoring device or as an actuator; thus, responds and executes the controller's instructions. Slave nodes are usually small low cost battery-powered devices, while masters are usually smartphones or wireless remotes. Z-Wave supports mesh networking allowing message to hop from one device to another, extending the reach of the network [43].
Z-Wave technology shares a lot of applications with ZigBee and Bluetooth, its main competitors, however, Z-Wave was specifically design to transmit small messages from one control unit to slave devices; instead of utilising a large bandwidth, it supports only short burst commands such as toggling the lights, set the alarm, locking the doors, turning on the sprinklers etc. The extremely low power consumption of the slave devices makes it an excellent choice in home automation - smart lightning, smart locks, smart security and alarms [44] - by ensuring long battery life and low latency transmission.
There are currently more than 700 members of the Z-Wave Alliance that manufacture Z-Wave devices, the most famous being Samsung SmartThings and General Electric, and in 2005 there were over 50 millions Z-Wave devices for a total of 70% of the home automation market share [45]. Z-Wave devices are easy to setup and is guaranteed vendor-agnostic levels of interoperability and backward compatibility [43], although the latter has raised some concern about security [46].
Z-Wave mesh networks can theoretically accommodate 232 nodes, although most vendors recommend to deploy no more than 40-50 devices. Another downside is that the maintenance of the network is cumbersome: moving a slave device once it was added to the network may prevents other devices from receiving messages, and there are cases of unusual behaviour reported by customers. The solution in this case is to factory reset the master unit and relearn the network topology [42].
The region-specific operating frequency is also a limiting factor, devices working in one country may not function in another one.

LoRaWAN

LoRaWAN - which stands for Long Range Wide Area Network - is a MAC layer, open protocol developed and maintained by the LoRa Alliance. The first specification was released in 2015, making it one of the youngest protocol for IoT. It is built on top of LoRa, a physical proprietary radio modulation technique derived from the Chirp Spread Spectrum

(CSS) technology.

The technology employs a spreading technique, according to which a symbol is encoded in a longer sequence of bits, thus reducing the signal to noise and interference ratio required at the receiver for correct reception, without changing the frequency bandwidth of the wireless signal. This make it possible to select the data rate in the range of 300 bit/s to 37.5 kbit/s, which offers a trade off between throughput and coverage range or power consumption [47]. This characteristics allows LoRa to cover a greater area with less gateways, compared to cellular network. LoRa and LoRaWAN together defines a Low Power Wide Area (LPWA) networking protocol designed to connect multiple battery-powered devices over long distances. The technology operates in the 169 MHz, 433 MHz and 915 MHz bands in the USA, but in Europe it works in the 868 MHz band.

LoRa networks are typically deployed in a star-of-stars topology: multiple end devices - wireless sensors and actuators - are connected through a single hop to one or many gateways, that are in turn connected to a common Network Server via standard TCP/IP protocols (fig. 2.5). The end devices are not required to associate to a specific gateway to gain access to the network, but only to the Network Server; the gateways act as simple relays and forward the data to the network server, which is responsible for filtering the messages and replying to the end devices.

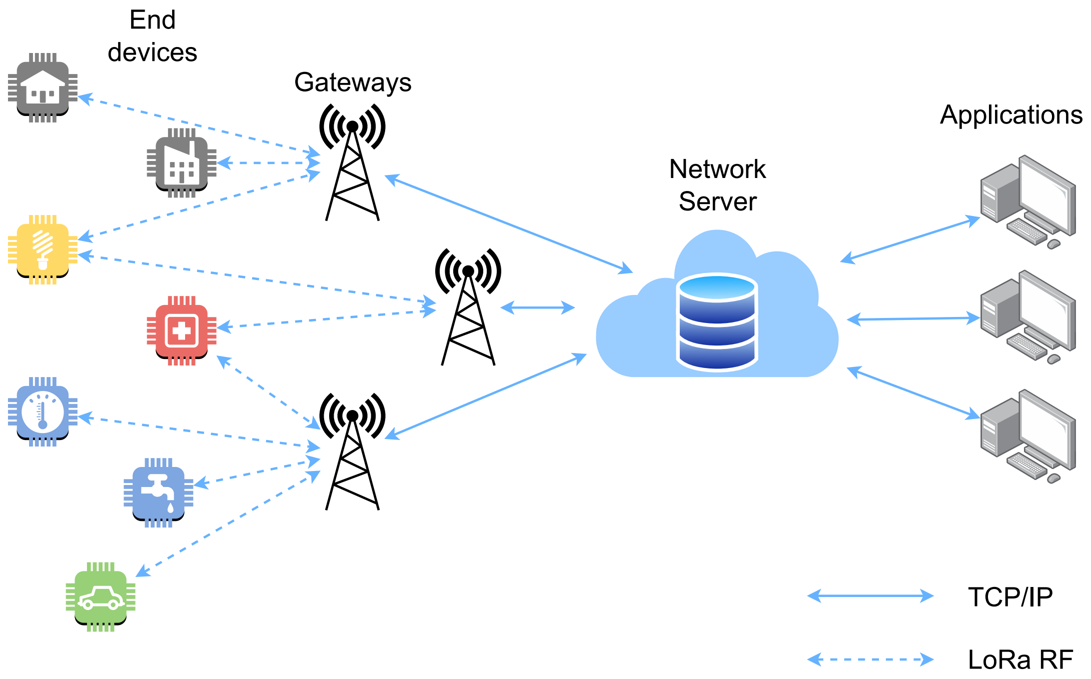

Figure 2.5: LoRaWAN network architecture overview.

Interoperability of end devices is guaranteed by the LoRa Alliance Certification Programs, which ensure end customers that their application-specific end devices will operate on any LoRaWAN network [48].

The long range coverage even in harsh environments and the indoor penetration capability, granted by operating in the sub-1 GHz spectrum, makes LoRa the technology of choice in large-scale residential, agricultural and smart city applications, such as room temperature monitoring of entire buildings [47], smart lightning and smart parking [49]. The coverage of a single gateway is between 2 and 6 km in urban scenarios, and up to 18 km in rural areas [50], making it possible to cover the extent of a large city with less then a hundred gateways, each of them capable of supporting 15000 nodes [47].

By operating in the unlicensed ISM band it's not required to buy expensive frequency spectrum license fees to deploy a LoRaWAN network, however this comes at the cost of adopting 1% (or 0.1%) duty cycled transmission depending on the transmission frequency, as regulated by regional and global entities, which is a limiting factor in terms of throughput and network size [51][52].

Another downside is that the star-of-stars network topology adopted by LoRaWAN, in particular the use of gateways to connect the end devices might be a bottleneck due to a single point of failure. LoRaWAN modules are also more expensive than other RF modules that employ GFSK and FSK, and the technology itslef is not suited for real time applications that require low latency.

Despite these flaws, LoRaWAN is a young and modern protocol - the most recent specification is from October 2020 [53] - that is bound to be a big player in smart cities IoT applications [47].

Sigfox

Sigfox is a French global network operator founded in 2010 with the goal of providing a controlled network for low-power IoT devices, similar to the cellular network. Sigfox proprietary technology employs ultra-narrow band (UNB) modulation, taking up only 192 KHz of the 868 MHz (Europe) and 91MHz (US) bands in the unlicensed ISM spectrum [54]. Sigfox offer ultra-long range bidirectional communication at extremely low power consumption. The UNB modulations allows for high resilience to interference, coverage from 3 km up to 10 km in urban area - and up to 30 km in rural area [55] - and good indoor penetration capabilities. This at the cost of an extremely low data rate of up to 100 bit/s and small size payload of 12 bytes.

Like other LPWA networking protocols, Sigfox aims at providing connectivity for low-end sensors distributed over a wide area in agricultural, industrial and smart city applications. For example, Sigfox has been employed in Marseille to monitor the water level of over 5,000 storm drains using self-powered sensors [56].

The SigFox network topology has been designed to provide a scalable, high-capacity network, with very low energy consumption, while maintaining a simple and easy to rollout one-hop star-based cell infrastructure, as show on fig. 2.6: the devices are connected to base stations, which act as gateways to forward the messages to Sigfox Support System over standard IP link.

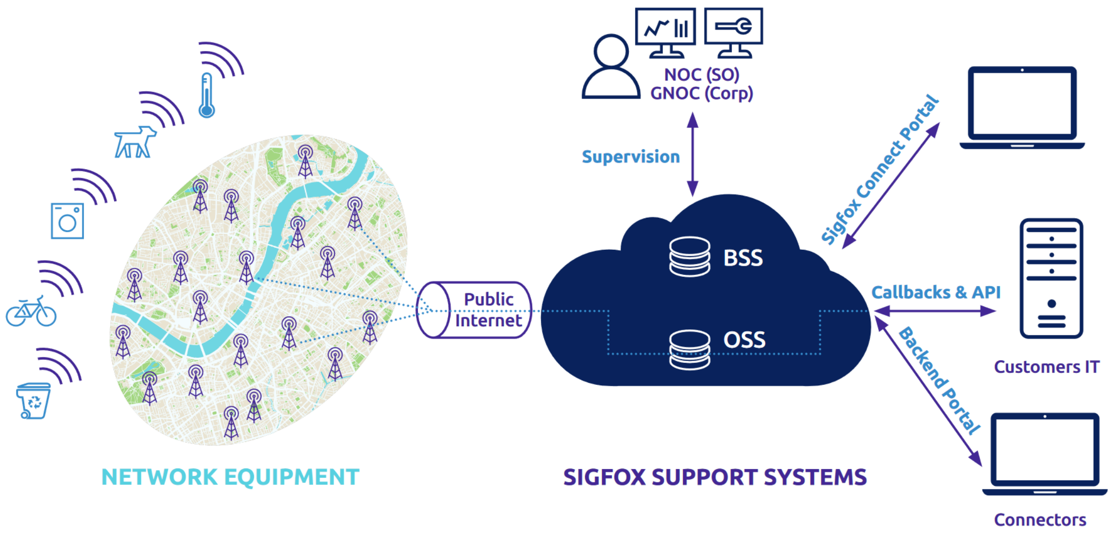

Figure 2.6: Sigfox network architecture overview [57].

The Sigfox Support System is in charge of processing the messages and send them through callbacks to the customer system. The data can be accessed and collected by the users from any device with an internet connection using web interfaces and API [57]. Unlike cellular connectivity protocols, Sigfox devices are not attached to a specific base station and the transmitted messages are received by any nearby stations (spatial diversity). The transmission is unsynchronized between the network and the device. The device emits a message on a random frequency and then sends 2 replicas on different frequencies and time (time and frequency diversity). Spatial diversity coupled with the time and frequency diversity of the repetitions are the main factors behind the high quality of service of the Sigfox network [57].

Sigfox is present in 75 countries and services 20 millions registered devices over 6 million square kilometers of covered area.
On the downsides, Sigfox suffer from the same ISM band usage restrictions of LoRa, constraining the devices to no more than 6 12-bytes per hour [57]. This, coupled with the 25 seconds of latency between reception and transmission make Sigfox unsuited for real time applications and scenarios where large volume of data is produced over short periods of time.
Other than that customers are required to purchase a subscription from Sigfox certified local cellular operator in order to deploy their smart embedded devices. Given the premises, Sigfox is a viable options for big business and public administrations that require infrequent monitoring of multiple sensors spread over a large area.

Radio Frequency Identification (RFID)

RFID is a technology based on the use of electromagnetic fields to transfer small quantity of data between two devices in close proximity. Predecessors of this technology have been around since the 1940s, but the first patented commercial RFID systems dates back to the 1980s [58]. Since then RFID systems have been widely deployed in logistic applications in industrial and commercial environments, such as items tracking, precision timing and telemetry, and toll collection.
A typical RFID system is composed of two main entities: a tag and an interrogator [59][60]. A micro chip and a small antenna constitute a tag. The tag information - from a few bits up to 8KB - is stored in a non-volatile memory that can be either read-only, read-write or write-once read-multiple. Tags have a unique ID and can be of three types:

(1) *Passive*: they cheaper and smaller because they have no battery, instead they are powered by the current induced by the signal sent from the interrogator to transmit the data;

(2) *Active*: they coupled with a small battery and periodically transmit their ID;

(3) *Battery-assisted passive*: they have batteries like active tags but are only activated when in presence of an interrogator.

RFID tags come in several form factors and are priced a few cents up to 100$, depending on frequency, memory size, battery life, encapsulation protections, etc [61]. High-end active tags can also mount sensors to capture environmental changes in temperature,

humidity, pressure and even GPS. Due to its simple nature, RFID is a technology prone to miniaturization and the smallest tag measure only 0.15x0.15 mm [62].
Interrogators are devices that transmit and receive radio waves and are responsible of the communication with RFID tags.
RFID operates in three frequency bands: low frequency (125–134.2 kHz and 140–148.5 kHz), high frequency (13.56 MHz) and ultra-high frequency (865–928 MHz). While low and high frequency RFID tags can be used globally, there is no global standard for ultra-high frequency tags, and the specific regulations differs from country to country. Depending on the type and the operational frequency, RFID tags can be read at less than 10 centimeters up to 100 meters regardless of obstacles and occlusion [63].
RFID tags are a relatively inexpensive solution that provide no-latency transmission of small quantities of data at short distance; they are widely employed for contactless payments, ID cards and machine readable documents, smart locks, tracking of shipping, luggage and livestock, anti-theft systems in retail and timing of sport events [64].
On the downside, the wide use of RFID tags raised several security concerns about the fact that the tags can be read by unauthorized users with malicious intents, and legitimate transactions can be eavesdropped from non-trivial distances. This allows the criminal to identify or track packages, persons, carriers, or the contents of a package [65] [66]. The term ”RFID skimming” is the practice of unlawfully obtain someone’s payment card information using a RFID reading device [67].
There are also privacy concerns since RFID tags used in retail remain functional after the customers have purchased and taken home the products, making it possible to be used for surveillance and other unlawful purposes unrelated to the original logistic functions of the tags.
Despite that, RFID tags are still widely employed and the total RFID market is expected to be worth over $12 billions in 2022 [68].

Near Field Communication (NFC)

NFC is a secure, short-range communication standard for interaction between two electronic devices wirelessly, without the need for any prior setup [69]; the first NFC specification was introduced in 2003 by Sony and Philips [70].
NFC technology is based on RFID and operates at 13.56 MHz in the unlicensed ISM band with the same functional principles: inductive coupling between two loop antennas.

The devices involved are an active initiator and a target, which may be both active - battery powered - or passive - draw the operating power from the initiator-provided magnetic field. NFC communications supports between two devices in close proximity - 10-20cm - at data rates of 106, 212 or 424 kbit/s. A NFC device can operate in three possible modes [71]:

(1) *Reader/Writer mode*: an active initiator can read/write data from/to targets detected in close proximity, such as passive RFID and NFC tags;

(2) *Peer-to-peer mode*: two active NFC devices can exchange information over a bidirectional half duplex channel, meaning that when one device is transmitting, the other one has to listen and should start to transmit data after the first one finishes;

(3) *Card emulation mode*: a portable NFC device such as a smartphone can act as a passive smart card to be read by an active reader.

NFC technology is integrated in every modern smartphone and widely employed as to perform contactless payments, ticketing and access control [71]. NFC plays an important role in IoT as a technology enabler [72]: smart objects are often equipped with NFC technology to replace the pairing of Bluetooth-enabled devices or the configuration of a Wi-Fi network through PINs and keys, by simply touching the devices.
NFC is subject to security concerns similar to those of RFID. The close range required by NFC communication does not make it immune to eavesdropping, which, with the right equipment, can be carried out at distances up to 10 meters; RFID jamming devices can transmit a signal that interfere with the transmission between a mobile NFC phone and a reader of a service provider, causing denial of service; malicious NFC tags that contains false information can be used to replace legitimate tags in a phishing attempt [73].

3 Bluetooth Low Energy

Bluetooth Low Energy (BLE) was defined for the first time in 2010 by the Bluetooth SIG as part of the Bluetooth 4.0 specification, it’s main feature being offering similar performances of classic Bluetooth in terms of range and data rate, while significantly reducing the power consumption.

3.1 Stack Architecture

The BLE stack (fig. 3.1)[74][75] is composed of two major blocks, the *Host* and the *Controller*, which act as separate logical containers in the architecture; the Host and the Controller communicate through the *Host Controller Interface* (HCI). This allows a Bluetooth system to consist of host and controller components from different manufacturers.

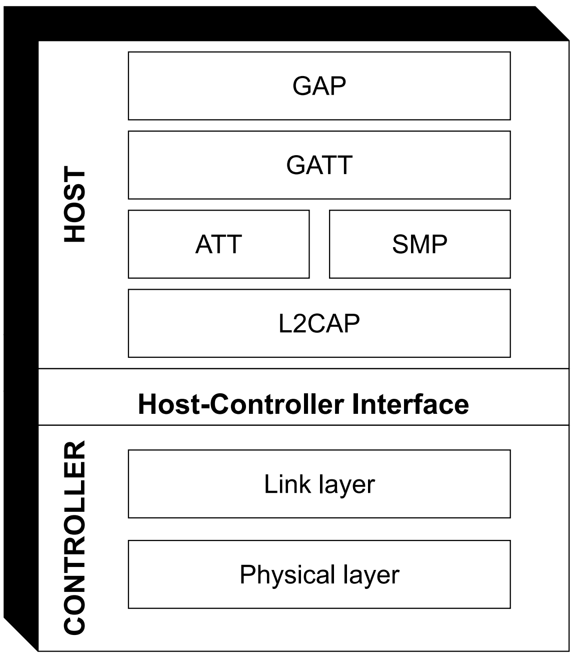

Figure 3.1: Bluetooth Low Energy protocol stack [75].

Physical layer

The *Physical layer* defines all the aspect of BLE technology related to the use of RF, such as modulation schemes, frequency bands and channel use. BLE operates in 2.4 GHz ISM band but instead of the classic Bluetooth 79 1-MHz channels, Bluetooth Low Energy has

40 2-MHz channels. These channels are divided into three advertising channels, which are used for device discovery, connection establishment and broadcast transmission, and 37 data channels, used for bidirectional communication between connected devices. The modulation scheme used by BLE is the Gaussian Frequency Shift Keying (GFSK).

Link layer

BLE devices can communicate according to two different patterns supported by the *Link layer*:

(1) *Advertiser/Scanner*: one device, the advertiser, broadcast the data unidirectionally in the advertising channels; the other device, the scanner, can receive the data. Also called connectionless communication;

(2) *Master/Slave*: the advertiser and the scanner establish a bidirectional connection and adopt the slave and master roles, respectively. The master can connect to multiple slaves at the same time.

Host Controller Interface

The HCI defines a standardized interface via which a host can issue commands to the controller and a controller can communicate with the host.

Logical Link Control and Adaptation Protocol

The *Logical Link Control and Adaptation Protocol* (L2CAP) provides connection-oriented and connectionless data services to the upper layer protocols with protocol multiplexing.

Attribute Protocol

The *Attribute Protocol* (ATT) defines the server and client roles. The server exposes a series of composite data items known as attributes to a client in a peer device. Attributes are identified by a unique index value, so that they can be referenced by the client; a *Universally Unique Identifier* (UUID) that identifies the attribute type; a set of permissions that indicate whether read, write or both forms of access are permitted; and a value, which is a byte array that contains the attribute's value.

Generic Attribute Profile

The *Generic Attribute Profile* (GATT) defines a framework that uses the ATT for the discovery of services, and the exchange of characteristics from one device to another. A characteristic is a set of data which includes a value and properties. The data related to services and characteristics are stored in attributes.

Security Manager Protocol

The *Security Manager Protocol* (SMP) supports the execution of security related procedures such as pairing, bonding and key distribution, and provides a cryptographic toolbox for security functions.

Generic Access Profile

The *Generic Access Profile* (GAP) defines the generic procedures related to discovery, link management, and security aspects for communication between BLE devices.

3.2 Specifications

The Bluetooth 4.0 specification [76] doesn't allow a slave node to take part in multiple connections simultaneously with other masters. Therefore, the only network topology supported by BLE is the star topology.

Despite the vibrant interest displayed by both industry and academia, BLE technology was falling short to populate the wireless home automation market, only allowing star networks being the bottleneck. Mesh networks, like those employed by BLE main competitors - *e.g.*, ZigBee and Z-Wave - proved to be more efficient in domestic context, where multiple smart objects in different rooms are hard to reach in a single hop, and communication between two non-adjacent end nodes may be required. The same problem can be observed in industrial, agricultural and urban scenarios, where direct communication between end devices may not be possible and the nodes are deployed over an area greater than the reach of a single device.

In 2013, the release of the Bluetooth 4.1 specification [77] introduces a fundamental change with regard to BLE mesh network support. A slave node is now allowed to be simultaneously connected to more than one master. In addition, one device can act both as slave and as a master, keeping parallel communications with its neighbors.

This game changer extends the possibilities in term of network topology by allowing devices to take part in mesh networks. At this point there is no official specification about the implementation of mesh networking for BLE technology, and multiple solutions are studied and proposed by both vendors and researcher.

Bluetooth 4.2 (2014)[78] and 5.0 (2016)[79] incorporate improvements in terms of range, data rate, security and advertising channel functionality, but do not offer further functionality to support BLE mesh networks.

The big step is taken in 2017 with the release of the the Bluetooth Mesh Profile specification [80], which defines fundamental requirements to enable an interoperable mesh networking solution for BLE technology.

3.3 Mesh Networks

A wireless mesh network consist of multiple nodes connected together in a many-to-many fashion; nodes can communicate with each other even if not directly connected, by sending messages that can hop through nodes in the network to reach their destination. Mesh networks are effective approach to providing coverage of large areas, extending range and providing resilience.

Routing vs. Flooding

Since the release of Bluetooth 4.0 there as been multiple proposal for implementing mesh networking with Bluetooth Low Energy devices, that can be classified in two distinct groups, based on the multihop paradigm adopted [81].
BLE devices can send data in two different ways, that is by broadcasting - advertising - the data or by establishing a connection. This duality makes way for two possible approaches:

- *Routing-based* approach: the data is forwarded through the network over the data channels by establishing a connection. A routing algorithm is used at every step to pick the next hop in order to reach the target node;
- *Flooding-based* approach: the data is broadcasted over the advertising channels to all the neighbouring nodes, which in turn broadcast the message until the target is reached.

The strong point of Flooding-based mesh networks is their simplicity: it is not required to establish a connection, and there is no need to employ complex routing protocols. This means no delay due to route discovery and less memory usage for storing and maintaining routing tables. On the downside, since the messages are broadcasted through all the network, flooding-based solutions suffer poor message throughput, which worsen with the increase of network size. The other issue is with security, since SMP services are only available over data packets sent through the data channels, and not over advertising packets [81].

Routing-based solutions instead employ a routing algorithm to find a route to deliver a data packet from one node to another. These solutions may use advertising channels for neighbor discovery and route formation but the data is exchanged over the data channels after a connection is established. Many routing-based solutions exploit the capability, introduced with Bluetooth 4.1, of performing multiple connections as both a slave and a master, while other, mostly older, approaches rely on one connection at a time.
Routing algorithms often require the construction and maintenance of routing table, which is a cost in overhead.

Bluetooth mesh architecture

Starting from the summer of 2017, with the release of the Bluetooth Mesh Profile specification [80], support for mesh network topology is officially available for BLE devices. The document defines the Bluetooth mesh stack as a layered architecture built on top of the BLE stack (fig. 3.2) [82].

Bearer Layer

The *Bearer Layer* defines how network messages are transported between nodes.
At the moment of writing the specification defines two mesh bearers over which mesh messages may be transported:

- the advertising bearer, used to send mesh packet;
- the GATT bearer, using to allow older BLE devices, which do not support the advertising bearer, to take part in a Bluetooth mesh network.

Network Layer

The *Network Layer* defines how transport messages are addressed towards one or more elements. It also decides whether to relay/forward messages, accept them for further

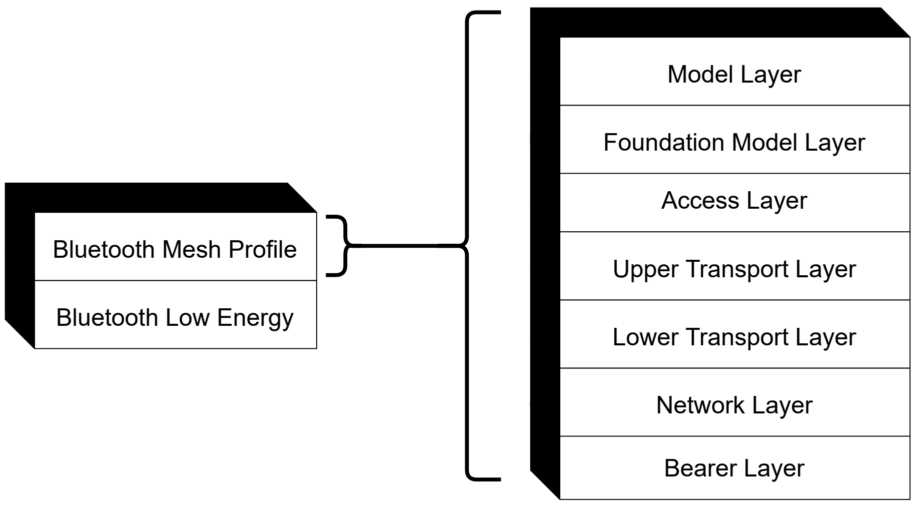

Figure 3.2: Bluetooth Mesh stack [80].

processing, or reject them, and defines how a network message is encrypted and authenticated.

Lower Transport Layer

The *Lower Transport Layer* takes PDUs from the *Upper Transport Layer* and sends them to the Lower Transport Layer on a peer device; it defines how Upper Transport Layer messages are segmented and reassembled into multiple Lower Transport PDUs in order to deliver large Upper Transport Layer messages to other nodes.

Upper Transport Layer

The Upper Transport Layer is responsible for the encryption, decryption and authentication of application data passing to and from the Access Layer.

Access Layer

The *Access Layer* defines the format of the application data and how higher layer applications can use the upper transport layer.

Foundation Layer

The *Foundation Model Layer* defines the states, messages, and models required to configure and manage a mesh network.

Model Layer

Finally, the *Model Layer* defines the models, that is standard software components that, when included in a product, determine what it can do as a mesh device [83]; a model defines the implementation of behaviors, messages, states, state bindings and basic functionality of nodes on a mesh network.
The Bluetooth Mesh Model specification [84] is a document that defines over 50 models for different applications, such as lightning, sensors and even models that are "deliberately positioned as generic, having potential utility within a wide range of device types" ([85]).

Bluetooth mesh operation

Nodes

Devices which take part in a BLE mesh network are called nodes, and they are the building blocks of the network. All nodes are able to receive and transmit messages, but in addition they can also be of one or more of these types:

- *Relay* nodes are able to retransmit the received messages over the advertising bearer. Relaying is the mechanism by which a message can propagate through the network, hopping from one node to another;
- *Proxy* nodes are able to transmit messages between GATT and advertising bearers. This functionality allows devices which posses a BLE stack but not a Bluetooth mesh stack to take part in a BLE mesh network, by connecting to a Proxy node;
- *Low Power* nodes are able to take part in a mesh network at low power consumption by operating at significantly reduced receiver duty cycle. This is only possible on conjunction with a Friend node. Low Power nodes are usually powered-constrained devices. A Low Power node can connect to only one Friend node;
- *Friend* nodes enable Low Power nodes to operate in a mesh network. A Friend node stores the messages directed to the Low Power node and send them to it whenever its possible - that is, whenever the Low Power node polls the Friend node. A Friend node can be connected to multiple Low Power nodes.

Unlike other low power mesh networks, such as those offered by ZigBee and Z-wave, Bluetooth mesh do not require a centralized controller: the messages hop across the network to reach their destination without the need to pass through a coordinator, which results in reduced latency and less overhead. Bluetooth mesh allows for a vast and diversified network (fig. 3.3), to which multiple devices of different kinds can take part: devices powered by a small battery can participate in the network through a Friend node without compromising battery life and interoperability is extended to older devices thanks to Proxy nodes.

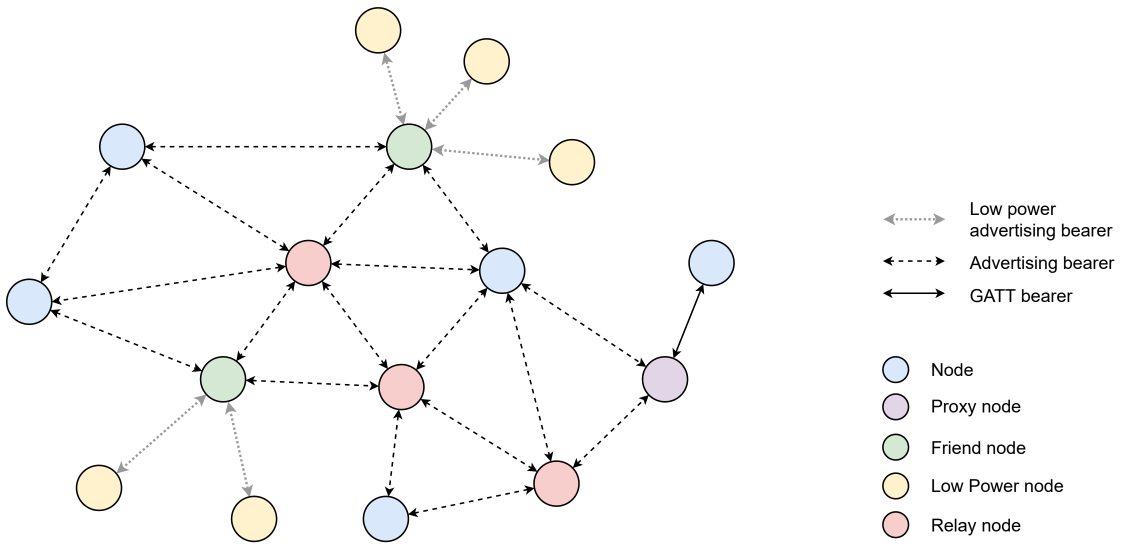

Figure 3.3: Example of a Bluetooth Mesh topology [80].

Elements and States

Every node in the network contains one or more *elements*; an element is an addressable entity within a node, and represent a component of the device that can be controlled independently. For example, a smart lightning product that has three separate lightbulbs is represented in a mesh network as a node containing three elements, one for each lightbulb.

The status of an element is described by one or more *states*; states are data items with one or more values that indicate the condition of the element. For example, a smart light controlled by a dimmed switch would possess two states, one to indicate whether the light is on or off and one to represent the brightness level.

When a state changes its value it's called *state transition*.

Sometimes a state transition may trigger a change in another state. This kind of relationship between states is called *state binding*. In the previous example, changing the value of brightness level to zero will trigger a state transition in the OnOff state from on to off.

Models

Models encapsulate the previous concepts to define some or all the functionalities of an element. Models can be of three types:

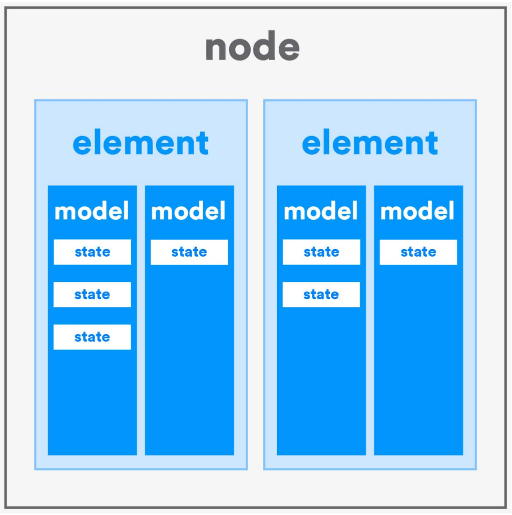

Figure 3.4: Bluetooth Mesh node composition [83].

- *Server models* define a collection of states, state transitions, state bindings and messages which the element containing the models may send or receive. They also define behaviors relating to messages, states and state transitions;
- *Client models* do not define any states. Instead, they defines the messages which they may send or receive in order to request, change, or obtain the value of the corresponding server model states.
- *Control models* encapsulate both a server model and a client model.

For example, a simple binary light switch contains an element, which represents the switch, whose functionality is defined by the Generic On/Off Client model. This model controls the Generic On/Off Server Model, which defines the functionality of a simple light, by sending it messages.

The Bluetooth Mesh Model specification [84] defines states, state transitions, state bindings and messages of over 50 models that describe the functionality of multiple elements spanning over different devices and applications.

Communication

Communication within the network is achieved via messages. Messages operate on states and for each state there is a defined set of messages that a server supports and that a client may use to request the value of a state or to change a state (fig. 3.5).

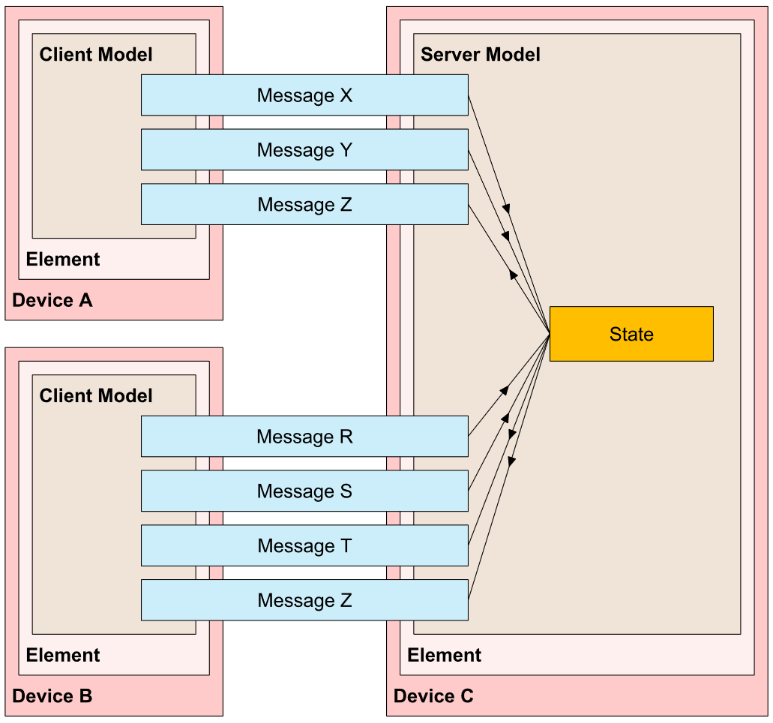

Figure 3.5: Bluetooth client-server model communication [80].

Messages can be acknowledged or unacknowledged. The former requires a response from the nodes that receive them to confirm that the message was delivered, while the latter doesn't.

Messages must be sent to addresses, which can be of three types:

- *Unicast addresses* identify a single element of a node, and are assigned during the provisioning process;
- *Group addresses* identify multiple elements spanning over one or more nodes. The Bluetooth SIG defines four Fixed Group Addresses named All-proxies, All-friends, All-relays and All-nodes.
  Group addresses may also be assigned dynamically by the user: for example, a group address may identify all the lights in a room in order to control them with a single switch;
- Virtual addresses identify multiple elements spanning over one or more nodes and are usually preconfigured by the manufacturer.

The exchange of messages in the network is defined as using the publish/subscribe paradigm (fig. 3.6). A node publish messages to an address - unicast, group or virtual - and nodes that are interest in receiving the messages will subscribe to these addresses.

Nodes may subscribe to multiple addresses.

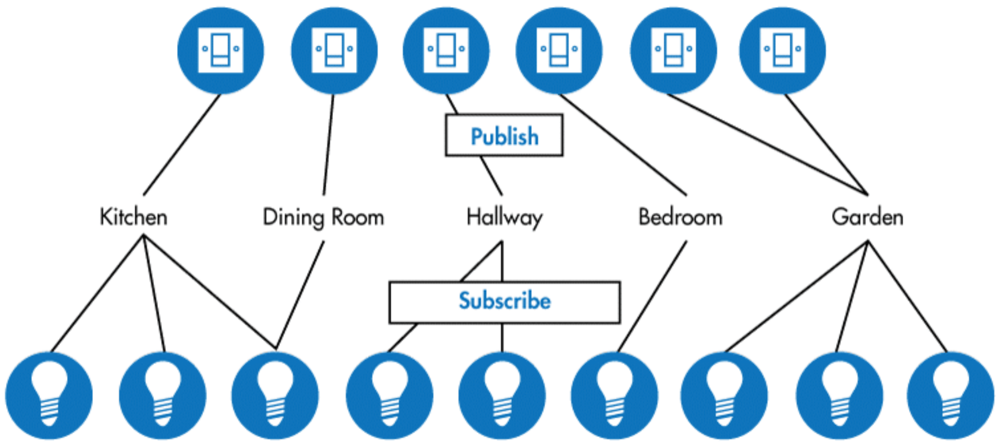

Figure 3.6: Example of publish/subscribe communication in a BLE mesh [82].

Provisioning

A device that is not member of a mesh network is called an *unprovisioned device*. Adding an unprovisioned device to the mesh network is a process called *provisioning*. This process is started by the unprovisioned device, which advertise its presence to the *Provisioner* - usually a smartphone. The Provisioner then invites the device to the network, which is followed by the exchange of public keys. Then the Provisioner requires an authentication: the user must enter into the Provisioner a random number that is output by the new device. After the provisioning has been completed, the provisioned device possesses the network key, which is used to secure and authenticate messages at the network layer: the device is now part of the mesh network and therefore a node.

Managed flooding

Managed flooding is the protocol chosen by the Bluetooth Mesh Working Group to allow nodes to exchange messages in a mesh network. Flooding is a technique based on advertising the message to all the neighboring nodes, which in turn will relay the message to other nodes, until the destination node is reached.

With respect to other flooding techniques, managed flooding offers some improvements:

- Messages are assigned a *time-to-live* (TTL) field, which limits the number of times that the message can be relayed. The TTL field is decremented every time the message is relayed until it reaches zero, then the message does not get relayed further. The TTL field optimize the overall power consumption of the network by preventing a message from being transmitted further than is required;

- *Heartbeat* messages are sent periodically to signal to other nodes in the network that the sender is still active. Heartbeat messages contain data which allows receiving nodes to estimate how far away is the sender in terms of number of hops required to reach it; this data can be used to tune the TTL field;
- Messages are cached by all the nodes. Every node has a cache that contains the message that the node has received and sent recently. Whenever a node receive a message that is present in its cache it discards it. This way messages are prevented from being transmitted multiple times by the same node;
- *Friendship* is the relationship between a Friend node and a Low Power node, allowing the latter to operate in a power efficient way.

4 Toit Platform

Toitware ApS is a Danish company founded in February 2018 by former Google software engineers Kasper Lund, Erik Corry, Florian Loitsch and Anders Johnsen.
Their product is the Toit Platform, a software development and deployment platform for IoT that allows for high-level programming of embedded devices without compromising on performance, with focus on connectivity and edge computing. Toit's mission is to make IoT developing accessible to everyone by providing a simple and efficient programming language and taking care of all the thorny aspect of embedded programming, such as connectivity, over-the-air software and firmware update and fleet management [86].

Toit Language

Toit applications are written in the Toit language, a high-level language with a syntax similar to Python's. When developing code for embedded devices the most common approach is to programming in native C language. This allows to access low-level operation such as bit wise data manipulation and precise memory management, while achieving high performances due to little to no overhead. However, C is a low-level language that might be hard to learn and require technical hardware knowledge when used for embedded systems programming. Simple functionalities take time to build and modern coding technique might be difficult to implement.
On the other hand, the use of high-level languages such MicroPython and Java SE Embedded implicate higher memory usage, higher power consumption and much lower performances. This is not ideal in IoT applications where the smart objects are typically battery-powered, low-end devices.
With these premises, Toit spent one year putting together an approachable high-level, object-oriented programming language that runs code efficiently on constrained devices. Toit simple syntax and recognizable programming style make it quickly to learn and easy to use. It comes with ready-made reliable libraries to access low-level functionalities. It is declarative, statically analyzable, memory safe and garbage collected [87]. The programs are compiled to compact binaries that execute 30 times faster than MicroPython [86].

Toit Virtual Machine

The Toit firmware is composed of the ESP-IDF operating system and the Toit Virtual Machine (VM) built on top of it [88]. The code developed runs on the Toit VM as one or more applications in a sandboxed environment - that is, Toit programs can't write to arbitrary memory locations. The applications run isolated from one another and from the underlying hardware. This means that one application's crashing does not constitute a problem for the system and the other applications, which keep running. Separate specification files controls the execution of each application, allowing for multiple applications running side by side.

Not only that, this means that applications can be installed and uninstalled on a device without affecting the execution of the other applications already installed on the platform. The traditional approach requires that all applications are compiled, linked and deployed

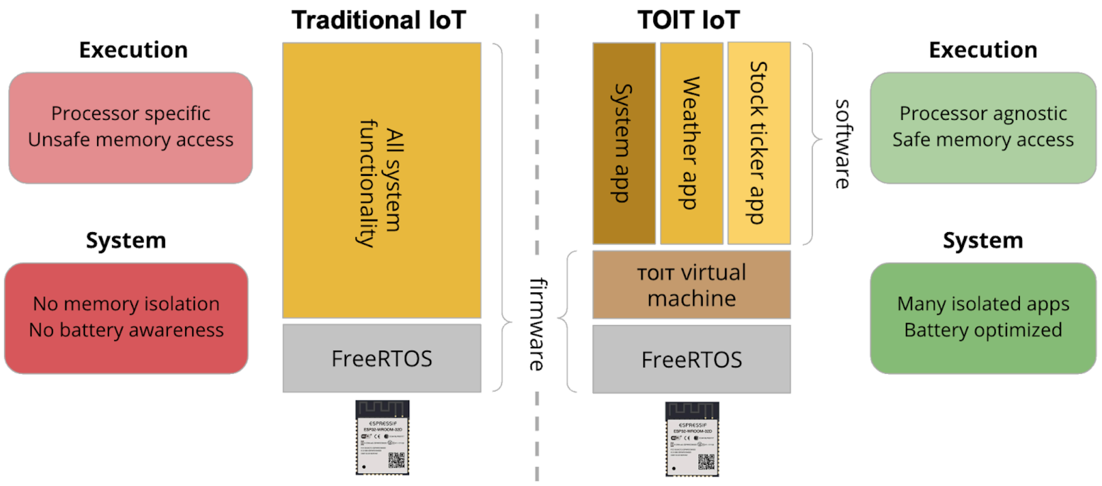

Figure 4.1: Architecture of the Toit Platform (right) in comparison with traditional firmware deployment (left) [88].

together, which makes the code prone to errors and hard to maintain; Toit VM offers a more flexible and robust alternative to deploy and run software that allows the developer to focus only on the end goal and not the hardware technicalities.

Toit Cloud and API

Devices provisioned with the Toit firmware can be organized through the Toit Cloud, which gives both a clear overview and detailed information about the health and status of the fleet [89]. The Cloud takes care of scheduling over-the-air updates of both firmware and software for online and offline devices alike. Software and firmware updates are in the form of small patches ranging from 40 to 500KB, meaning that they can be quickly deployed even over an unstable Wi-Fi or cellular connection; the connectivity logic in the Toit firmware is built as a stand-alone feature, completely isolated from the application code, meaning that updates can be safely carried out even in the event of connectivity drops [90].

Toit devices can communicate with each other through the Toit Cloud using an out-of-the-box, publish/subscribe messaging service. An application can publish data on a certain topic, the data is sent to all the applications subscribed to that topic. Since Toit connectivity is isolated from application code, a Toit app focuses only on producing data and saving it on the device. The data is then uploaded to the cloud each time a device comes online, without loss of data over connectivity issues [90].

All communication between the device and the cloud is end-to-end encrypted using modern public-key encryption [89].

Toit offers public API to give full programmatic control of the devices and access to the data published by Toit applications [89].

Hardware

The Toit Platform runs on the ESP32 chip from Espressif [91].

The ESP32 System-on-Chip (SoC) is designed for ultra low-power consumption, mobile, wearable electronics, and IoT applications. It's capable of functioning reliably in industrial environments, with an operating temperature ranging from –40°C to +125°C. It mounts two Xtensa 32-bit LX6 microprocessors that runs at 240 MHz and 520KB of RAM [92][93].

It comes with 34 GPIO pins for peripherals, configurable to be used for communication protocols (SPI, I2C, UART), analog/digital interfacing (ADC/DAC), Ethernet and PMW.

It has built in Wi-Fi and Bluetooth modules, and since 2019 it has officially passed the SIG Bluetooth LE 5.0 certification.

The ESP32 SoC has been chosen by Toit because it perfectly fulfills the characteristics of a battery-powered smart device in most of IoT applications, that is:

- Low-power consumption and advanced power-management technologies: with five different power modes the ESP32 can run on standard AA batteries for years;
- Cost-effective: ESP32 chips are low cost, which means that a large scale deployment would be less expensive compared to other MCUs;

- High connectivity: ESP32 offers both wireless and wired connectivity options, making it easy to build an IoT infrastructure around them, regardless of the use case;
- Computing power: the dual-core microprocessor enable more processing and control of the data at the edge, before sending it to the cloud.

Use cases

Toit provides a cheap solution for a large scale deployment of smart devices with enough computing power to manage multiple sensors and actuators, process the data gathered by the sensors and forward it to the cloud; Toit takes care of all the pain points in IoT development leaving the developers to focus only on the applications that run on the smart devices. This allows business who operate in IoT to cut their expenses in hardware and connectivity gateways while reducing the workload of the developers.
Example of real use cases of the Toit Platform are:

- Consibio [94] is a tech-spinout company that builds custom hardware for monitoring and optimization of bioprocesses. Their typical product consist of several sensors mounted on microcontrollers, connected to a Linux gateway, working with an established IoT platform that ensured connectivity to the cloud. The different units had to be programmed using various programming languages and it was not possible to update the sensors connected to the gateway.
  Adopting the Toit Platform allowed Consibio to have all the components connected only to the microcontroller, with no need for a gateway (fig. 4.2). Other than that Consibio reported reduced cost in hardware and time spent programming the devices, along with the other benefits such as lower power consumption, secure and quick over-the-air updates and remote managing of the deployed fleet.
  Toit-powered devices have been deployed in a biofactory that grows insects for sustainable protein production. The devices measure the temperature and humidity of the farm, making sure that the optimal condition for insect growth are met, and send an SMS alert otherwise. They also operate an industrial valve that controls the airflow when the presence of malodorous air is detected.
- Trifork [95] is a IT and business service provider that supply high-quality custom-built applications and end-to-end software solutions. One of their product, the Foodbox,

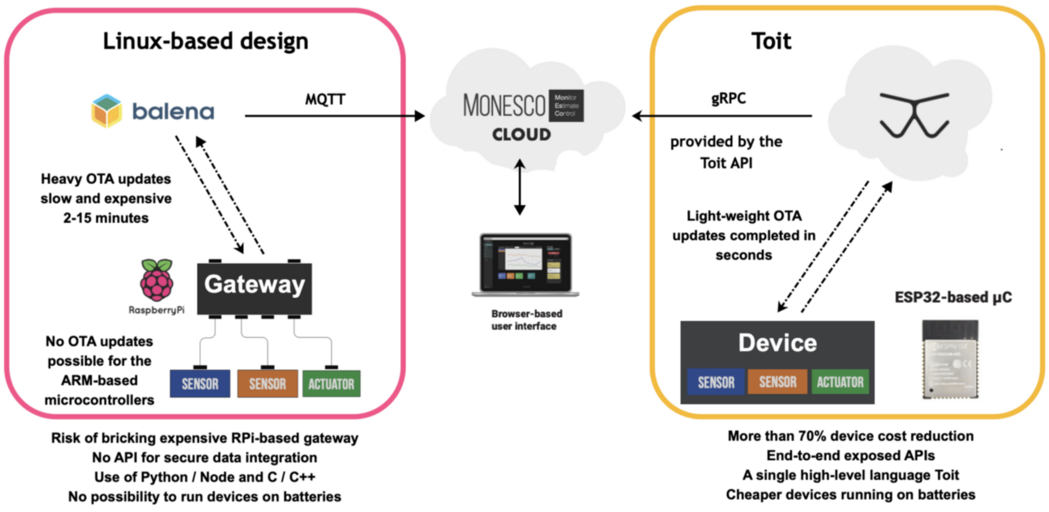
Linux-based design
balena
MQTT
Heavy OTA updates slow and expensive 2-15 minutes
RaspberryPi
Gateway
No OTA updates possible for the ARM-based microcontrollers
SENSOR
SENSOR
ACTUATOR
Risk of bricking expensive RPi-based gateway
No API for secure data integration
Use of Python / Node and C / C++
No possibility to run devices on batteries
MONESCO
CLOUD
Browser-based user interface
Toit
gRPC
provided by the Toit API
Light-weight OTA updates completed in seconds
ESP32-based µC
Device
SENSOR
SENSOR
ACTUATOR
More than 70% device cost reduction
End-to-end exposed APIs
A single high-level language Toit
Cheaper devices running on batteries

Figure 4.2: Design of Consibio product before (left) and after (right) adopting Toit [94].

is a growing chamber for vertical - also known as aeroponic - indoor farming. In aeroponics farm, the roots of the plants are exposed to air and get their nutrients from a solution sprayed as mist, needing only 5% of the water used in a traditional farming system. The exposed roots have access to more oxygen than when submerged in water and thus the plants grow faster, however they are very sensible to changes in humidity, therefore the environmental conditions of the growing chamber must be monitored constantly.

The Foodbox cabinet contains several shelves for the plants to grow, and on the lowest shelf there is a system made of a reservoir, a pump, a pressure accumulator, and spraying nozzles to spray the nutrient solution into mist. The cabinet also contains temperature and humidity sensors and an artificial lightning system.

The condition monitoring is done by a single Toit-powered ESP32 that runs three applications. One app measures the temperature and relative humidity in the Foodbox. A second app controls the lighting. A third app is used to schedule the spraying of the nutrient solution.

Using the PubSub service the applications logs the measurements on the cloud, that is accessed by a smartphone app thanks to the public APIs.

Thanks to the Toit Platform, the design of the cabinet is modular and easy to maintain: by adding extra sensors is possible to include the monitoring of the quality of the nutrient solution and the water level, with an alarm triggered when the latter is running low.

The application that take care of this can be easily installed over-the-air without altering the execution of the other applications.

5 Implementation of BLE mesh network with Toit

Motivation

Despite the fact that the ESP32 SoC has passed the Bluetooth 5.0 certification [96] and the ESP-BLE-MESH implementation is fully Bluetooth SIG-certified [97] since 2019, at the time of writing the Toit Platform does not support all the BLE features. That is, at the moment a Toit device could not take part in a Bluetooth mesh as defined by the official specification.

Since Toit devices work mostly as edge gateway devices, it is of interest to study a solution to allow a wireless network of Toit-powered devices to communicate without requiring the data to pass through the cloud.

The Toit Platform performs greatly under the assumption that all devices are under Wi-Fi or cellular coverage in order to communicate wirelessly with other Toit devices and the cloud. While this assumption holds true in most applications targeted by the Toit product, this may not always be the case.

For example, only a few devices of a wireless sensor network deployed inside a large building may be within the reach of Wi-Fi, and a wired connection to collect the data from the end nodes may not be possible. In this case it is of interest to employ a communication solution to collect data from the edge nodes, without having to pass through the cloud.

This chapter will propose an implementation of a mesh network to allow multiple Toit devices to exchange messages over Bluetooth even if not directly connected.

Since the BLE functionalities available with Toit are limited - it's not possible for a slave to connect to multiple masters, and for a node to act as both a slave and a master - the solution proposed will employ only the primitive functions to advertise, scan, connect and access read/write characteristics of a server.

5.1 Overview

The solution proposed is inspired by the tree network topology presented in [98]. The structure of the network is that of a undirected tree that is generated starting from a root node. The tree is composed of three types of nodes:

- The root node is a unique node; there is only one root node and the network expands from the root. The root act as a sink in the network, that is, all the messages pass through the root. In the previous example, the root node is represented by a Toit device under Wi-Fi/cellular coverage that act as a gateway device. Cloud connectivity for other nodes is not required, as all the messages pass through the root node;
- The intermediary node is a node connected to exactly one node situated in a higher level of the tree, and to one or more nodes in a lower level. Intermediary nodes have the main function of relaying messages through the tree but they can also incorporate sensors and actuators that produce and consume messages;
- The leaf node is a node that is only connected to exactly one node situated in a higher level of the tree. Leaf nodes are typically sensors and actuators that produce and consume data.

Every node is identified by a 2-byte address that is assigned by a higher level node during the tree generation phase, except for the root node, which has address equal to 0. Every node advertise its address, which is used by sender nodes to route the messages in the network. The structure of the network and the addressing is presented in fig. 5.1.

Direct communication in the network is only possible from parent to child and vice versa. The main advantage of this restriction is that the routing algorithm is extremely simple: the next node can be easily selected by looking at the address pattern. The other advantage is that no routing table is required, as one node can only send messages to its parent node or its children nodes; there is exactly only one path between two nodes.

The disadvantage, with respect to Bluetooth mesh, is that all the messages need to pass through the root and a node can forward messages only to its descendants and ancestors.

A node can send a message by:

(1) Scanning the neighbouring devices, looking for its parent or one of its children;

(2) Connecting to the receiver node. The receiver node is connected as slave, the sender node is connected as master;

(3) The sender node writes the message in the characteristics of the receiver;

(4) The sender node disconnects from the receiver.

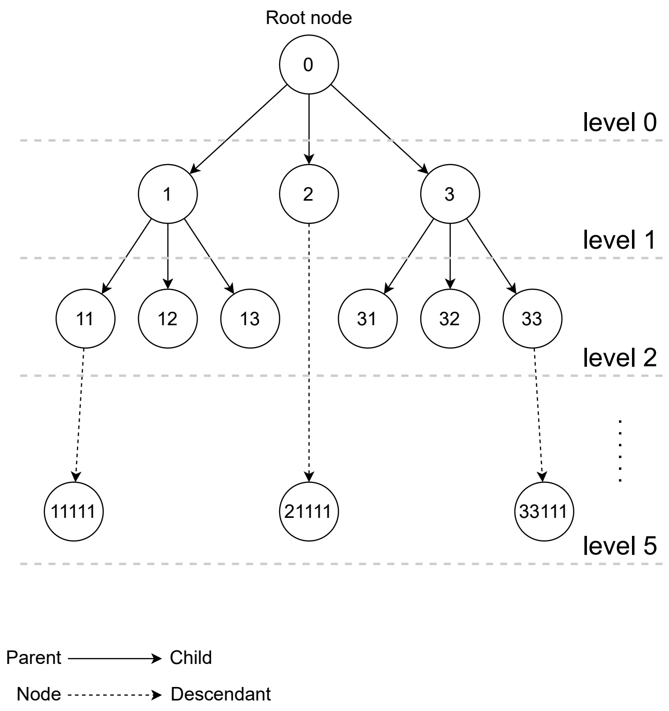

Figure 5.1: Tree network structure and addressing.

Whenever a node receives a message that is directed to another node, it forwards it to the next node in the path. Communication from a leaf node to the root is presented in fig. 5.2.

This implementation is meant to work in parallel with the code that produce and consume the data.

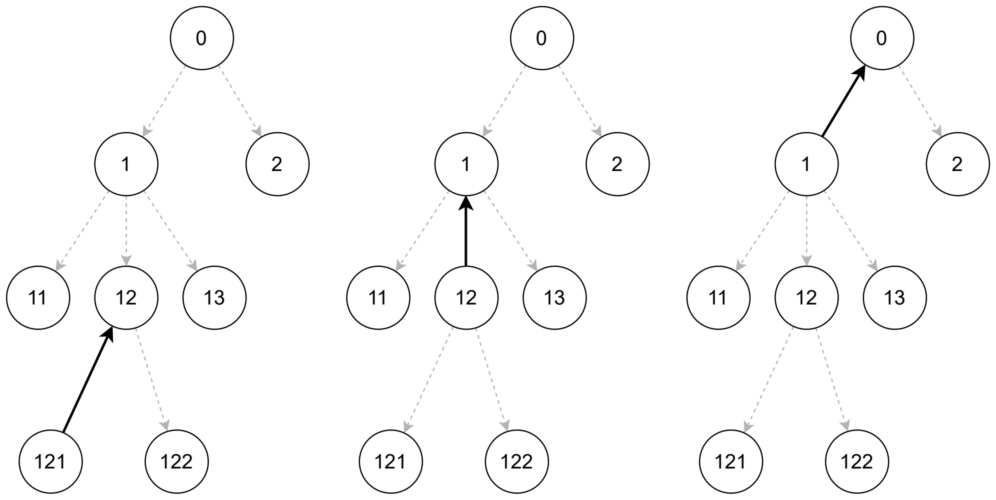

Figure 5.2: Message propagation from leaf node to root. The dashed lines represent parent-child relationship, while the solid lines represent master-slave active connection

5.2 Implementation details

5.2.1 Data structure

Communication in the network require all node to act as slaves, to receive a message, and as masters, to send a message. Therefore every nodes will expose a GATT service that contains the characteristics illustrated in table 5.1. The UUID for the characteristics have been randomly generated [99].

Table 5.1: Description of the characteristics exposed by the GATT service.

Characteristic	**UUID**	**Size (bytes)**	**Permission**	**Description**
ownAddr	0x3082	2	read/write	Address of the node
newMsg	0x936A	1	read/write	Indicates whether a new message has been received
msg	0xE51C	8	read/write	Received message

The structure of a message is a 8 byte array that contains the source address, the destination address and the payload (table 5.2); little endianess is used.
The OP code field allows to define behaviours in response to messages. For example the root can send a message to a node that triggers a response, like a data request.

Table 5.2: Tree message composition

Byte 7	**Byte 6**	**Byte 5**	**Byte 4**	**Byte 3**	**Byte 2**	**Byte 1**	**Byte 0**
destination address		source address		message payload			
				opcode	message data		

In addition, every node has an internal FIFO buffer where messages are stored before they are sent, and a 1 byte variable that indicates the number of children nodes. Whenever a message needs to be sent, it is simply inserted into the message buffer.

5.2.2 Network operation

The network operation can be described by a five-state machine (fig. 5.3). The device set as root starts with address equal to zero and kick-start the tree generation by provisioning the neighboring nodes.
All the other devices start with address 0xFFFF and proceed to the *Advertise state*.

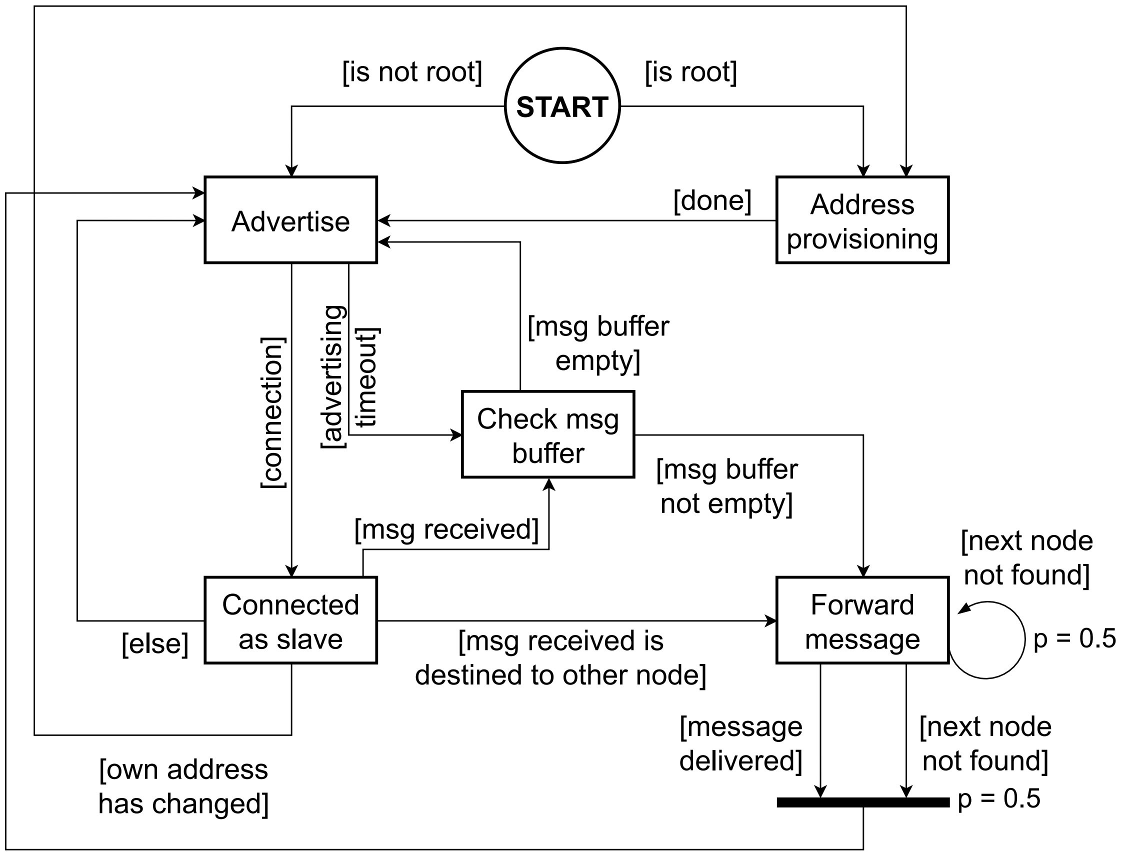

Figure 5.3: State diagram of the suggested BLE tree routing service.

Address Provisioning

This state is used to expand the network. The node starts by scanning the neighboring advertising devices. For each device with address 0xFFFF discovered, the scanning device establish a connection as master and assign an address to the slave by writing the ownAddr characteristic of the unprovisioned device. The address is calculated as $(ownAddr * 10) + i$, where i is a 1-byte variable that tracks the number of children, and is incremented at every iteration; this makes for an easy identification of parent node and children nodes. By using two byte for the address, one tree can include up to 65535 devices over six levels.

When there are no more neighboring unprovisioned devices, the node proceed to the Advertise state.

Advertise

At this state the node advertise his address to the neighboring devices, waiting for a connection to be assigned an address or to receive a message. When a master device establishes a connection, the node transitions to the *Connected as Slave* state. If no connection happens before the advertising timeout - which is a parameter to be tuned - the device goes to the *Check Message Buffer* state.

Connected as Slave

When a node is connected as a slave, it waits for the master to disconnect. At this point it checks if the values of its characteristics have changed:

- If the newMsg characteristic is set to 0x10 it means that the node has just received a message. The node checks the destination address of the message:
  - If the destination address corresponds to the node own address, then the message has reached its destination. The node handles the message and then goes to the Check Message Buffer state;
  - If the destination address is different from the node own address, the message goes to the *Forward Message* state, in order to send the message to the next node.
- If the newMsg characteristic is set to 0x11, it means that the master has provisioned the node with a new address. The node transition to the Address Provisioning state to continue the generation of the tree;
- If none of the above, the node goes back to the Advertise state.

Forward Message

While in this state, the goal of the node is to forward the message to the next device. The node scan the neighboring devices. The structure of the tree makes routing easy, since there is only one path that connects two nodes.

- If the destination is at a higher level of the tree - this can be checked easily by verifying that $dstAddr < ownAddr$, then the node must forward the message to its parent. That is, the device whose address is equal to $ownAddr/10$;
- If the destination is at a lower level, then the node must find, amongst its children, the only one who could be an ancestor of the destination address. Let nd be 1 plus the number of digits of the node own address. Then the node must forward the message to the device whose address is equal to the first nd digits of the destination address.

If such a device is found, the node establishes a connection as master, writes the message

in the msg characteristic of the slave and sets its newMsg flag to 0x10. Then transition to the Advertise state. If the node is not able to find such device, it means that the latter is not advertising, either because it’s no longer alive in the network, or likely because it is trying to forward a message itself.
At this point, the node makes a random choice:

- with probability 0.5 it starts scanning again;
- with probability 0.5 it saves the current message on top of the message buffer, then proceed to the Advertise state.

Check Message Buffer

The node simply check if the message buffer is empty:

- If the message buffer is empty it means that there are no messages to be forwarded. The node transitions to the Advertise state;
- If the message buffer is not empty, it means that there is at least one message waiting to be forwarded. The first message in the buffer is removed and stored into memory, then the node transitions to the Forward Message state.

5.2.3 Messaging functionalities

The messaging functionalities offered by this implementation allow communication between every pair of nodes in the network.
Standard communication can only happen between ancestors and descendants, and reachability can be easily computed by comparing the node address with the destination address: whenever a node receives a message destined to an unreachable node - a node that is neither a descendant or an ancestor - the message is discarded. However, such unreachable nodes can be reached by passing through the root in a way that is called forwarding communication.
This thesis propose some basic messaging functionalities, which can be expanded to accommodate different applications. These functionalities are defined by the OP codes in the opcode 1-byte field of every message (see Appendix A).
The most basic type of message is a Data Message, which is simply a message containing some 3-byte data payload. This type of message requires no response from the receiver. In WSN applications, sensor nodes can be set to send a Data Message to the root node at a fixed time period.

Data Request and Response

To request data from a node, a Data Request message can be sent. The reception of a Data Request message will trigger the transmission of Data Response message. A Data Response message is only sent in response of a Data Request message, and is sent to the source address of the Data Request message.

Both Data Request and Data Response messages must be sent to descendants or ancestors.

Forwarding Communication

Forwarding communication allows to send messages to every node in the network by sending a message to the root: since every node is a descendant of the root, the latter will be able to forward the message to every node in the network.

This thesis propose a data request and response communication mechanism that allows a node to request data from every other node in the network in four steps (fig. 5.4):

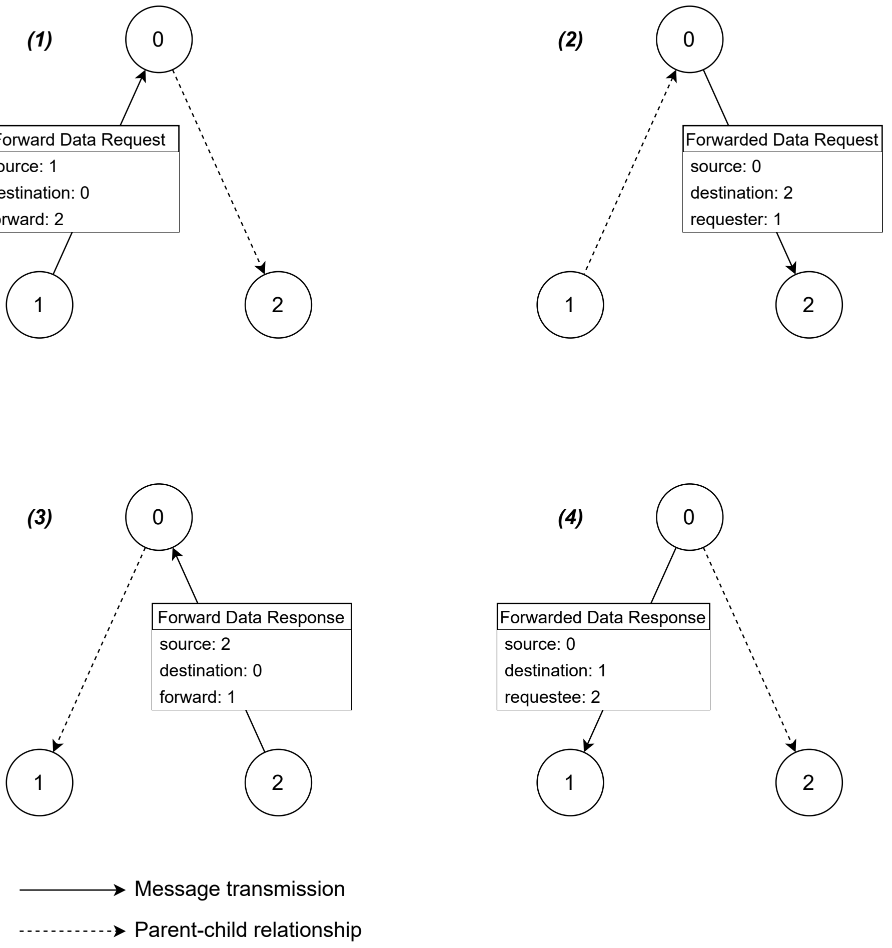

Figure 5.4: Four steps of forwarding communication

(1) The requester sends a Forward Data Request to the root. The forwarding address, that is the address of the requestee node, is stored in the message data field;

(2) The root receives the Forward Data Request and sends a Forwarded Data Request to the forwarding address. The address of the requester is stored in the message data field;

(3) The requestee node receives a Forwarded Data Request, which triggers the transmission of a Forward Data Response to the root. The requester address is stored in the message field, along with the requested data - which must fit in 1 byte;

(4) the root receives the Forward Data Response and sends a Forwarded Data Response to the requester. The message data contains the requestee address and the requested data.

Provisioning Request

A Provisioning Request message is a special kind of message which triggers the transition to the Address Provisioning state in the receiver. A node which receives a Provisioning Request message will resend the message to all of its children. This can be used to expand the network, by making the root sending a Provisioning Request to all of its children when new nodes must be added to the network. In this way, the new nodes will automatically be placed in the tree structure.

5.2.4 Tuning

The operation of the network can be tuned to accommodate different application by changing the values of some parameters:

- *Provisioning Scan Duration*: it's the duration of the scan during the Address Provisioning state, that is, the amount of time that a node spends looking for neighbouring unprovisioned devices;
- *Forwarding Scan Duration*: it's the duration of the scan during the Forward Message state, that is, the maximum amount of time that a node spends looking for the next hop to transmit a message - the scan stops as soon as the next hop is found;
- *Advertising Duration*: it's the duration of the advertising during the Advertise state, that is, the amount of time that a node spends advertising it's address. This parameter influences the most the operation of the network and ultimately bottleneck the throughput of the nodes. A high value makes it easier to find and connect to a node but decreases the message throughput, as buffered messages will sit in the buffer

for the whole duration of the Advertise state. A low value will increase the throughput but will make it difficult for other nodes to find and connect to each others;

- *Keep Looking Probability*: it's the probability that a node in the Forward Message state will stay in the Forward Message state when the next hop is not found. This prevents a deadlock situation, whenever two nodes are both scanning looking for each other, by making them randomly choose whether to keep scanning or advertising their address (see section 5.3.2).

The value of these parameters adopted by this implementation is illustrated in table 5.3

Table 5.3: Value of the tuning parameters

Parameter	Value
Provisioning Scan Duration	500 ms
Forwarding Scan Duration	700 ms
Advertising Duration	4000 ms
Keep Looking Probability	50 %

5.3 Practical issues

5.3.1 Latency

The main issue with this implementation is that a connection must be established and broken multiple times in order to deliver a message. The time required to scan, connect, write and disconnect makes up for a greater latency with respect to other solutions, where the devices keep their connection active. Unfortunately, the Toit platform does not implement the possibility for a node to act as both a slave and a master yet.

5.3.2 Starvation

The network structure restrict a pair nodes to be involved in only one message exchange at a time. A deadlock situation may happen if two nodes, both carrying a message, are scanning the network looking for each other (fig. 5.5). The deadlock is avoided by making the nodes randomly choose between storing the message and advertising or keep scanning, whenever they fail to find a receiver. However, this may cause "*unlucky*" nodes, who never get a chance to connect as masters, to pile up undelivered messages, leading to congestion in the network. If the nodes produce data at a quick rate this may lead to starvation, heavy delays and messages delivered in non-chronological order. This problem can be partially solved by implementing a priority system that evaluates how much time a message has spent in the buffer, how many time a node has failed to find a receiver and the direction of message to influence the decision to keep looking for a receiver or going back to advertising.

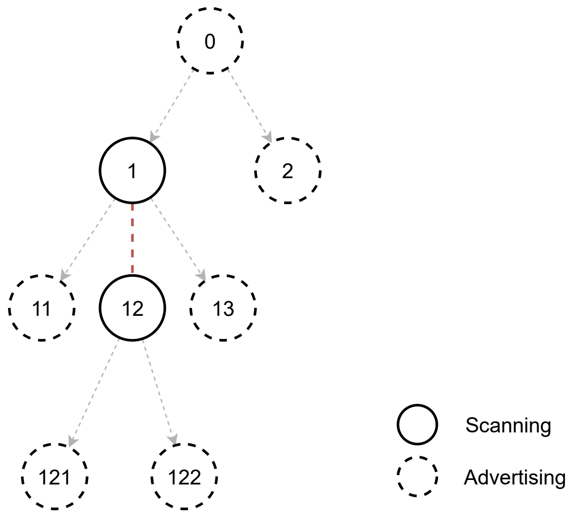

Figure 5.5: Deadlock situation where two nodes are both scanning, looking for each other. Node 12 is scanning for node 1, in order to deliver a message to the root; node 1 is scanning for node 12 in order to deliver a message to node 121. The passage between node 1 and 12 is blocked and all the messages that need to cross it pile up.

5.3.3 Robustness and Security

This implementation does not take into account security concerns. The tree structure of the network suffers from single-node failure, meaning that if one single node becomes inactive, the whole sub-tree of devices becomes unreachable. The network lacks self-healing capabilities, that is, it's not able to rearrange itself to cope with nodes becoming inactive. Although the messages are sent over data channels, the simple addressing pattern allows for a device to insert itself in the network by guessing an address and advertising it. Other than that, advertising nodes accept all connections, therefore an attacker device can just connect to a random node and keep it occupied, blocking every message from passing through.

6 Evaluation

In this chapter the tree network implementation proposed in the previous chapter will be evaluated in terms of time performances and functionalities.

6.1 Experimental Setup

The implementation will run on three M5Stack Core 2, which is a IoT-oriented device that uses an ESP32 model D0WDQ6-V3 as MCU. In addition to the feature of ESP32, the M5 device mounts a 2.0-inch capacitive touch screen, USB type-C interface, RTC module and a 390mAh battery.
These devices are programmed with Jaguar, a Toit application that uses the Toit VM to update and restart the code on ESP32 over Wi-Fi.
The network will be tested in three different configuration to evaluate the messaging functionalities (fig. 6.1), that is:

(1) Single-hop communication: direct communication between two nodes, *i.e.,* parent-child communication;

(2) Multi-hop communication: communication between two nodes that are separated by one or more level in the tree, *i.e.,* ancestor-descendant communication;

(3) Forwarding communication: communication between two nodes passing through the root;

Latency between transmission from source node and reception at destination node will be measured using the built in RTC module: the devices clock will be synchronized and the timestamps at transmission and reception will be compared. The measurements are taken in a network without traffic.

6.2 Single-hop communication

In this section the time performances of single-hop communications will be evaluated. The tree network is composed of just two nodes.

6.2.1 Data Message

A node is set to send a simple Data Message to its parent node; the following time measurements are taken:

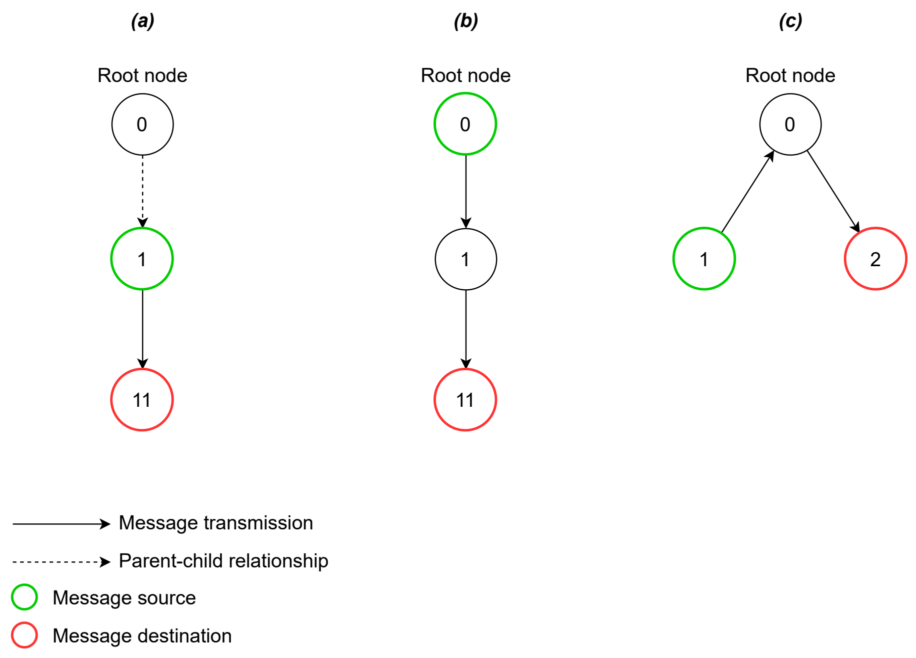

Figure 6.1: Test tree configurations: ***(a)*** single-hop communication, ***(b)*** multi-hop communication and ***(c)*** forwarding communication.

- Δ_{t1} indicates the amount of time since when the message is buffered to when it is removed from the buffer to be transmitted. This value is heavily influenced by the Advertising Duration parameter and at which moment the message is buffered;
- Δ_{t2} indicates the amount of time that passes between transmission and reception, that is the time since when a message is found in the buffer to when the parent node receives it;
- t_{tot} is the amount of time that passes since when a message is buffered to when the parent node receives it, that is $\Delta_{t1} + \Delta_{t2}$.

The time measurements can be visualized in fig. 6.2.

6.2.2 Data Request and Response

A node is set to send a Data Request message to its parent node, which will send back a Data Response message. The following time measurements are taken:

- Δ_{t1} indicates the amount of time since when the Data Request message is buffered to when it is removed from the buffer to be transmitted, as before;

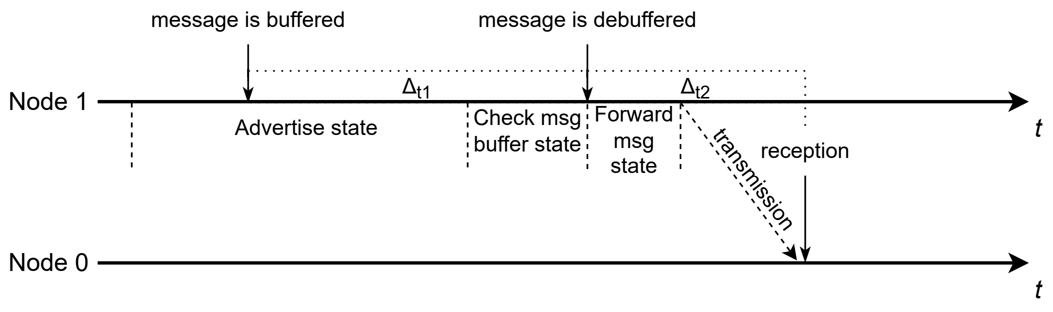

Figure 6.2: Representation of the single-hop Data Message transmission time measurements. It can be easliy seen that Δ_{t1} depends on the duration of the Advertise state and at which moment the message is buffered. The duration of the Check Message Buffer state is fixed, and the duration of the Forward Message state is upper bounded by the Forwarding Scan Duration parameter - since there is no traffic in the network, the destination node is always found on the first try.

- Δ_{t2} indicates the amount of time that passes between the transmission of the request and the reception of response, that is the time since when the Data Request message is found in the buffer of the child node to when the Data Response message is received by the same node;
- t_{tot} is the amount of time that passes since when a Data Request message is buffered to when the corresponding Data Response message is received, that is $\Delta_{t1} + \Delta_{t2}$.

The time measurements can be visualized in fig. 6.3.

6.3 Multi-hop communication

In this section the time performances of multi-hop communications will be evaluated. The tree network is composed of three nodes on three different levels.

6.3.1 Data Message

Node 11, on the second level, is set to send a simple Data Message to the the root; the message will need to pass through the node 1, on level 1. The following time measurements are taken:

- Δ_{t1} indicates the amount of time since when the Data message is buffered to when

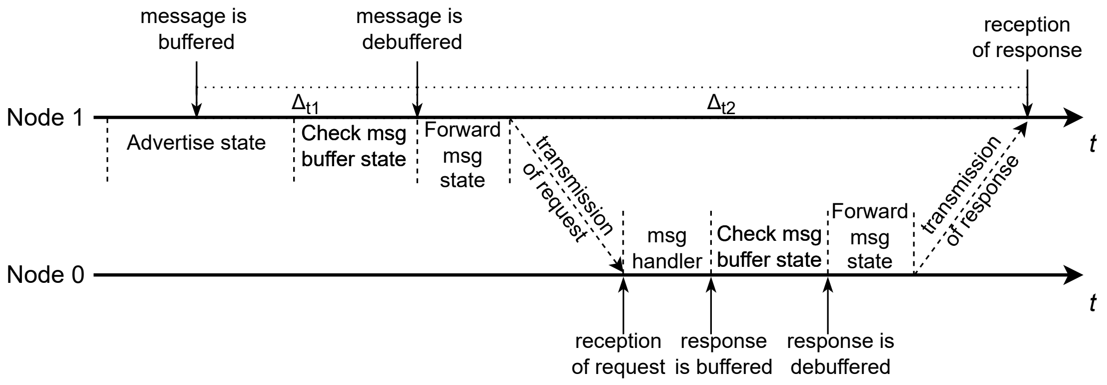

Figure 6.3: Representation of the single-hop Data Request and Response transmission time measurements.

it is removed from the buffer to be transmitted, as before;

- Δ_{t2} indicates the amount of time that passes between the transmission and reception, that is the time since when the message is found in the buffer to when it is received by the root;
- t_{tot} is the amount of time that passes since when the request message is buffered to when the root node receives it, that is $\Delta_{t1} + \Delta_{t2}$.

The time measurements can be visualized in fig. 6.4.

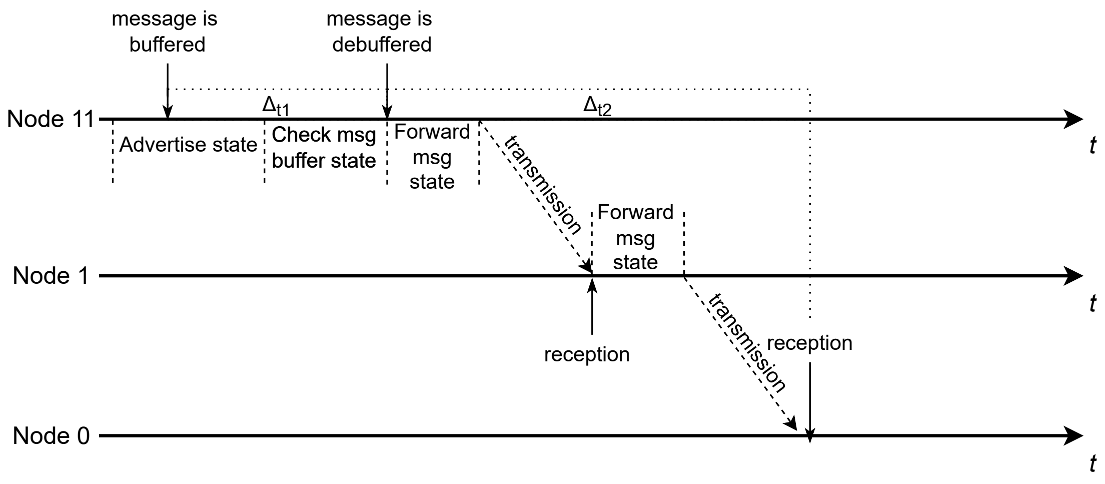

Figure 6.4: Representation of the multi-hop Data Message transmission time measurements.

6.3.2 Data Request and Response

Node 11, on the second level, is set to send a Data Request message to the the root; the message will need to pass through the node 1, on level 1. Upon reception, the root will send back a Data Response message, destined to node 11. The following time measurements are taken:

- Δ_{t1} indicates the amount of time since when the Data Request message is buffered to when it is removed from the buffer to be transmitted, as before;
- Δ_{t2} indicates the amount of time that passes between the transmission of the request and the reception of response, that is the time since when the Data Request message is found in the buffer of node 11 to when the Data Response message is received by the same node;
- t_{tot} is the amount of time that passes since when a Data Request message is buffered to when the corresponding Data Response message is received, that is $\Delta_{t1} + \Delta_{t2}$.

The time measurements can be visualized in fig. 6.5.

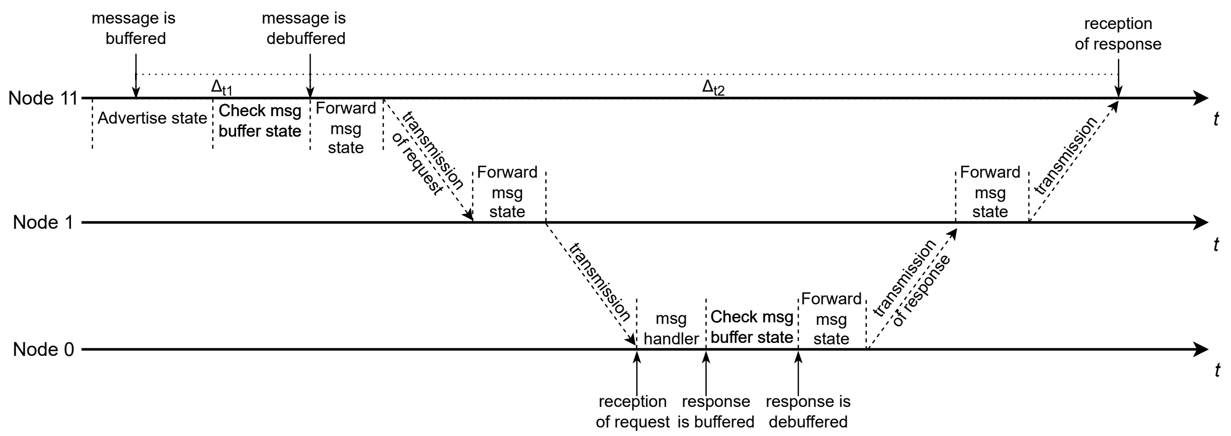

Figure 6.5: Representation of the multi-hop Data Request and Response transmission time measurements.

6.4 Forwarding communication

In this section the time performances of forwarding communications will be evaluated. The tree network is composed of three nodes on two levels: the root and two child nodes, 1 and 2.

Node 1 is set to send a Forward Data Request message to the root, with forward address 2. The root will receive the message and send a Forwarded Data Request message to node 2. Upon reception, node 2 will send a Forward Data Response message to the root, with forward address 1. The root will receive the message and send a Forwarded Data Response message to node 1. The following time measurements are taken:

- Δ_{t1} indicates the amount of time since when the Forward Data Request message is buffered to when it is removed from the buffer to be transmitted, as before;
- Δ_{t2} indicates the amount of time that passes between the transmission of the request and the reception of response, that is the time since when the Forward Data Request message is found in the buffer of node 1 to when the Forwarded Data Response message is received by the same node;
- t_{tot} is the amount of time that passes since when a Forward Data Request message is buffered to when the corresponding Forwarded Data Response message is received, that is $\Delta_{t1} + \Delta_{t2}$.

The time measurements can be visualized in fig. 6.6.

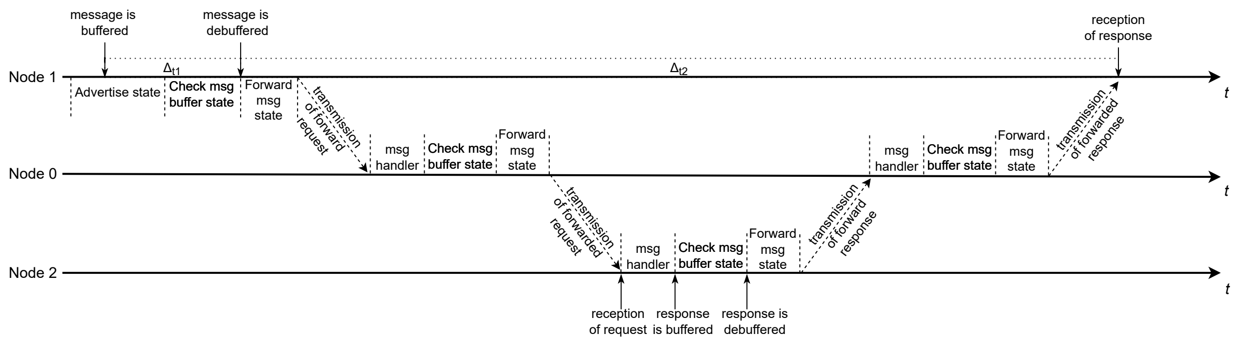

Figure 6.6: Representation of the Forward Data Request and Response transmission time measurements.

6.5 Results

The time measurements explained in the previous sections are reported in the table 6.1,table 6.2 and table 6.3 respectively.

Every messaging functionality has been tested over different number of messages: the reason is explained in section 6.6

Table 6.1: Single-hop communication time performances

	Δ_{t1} (ms)	Δ_{t2} (ms)	t_{tot} (ms)
Max	4184	1978	5758
Min	97	1225	1550
Avg	1965	1591	3556

(a) Single-hop Data Message time performances over 40 messages

	Δ_{t1} (ms)	Δ_{t2} (ms)	t_{tot} (ms)
Max	4020	3421	6881
Min	10	2421	2798
Avg	2166	2804	4970

(b) Single-hop Data Request and Response time performances over 25 messages

Table 6.2: Multi-hop communication time performances

	Δ_{t1} (ms)	Δ_{t2} (ms)	t_{tot} (ms)
Max	4033	3639	7276
Min	107	2319	2958
Avg	2856	2867	5723

(a) Multi-hop Data Message time performances over 20 messages

	Δ_{t1} (ms)	Δ_{t2} (ms)	t_{tot} (ms)
Max	4027	5408	9396
Min	3519	5120	8705
Avg	3810	5270	9079

(b) Multi-hop Data Request and Response time performances over 5 messages

Table 6.3: Forwarding communication time performances over 5 messages

	Δ_{t1} (ms)	Δ_{t2} (ms)	t_{tot} (ms)
Max	3716	6827	10 543
Min	478	5381	6072
Avg	3073	6065	9137

It can be easily noticed that Δ_{t1} is highly variable, as it depends entirely on how much time of the Advertise state is left when the message is buffered, plus the short amount of time required to inspect the message buffer and extract the first message. While the former is upper bounded by the Advertising Duration parameter - 4000 ms - the latter is

expected to grow with the number of messages sitting in the buffer. The Δ_{t1} time is the same for every type of message, as explained in the previous section.
The Δ_{t2} time instead strongly depends on the number of hops that a message take before reaching its final destination, plus a small overhead. This overhead is composed of the time needed to find the next node, which is bounded by the Forwarding Scan Duration parameter, plus the time needed to handle the message. The former is present in every message hop, while the latter is present only in advanced messaging functionalities.
For example in table 6.2b the message goes through three nodes before reaching the destination: after being sent by node 11 it goes through node 1, then node 0, then again node 1, and finally it reaches node 11. Whenever the message is received by node 1, it is immediately forwarded. Only when the message reaches the root, it require to be handled, which means generating a response, buffer it, debuffer it and send it.
In table 6.3 the message still goes through three nodes, but the handling process is done every time, which causes a little more overhead, and it is why the Δ_{t2} times are slightly larger in the latter case.
This difference can be visually seen in figures fig. 6.5 and fig. 6.6.
While the time performances measured are no match for a system employing the official Bluetooth Mesh implementation [100], the implementation proposed is still suitable for a WSN where the nodes produce non-time-sensitive data at large time intervals.
The latencies can be surely improved by keeping the connections alive during the entire network operations, therefore greatly reducing the Δ_{t1} times, as well as the time needed to find the next hop. However this possibility is not feasible at the current state of the Toit platform.

6.6 Issues

All the messaging functionalities presented were tested, and work as intended. However, the way the Toit VM manages the memory and the BLE resources of the device causes the application to crash frequently, especially every time a message is received. These crashes happens at different times during the run of the application: sometimes the application crashes at the beginning, other times it is able to run for tens of minutes before crashing.
This problem become more frequent with the number of hops that a message goes through, and it is the reason behind the small number of test messages for multi-hop and forwarding communication measurements.

The downside of the Toit platform is that the memory management is handled by the Toit VM and its garbage collector, with little to no freedom left to the developer.
The reasons behind the crashes can also be blamed on the fact that the implementation of BLE functionalities in Toit is at an infant state, barely up to the 4.0 Bluetooth specifications; the same functionalities changed during the development of this project.

7 Conclusion

In the previous chapter it was proposed a communication solution to achieve wireless data exchange between devices without involving cloud connectivity, in a wireless sensor network scenario. The solution proposed is based on a tree structure and a simple addressing pattern that allows for a quick routing algorithm without the need for routing tables. The tree exploits BLE advertising channels for the routing process, while the actual data is exchanged over data channel after a connection is established. The connections between devices are established and kept up only in the event of data exchange.
The solution proposed was developed for running on Toit-provisioned ESP32 at the current state of the BLE functionalities implemented in the Toit platform.
The solution was tested in terms of functionalities and time performances. While all the functionalities proposed work as intended, the application often crashes unpredictably due to the memory and BLE resource management of the Toit VM.
Future updates of the Toit platform BLE implementation will allow to improve the proposed solution by allowing the nodes to keep the connections active for the whole operation of the network, decreasing the latency between message transmission and reception.
Despite the memory leakage issues in this particular application, the Toit platform is an innovative solution when it comes to IoT development, that takes care of many thorny aspect of microcontroller programming; the Toit language is easy to learn and to use. The whole Toit framework is a highly functional alternative for IoT application that targets developers and companies that cannot or do not want to invest the time and resources into classical microcontroller development, which requires hardware and C language knowledge that are usually difficult to obtain. The team behind Toit is constantly at work to update and improve their product, and they offer constant support over their webside and Discord server.
The approach to IoT development taken by Toitware ApS will likely become more popular as the Internet of Things will continue to spread.

7 Conclusione

Nel capitolo precedente è stata proposta una soluzione di comunicazione per permettere lo scambio di dati wireless tra dispositivi senza l'utilizzo di connettività cloud, in uno scenario di tipo wireless sensor netowrk. La soluzione proposta è basata su una struttura ad albero e su un semplice schema di indirizzamento che consentono l'utilizzo di un algoritmo di routing rapido, che non necessita di routing table. La struttura ad albero sfrutta i canali BLE di advertising per la fase di routing, mentre i dati veri e propri vengono scambiati tramite i canali data, dopo aver stabilito una connessione. Le connessioni tra dispositivi vengono stabilite e mantenute solo nell'eventualità di scambio dati.
La soluzione proposta è stata sviluppata per essere utilizzata su ESP32 provviste di firmware Toit, all'attuale stato dell'implementazione delle funzionalità BLE da parte della piattaforma Toit.
La soluzione è stata testata in termini di funzionalità e performance temporali. Nonostante tutte le funzionalità proposte funzionano come previsto, l'applicazione soffre di crash frequenti e imprevedibili a causa della gestione della memoria e delle risorse BLE da parte della macchina virtuale Toit.
Futuri aggiornamenti dell'implementazione BLE nella piattaforma Toit potranno migliorare la soluzione proposta permettendo ai nodi di mantenere attive le connessioni durante tutto il funzionamento del network, diminuendo la latenza tra trasmissione e ricezione.
Nonostante i problemi di gestione della memoria in questa particolare applicazione, la piattaforma Toit rappresenta una soluzione innovativa per lo sviluppo IoT; il linguaggio Toit è facile da imparare ed utilizzare. L'intero framework Toit è una alternativa altamente funzionale per applicazioni IoT, mirata ad aziende e sviluppatori che non possono investire tempo e risorse nel classico sviluppo software per microcontrollori, che richiede profonde conoscenze dell'hardware e del linguaggio C, tipicamente difficili da ottenere. Il team dietro Toit è costantemente al lavoro per aggiornare e migliorare il loro prodotto, e offrono continuo supporto tramite il loro sito web e il loro server Discord.
L'approccio alternativo allo sviluppo IoT intrapreso da Toitware ApS diventerà probabilmente molto popolare in futuro, con la incrementale diffusione dell'Internet delle Cose.

Bibliography

[1] Mark Weiser. “The Computer for the 21st Century”. In: *Scientific American* (Sept. 1991).

[2] Kevin Kelly. “The next 5000 days of the web”. In: *EG Conference*. Monterey, CA, 2007.

[3] Abiy Biru Chebudie, Roberto Minerva, and Domenico Rotondi. “Towards a definition of the Internet of Things (IoT)”. PhD thesis. Aug. 2014.

[4] Alexander S. Gillis. *What is the internet of things (IoT)?* Tech Target. 2022. URL: https://www.techtarget.com/iotagenda/definition/Internet-of-Things-IoT.

[5] Brien Posey. *IoT devices (internet of things devices)*. Ed. by Sharon Shea. Tech Target. 2022. URL: https://www.techtarget.com/iotagenda/definition/IoT-device.

[6] Bojan Jovanovic. *Internet of Things statistics for 2022 - Taking Things Apart*. DataProt. 2022. URL: https://dataprot.net/statistics/iot-statistics/.

[7] Jack Steward. *The Ultimate List of Internet of Things Statistics for 2022*. findstack. 2022. URL: https://findstack.com/internet-of-things-statistics/.

[8] Sam Smith. *‘Internet of Things’ Connected Devices to Triple by 2021, Reaching Over 46 Billion Units*. Juniper Research. Dec. 2016. URL: https://www.juniperresearch.com/press/internet-of-things-connected-devices-triple-2021.

[9] A Senthil Kumar and Easwaran Iyer. “An industrial IoT in engineering and manufacturing industries–benefits and challenges”. In: *International Journal of Mechanical and Production Engineering Research and Dvelopment (IJMPERD)* 9.2 (2019), pp. 151–160.

[10] P. Jagannadha Rao et al. “Article: Detection of Rain Fall and Wind Direction using Wireless Mobile Multi Node Energy Efficient Sensor Network”. In: *International Journal of Applied Information Systems* 3.9 (Aug. 2012). Published by Foundation of Computer Science, New York, USA, pp. 33–37.

[11] S. Jaiganesh, K. Gunaseelan, and V. Ellappan. “IOT agriculture to improve food and farming technology”. In: *2017 Conference on Emerging Devices and Smart Systems (ICEDSS)*. 2017, pp. 260–266.

[12] B. Dhanalaxmi and G. Apparao Naidu. “A survey on design and analysis of robust IoT architecture”. In: *2017 International Conference on Innovative Mechanisms for Industry Applications (ICIMIA)*. 2017, pp. 375–378.

[13] Aditya Gaur et al. “Smart City Architecture and its Applications Based on IoT”. In: *Procedia Computer Science* 52 (2015). The 6th International Conference on Ambient Systems, Networks and Technologies (ANT-2015), the 5th International Conference on Sustainable Energy Information Technology (SEIT-2015), pp. 1089–1094. URL: https://www.sciencedirect.com/science/article/pii/S1877050915009229.

[14] Sarah A. Al-Qaseemi et al. “IoT architecture challenges and issues: Lack of standardization”. In: *2016 Future Technologies Conference (FTC)*. 2016, pp. 731–738. DOI: 10.1109/FTC.2016.7821686.

[15] Mary K. Pratt. *IoT interoperability standards complicate IoT adoption*. TechTarget. Sept. 2021. URL: https://www.techtarget.com/iotagenda/tip/IoT-interoperability-standards-complicate-IoT-adoption.

[16] Andreas Fink. *IoT: Lack of standards becoming a threat*. URL: https://www.iotglobalnetwork.com/iotdir/2017/04/18/iot-lack-of-standards-becoming-a-threat-5173/.

[17] Phillip Lessner. *Will a lack of standardization slow IoT development?* embedded. May 2021. URL: https://www.embedded.com/will-a-lack-of-standardization-slow-iot-development/.

[18] Kasper Lund. *The Toit language is now open source*. Nov. 2021. URL: https://blog.toit.io/the-toit-language-is-now-open-source-14bdcb1604d9.

[19] Usama Mehboob, Qasim Zaib, and Chaudhry Usama. *Survey of IoT Communication Protocols*. Techniques, Applications, and Issues. xFlow Research Inc, 2016.

[20] Cisco white paper. *Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are*. 2015.

[21] Thomas Bittman. *The Edge Will Eat The Cloud*. Gartner. Mar. 2017. URL: https://blogs.gartner.com/thomas_bittman/2017/03/06/the-edge-will-eat-the-cloud/.

[22] JR. Fuller. *The 4 stages of an IoT architecture*. TechBeacon. May 2016. URL: https://techbeacon.com/enterprise-it/4-stages-iot-architecture.

[23] Vivek Thoutam. “An Overview On The Reference Model And Stages Of Iot Architecture”. In: *Journal of Artificial Intelligence, Machine Learning and Neural Network (JAIMLNN) ISSN: 2799-1172* 1.01 (2021), pp. 34–42.

[24] Lu Tan and Neng Wang. “Future internet: The Internet of Things”. In: *2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE)*. Vol. 5. 2010, pp. V5-376-V5–380. DOI: 10.1109/ICACTE.2010.5579543.

[25] Miao Wu et al. “Research on the architecture of Internet of Things”. In: *2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE)*. Vol. 5. 2010, pp. V5-484-V5–487. DOI: 10.1109/ICACTE.2010.5579493.

[26] Smruti R. Sarangi Pallavi Sethi. “Internet of Things: Architectures, Protocols, and Applications”. In: *Journal of Electrical and Computer Engineering* 2017 (Jan. 2017). Ed. by Rajesh Khanna, pp. 33–37.

[27] Bluetooth. *Bluetooth Technology Overview*. Accessed on May 2022. Bluetooth SIG. URL: https://www.bluetooth.com/learn-about-bluetooth/tech-overview/.

[28] Bluetooth. *Bluetooth Qualification Process Overview*. Accessed on May 2022. Bluetooth SIG. URL: https://www.bluetooth.com/develop-with-bluetooth/qualification-listing/.

[29] Ionut Arghire. *BrakTooth: New Bluetooth Vulnerabilities Could Affect Millions of Devices*. SecurityWeek. Sept. 2021. URL: https://www.securityweek.com/braktooth-new-bluetooth-vulnerabilities-could-affect-millions-devices.

[30] OASIS. *Cisco, Eclipse Foundation, Eurotech, IBM, Kaazing, Machine-To-Machine Intelligence (M2Mi), Red Hat, Software AG, TIBCO, and Others Partner to Standardize MQTT Protocol*. OASIS OPEN. Apr. 2013. URL: https://www.oasis-open.org/news/pr/oasis-members-to-advance-mqtt-standard-for-m2m-iot-reliable-messaging/.

[31] HiveMQ. *HiveMQ‘s Reliable IoT Communication Enables Real-time Monitoring of Matternet’s Autonomous Drones*. Accessed on May 2022. URL: https://www.hivemq.com/case-studies/matternet/.

[32] Hema. *MQTT Implementation on Celikler Holding’s Power Plant Monitoring*. Bevywise. Sept. 2020. URL: https://www.bevywise.com/blog/iot-success-stories-mqtt-broker-celikler-holding/.

[33] EMQ. *EMQ helps IoT innovation in the petrochemical industry*. Accessed on May 2022. URL: https://www.emqx.com/en/customers/emq-helps-innovation-in-the-oil-iot.

[34] HiveMQ. *CASO Design creates Smart Kitchen Appliances with HiveMQ*. Accessed on May 2022. URL: https://www.hivemq.com/case-studies/caso/.

[35] Paul Duffy. *Beyond MQTT: A Cisco View on IoT Protocols*. Cisco. Apr. 2013. URL: https://blogs.cisco.com/digital/beyond-mqtt-a-cisco-view-on-iot-protocols.

[36] Michael Yuan. *Getting to know MQTT*. IBM Developer. May 2017. URL: https://developer.ibm.com/articles/iot-mqtt-why-good-for-iot/.

[37] OASIS Standard. *MQTT Version 5.0*. Tech. rep. Mar. 2019. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html.

[38] Amelia Dalton. *Is Exactly-Once Delivery Possible with MQTT?* May 2015. URL: https://www.eejournal.com/2015/05/28/is-exactly-once-delivery-possible-with-mqtt/.

[39] *What is Zigbee Technology? Architecture, Topologies and Applications*. Electronics Hub. Nov. 2017.

[40] OpenSystems Media. *zigbee 3.0 compliant platforms support interoperability*. Feb. 2017.

[41] Ravi Kishore Kodali, Govinda Swamy, and Boppana Lakshmi. "An implementation of IoT for healthcare". In: *2015 IEEE Recent Advances in Intelligent Computational Systems (RAICS)*. IEEE. 2015, pp. 411–416.

[42] Musewerx. *Z-Wave Wireless Control: Technology, System and Applications*. Tech. rep.

[43] Daniel Anglin Seitz. *What Is Z-Wave?* Lifewire. Jan. 2020. URL: https://www.lifewire.com/what-is-z-wave-4588924.

[44] Rajiv. *Applications of Z-wave technology*. RF Page. Mar. 2018. URL: https://www.rfpage.com/applications-of-z-wave-technology/.

[45] Brandon Lewis. *Z-Wave opens up as smart home connectivity battle closes in*. Embedded Computing Design. Sept. 2016. URL: https://embeddedcomputing.com/application/consumer/smart-home-tech/z-wave-opens-up-as-smart-home-connectivity-battle-closes-in.

[46] Thomas Brewster. "A Basic Z-Wave Hack Exposes Up To 100 Million Smart Home Devices". In: *Forbes* (May 2018).

[47] Marco Centenaro et al. "Long-Range Communications in Unlicensed Bands: the Rising Stars in the IoT and Smart City Scenarios". In: *IEEE Wireless Communications* 23 (Oct. 2015). DOI: 10.1109/MWC.2016.7721743.

[48] *Certifying LoRaWAN® Products*. LoRa Alliance. URL: https://lora-alliance.org/lorawan-certification/.

[49] *Cities Vertical Market*. LoRa Alliance.

[50] Ramon Sanchez-Iborra et al. “Performance Evaluation of LoRa Considering Scenario Conditions”. In: *Sensors (Basel)* 18 (Mar. 2018).

[51] Ferran Adelantado et al. “Understanding the Limits of LoRaWAN”. In: *IEEE Communications Magazine* 55.9 (2017), pp. 34–40. DOI: 10.1109/MCOM.2017.1600613.

[52] Martijn Saelens et al. “Impact of EU duty cycle and transmission power limitations for sub-GHz LPWAN SRDs: an overview and future challenges”. In: *EURASIP Journal on Wireless Communications and Networking* 219 (Sept. 2019). DOI: https://doi.org/10.1186/s13638-019-1502-5.

[53] *LoRaWAN® 1.0.4 Specification Package*. LoRa Alliance, Oct. 2020. URL: https://lora-alliance.org/resource_hub/lorawan-104-specification-package/.

[54] Tomas Hegr and Radim Kalfus. “Ultra Narrow Band Radio Technology in High-Density Built-Up Areas”. In: Oct. 2016. ISBN: 978-3-319-46253-0. DOI: 10.1007/978-3-319-46254-7_54.

[55] Khaldoun Agha, Guy Pujolle, and Tara Ali Yahiya. *Mobile and Wireless Networks*. Aug. 2016. DOI: 10.1002/9781119007548.

[56] Antoine Mège. *GreenCityZen and Sigfox France connect the city of Marseille’s storm drains*. Press release. Nov. 2021.

[57] Sigfox. *Sigfox Technical Overview*. Tech. rep. Jan. 2018.

[58] Jeremy Landt. “Shrouds of Time: The history of RFID”. In: (Oct. 2001).

[59] Gavin Phillips. *How Does RFID Technology Work?* MUO. May 2017. URL: https://www.makeuseof.com/tag/technology-explained-how-do-rfid-tags-work/.

[60] R. Want. “An introduction to RFID technology”. In: *IEEE Pervasive Computing* 5.1 (2006), pp. 25–33. DOI: 10.1109/MPRV.2006.2.

[61] *How much does an RFID tag cost today?* Accessed on May 2022. RFID Journal. URL: https://blog.acdist.com/understanding-rfid-and-rfid-operating-ranges.

[62] Sarah Gingichashvili. *Hitachi Develops World’s Smallest RFID Chip*. The Future Of Things. Oct. 2007. URL: https://web.archive.org/web/20090416235559/http://thefutureofthings.com/news/1032/hitachi-develops-worlds-smallest-rfid-chip.html.

[63] Advanced Controls & Distribution. *Understanding RFID and RFID Operating Ranges*. Mar. 2017. URL: https://blog.acdist.com/understanding-rfid-and-rfid-operating-ranges.

[64] James Thrasher. *How is RFID Used in the Real World*. Aug. 2013. URL: https://www.atlasrfidstore.com/rfid-insider/what-is-rfid-used-for-in-applications/.

[65] Markus Hansen and Sebastian Meissner. "Identification and Tracking of Individuals and Social Networks using the Electronic Product Code on RFID Tags". In: *The Future of Identity in the Information Society*. Ed. by Simone Fischer-Hübner et al. Boston, MA: Springer US, 2008, pp. 143–150. ISBN: 978-0-387-79026-8.

[66] Lothar Fritsch. "Business risks from naive use of RFID in tracking, tracing and logistics". In: *5th european Workshop on RFID Systems and Technologies*. 2009, pp. 1–7.

[67] Gerhard Hancke. "Practical Eavesdropping and Skimming Attacks on High-Frequency RFID Tokens". In: *Journal of Computer Security* 19 (Mar. 2011), pp. 259–288. DOI: 10.3233/JCS-2010-0407.

[68] Raghu Das, Dr Yu-Han Chang, and Dr Matthew Dyson. *RFID Forecasts, Players and Opportunities 2022-2032*. IDTechEX, Nov. 2021.

[69] Cameron Faulkner. *What is NFC? Everything you need to know*. Tech Radar. May 2017. URL: https://www.techradar.com/news/what-is-nfc.

[70] Sony Corporation and Philips. *Philips and Sony announce strategic cooperation to define next generation Near Field Radio-Frequency Communications*. Press release. Sept. 2002. URL: https://www.sony.com/en/SonyInfo/News/Press_Archive/200209/02-0905E/.

[71] Ekta Desai and Mary Grace Shajan. "A Review on the Operating Modes of Near Field Communication". In: *International Journal of Engineering and Advanced Technology (IJEAT)*. Vol. 2. Dec. 2012.

[72] NFC Forum. *NFC as Technology Enabler*. 2013. URL: https://web.archive.org/web/20131127230007/http://www.nfc-forum.org/aboutnfc/tech_enabler/.

[73] Arwa Alrawais. "Security Issues in Near Field Communications (NFC)". In: *International Journal of Advanced Computer Science and Applications* 11.11 (2020).

[74] Carles Gomez, Joaquim Oller, and Josep Paradells. "Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology". In: *Sensors* 12.9 (2012), pp. 11734–11753. ISSN: 1424-8220. DOI: 10.3390/s120911734. URL: https://www.mdpi.com/1424-8220/12/9/11734.

[75] Martin Woolley. *The Bluetooth Low Energy Primer*. Bluetooth SIG. May 2022.

[76] Mesh Working Group. *Bluetooth Core Specification 4.0*. Bluetooth SIG. June 2010.

[77] Mesh Working Group. *Bluetooth Core Specification 4.1*. Bluetooth SIG. Dec. 2013.

[78] Mesh Working Group. *Bluetooth Core Specification 4.2*. Bluetooth SIG. Dec. 2014.

[79] Mesh Working Group. *Bluetooth Core Specification 5.0*. Bluetooth SIG. Dec. 2016.

[80] Mesh Working Group. *Bluetooth Mesh Profile specification*. Bluetooth SIG. July 2017.

[81] Seyed Mahdi Darroudi and Carles Gomez. "Bluetooth Low Energy Mesh Networks: A Survey". In: *Sensors* 17.7 (2017). ISSN: 1424-8220. DOI: 10.3390/s17071467. URL: https://www.mdpi.com/1424-8220/17/7/1467.

[82] Martin Woolley. *Bluetooth Mesh Networking: An Introduction for Developers*. Bluetooth SIG. Dec. 2020.

[83] Martin Woolley. *Bluetooth Mesh Models: Technical Overview*. Bluetooth SIG. Mar. 2019.

[84] Mesh Working Group. *Bluetooth Mesh Model specification*. Bluetooth SIG. Dec. 2016.

[85] *Bluetooth Mesh Glossary of Terms*. Bluetooth SIG. URL: https://www.bluetooth.com/learn-about-bluetooth/recent-enhancements/mesh/mesh-glossary/.

[86] Kasper Lund. *Leaving Google for a couple of devices*. Feb. 2019. URL: https://blog.toit.io/building-for-billions-bcb48814d864.

[87] *Toit language basics*. Toit. URL: https://docs.toit.io/language.

[88] *Toit platform overview*. Toit. URL: https://docs.toit.io/platform/concepts.

[89] *Cloud Fleet orchestration*. Toit. URL: https://toit.io/product/cloud-orchestration.

[90] Toit. *OTA updates that let you sleep at night*. Sept. 2021. URL: https://blog.toit.io/otas-that-let-you-sleep-at-night-614fb8bedff7.

[91] Espressif Systems. *ESP32 Overview*. URL: https://www.espressif.com/en/products/socs/esp32?1&7&0.

[92] *ESP32 Technical Reference Manual*. Espressif Systems. Nov. 2021.

[93] *ESP32 Datasheet*. Espressif Systems. Mar. 2022.

[94] Céline Materna and Nils Westerlund. *Consibio chooses Toit to optimize bioprocesses*. Mar. 2021. URL: https://blog.toit.io/consibio-chooses-toit-to-optimize-bioprocesses-794f6bda6d3d.

[95] Toit. *Toit lets Trifork build an IoT product without firmware developers*. June 2021. URL: https://blog.toit.io/toit-lets-trifork-build-an-iot-product-with-no-need-for-firmware-developers-f1e032eb06a2.

[96] *ESP32 Is Now Bluetooth LE 5.0-Certified*. Espressif Systems. Dec. 2019. URL: https://www.espressif.com/en/news/BLE_5.0_Certification.

[97] *ESP-BLE-MESH Is Now Fully Certified by Bluetooth-SIG*. Espressif Systems. Oct. 2019. URL: https://www.espressif.com/en/news/ESP_BLE_MESH_SIG_Certified.

[98] Bishnu Kumar Maharjan, Ulf Witkowski, and Reza Zandian. “Tree network based on Bluetooth 4.0 for wireless sensor network applications”. In: *2014 6th European Embedded Design in Education and Research Conference (EDERC)*. 2014, pp. 172–176. DOI: 10.1109/EDERC.2014.6924382.

[99] Mohammad Afaneh. *How do I choose a UUID for my custom services and characteristics?* Oct. 2016. URL: https://www.novelbits.io/uuid-for-custom-services-and-characteristics/.

[100] Adnan Aijaz et al. “Demystifying the Performance of Bluetooth Mesh: Experimental Evaluation and Optimization”. In: *2021 Wireless Days (WD)*. IEEE. 2021, pp. 1–6.

A Message OP Codes

OP codes are used to define advanced messaging functionalities between nodes. This thesis propose some basic message types and their receiving behaviours that can be expanded to satisfy the requirement of the deployment.

Data Message

Consist of a simple message containing data. It is sent independently and does not require a response from the receiver

Table A.1: Data Message structure

Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
destination address		source address		0x00	message data		

Data Request

A Data Request message is sent to a node in order to request a Data Response message. The data payload field is left reserved: depending on the application, it can be used to specify the type of data requested.
The message must be sent only to direct descendants and ancestors. To send a data request to other nodes the forwarding functionality must be used.

Table A.2: Data Request message structure

Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
destination address		source address		0x01	reserved		

Data Response

A Data Response message is sent when a Data Request message is received. The destination address is the source address of the Data Request. The data payload contains the data requested.

Table A.3: Data Response message structure

Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
destination address		source address		0x02	requested data		

Forward Data Request

A Forward Data Request message is sent to request data from a node that is not a descendant or an ancestor, therefore not reachable with a standard Data Request message. The Forward Data Request message is sent to the root node, which will forward the request to the recipient node, by sending a Forwarded Data Request message. The data payload contains the requestee address, that is the address of the final recipient node. The Byte 0 is left reserved: depending on the application, it can be used to specify the type of data requested.

Table A.4: Forward Data Request message structure

Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
0x0000 (root address)		requester address		0x03	requestee address		reserved

Forwarded Data Request

When the root node receives a Forward Data Request message, it sends a Forwarded Data Request to the requestee address. The requester address of the Forward Data Request message is contained in the data payload field. The Byte 0 is the same Byte 0 of the corresponding Forward Data Request message.

Table A.5: Forwarded Data Request message structure

Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
requestee address		0x0000 (root address)		0x04	requester address		reserved

Forward Data Response

When a node receive a Forwarded Data Request message, it sends as a response a Forward Data Response message. This message is sent to the root node, which will forward it to the requester. The requester address is contained in the data payload. The Byte 0 contains the requested data.

Table A.6: Forward Data Response message structure

Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
0x0000 (root address)		requestee address		0x05	requester address		requested data

Forwarded Data Request

When the root node receives a Forward Data Response it sends a Forwarded Data Response to the requester node. The requester address contained in the Forward Data Request is not the destination address. The requestee address is the address of the node that sent the Forwarded Data Request message. The Byte 0 contains the requested data.

Table A.7: Forwarded Data Response message structure

Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
requester address		0x0000 (root address)		0x06	requestee address		requested data

Provisioning Request

The Provisioning Request message is used to trigger the Address Provisioning state in order to expand the network. When a node receive a Provisioning Request message it transition to the Address Provisioning state. Then it sends a Provisioning Request message to all its children nodes. The last three bytes are reserved.

Table A.8: Provisioning Request message structure

Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
destination address		source address		0x07	reserved		

B Tree Message

The TreeMessage class defines a message object, which is exchanged by node during the operation of the network. As seen before, a message is an 8-byte data structure containing the source and destination addresses, the OP code and the message data.
In a TreeMessage object these four elements are stored as four separate ByteArrays - in Toit, a ByteArray is a fixed-length list for 8-bit unsigned integers.
The class provides two constructor to initialize a TreeMessage object, either by providing a 8-byte ByteArray in the format of a message, or by providing the four distinct field as integer arguments.
The class provides dynamic getter methods to obtain source and destination addresses, OP code and message data, as well as convenience static methods for conversion between ByteArray and elements and vice versa, while taking care of endianess.
Before the class the OP codes are defined as constants

Listing B.1: TreeMessage class and constant definitions

```
/**
Messages OPCODE
*/
DATA_MESSAGE                ::= 0x00
DATA_REQUEST                ::= 0x01
DATA_RESPONSE               ::= 0x02
FORWARD_DATA_REQUEST        ::= 0x03
FORWARDED_DATA_REQUEST      ::= 0x04
FORWARD_DATA_RESPONSE       ::= 0x05
FORWARDED_DATA_RESPONSE     ::= 0x06
PROVISIONING_REQUEST        ::= 0x07

NOT_A_TREE_MESSAGE_EXCEPTION   ::= "The ByteArray is not in the
    TreeMessage format!"

/**
A message exchanged in the tree network.
*/
class TreeMessage:
```

```
srcAddr /ByteArray :=?
dstAddr /ByteArray :=?
msg_data /ByteArray :=?
opcode /ByteArray :=?

/**
Build a message by receiving destination address, source address,
    opcode and message data.
*/
constructor dst/int src/int opc/int data/int :
  msg_data = data_to_bytearray data
  dstAddr = addr_to_bytearray dst
  srcAddr = addr_to_bytearray src
  opcode  = opcode_to_bytearray opc

/**
Build a message from the 8-byte bytearray.
*/
constructor msg/ByteArray :
  if msg.size != 8 :
    throw NOT_A_TREE_MESSAGE_EXCEPTION
  else:
    dstAddr = msg[..2]
    srcAddr = msg[2..4]
    opcode = msg[4..5]
    msg_data = msg[5..]


/**
Takes an int message data and return the corresponding 3-byte
    bytearray.
Little endianess is used.
The data must be 3-byte long, that is data <= 0xFFFFFF.
*/
static data_to_bytearray data/int -> ByteArray:
  byte_2 := (data & BYTE_2_MASK) >> 16
  byte_1 := (data & BYTE_1_MASK) >> 8
  byte_0 := (data & BYTE_0_MASK)

  return #[byte_0, byte_1, byte_2]

/**
Takes an int address and return the corresponding 2-byte bytearray.
Little endianess is used.
addr must be 2-byte long, that is addr <= 0xFFFF.
*/
static addr_to_bytearray addr/int -> ByteArray:
  byte_1 := (addr & BYTE_1_MASK) >> 8
  byte_0 := (addr & BYTE_0_MASK)

  return #[byte_0, byte_1]

/**
Takes an int opcode and return the corresponding 1-byte bytearray.
opc must be 1-byte long, that is addr <= 0xFF.
*/
static opcode_to_bytearray opc/int -> ByteArray:
  byte_0 := opc & BYTE_0_MASK

  return #[byte_0]


/**
Takes a 4-byte advertising data and returns the corresponding int
    address.
*/
static adv_data_to_address adv_data/ByteArray -> int:
  return adv_data[2] + (adv_data[3]<<8)

/**
Takes a 2-byte bytearray and returns the corresponding int address.
*/
static bytearray_to_address array/ByteArray -> int:
  return array[0] + (array[1]<<8)

/**
```

```
Returns the message 8-byte bytearray.
*/
get_message_bytearray -> ByteArray:
  return #[dstAddr[0],dstAddr[1],srcAddr[0],srcAddr[1],opcode[0],
      msg_data[0],msg_data[1],msg_data[2]]

/**
Returns the source address of the message.
*/
get_src_addr -> int:
  return srcAddr[0] + (srcAddr[1]<<8)

/**
Returns the destination address of the message.
*/
get_dst_addr -> int:
  return dstAddr[0] + (dstAddr[1]<<8)

/**
Returns the opcode of the message.
*/
get_opcode -> int:
  return opcode[0]

/**
Returns the message data of the message.
*/
get_msg_data -> int:
  return msg_data[0] + (msg_data[1]<<8) + (msg_data[2]<<16)
```

C Tree Node

The TreeNode class defines a node which takes part in a Tree network. A TreeNode object is initialized with a boolean value, which indicates whether that node is the root. The message buffer, the current and next state of the node, the number of children and the current message that is being handled are stored as private fields.

A TreeNode object start operating with the method StartNetwork, which can only be called once per node. This method launches a task - a Toit task is a block of code with an independent control flow that takes turn to run - containing a while loop. In this while loop, the 5-state machine is implemented with separate functions, each function returns an integer indicating which function - that is, which state - should be executed at the next iteration.

The class also offers methods to obtain the address, the children and the current state of the node.

A message is sent by inserting it in the message buffer, which will be checked in the next Check Message Buffer state.

The class also contains a message handler function, which defines the behaviours upon message reception, depending on the message OP code.

Before the class, the characteristics UUIDs, the states indicator, the tuning parameters, the exceptions and other constants are defined.

Listing C.1: TreeNode class and constant definitions

```
TREE_MESSAGE_SERVICE    ::= ble.uuid 0x8AEC
OWN_ADDRESS             ::= ble.uuid 0x3082  // 2-byte
NEW_MSG                 ::= ble.uuid 0x936A  // 1-byte
TREE_MSG                ::= ble.uuid 0xE51C  // 8-byte

STATE_INIT                      ::= 0x01
STATE_ADDRESS_PROVISIONING      ::= 0x02
STATE_ADVERTISE                 ::= 0x03
STATE_CHECK_MSG_BUFFER          ::= 0x04
STATE_FORWARD_MESSAGE           ::= 0x05

NEW_ADDRESS_RECEIVED            ::= 0x10
```

```
NEW_MESSAGE_RECEIVED          ::= 0x11

PROVISIONING_SCAN_DURATION   ::= Duration --ms = 0_500
ADVERTISING_DURATION          ::= Duration --s = 4
FORWARDING_SCAN_DURATION     ::= Duration --ms = 0_700

ROOT_ADDR                    ::= 0x0000
UNPROVISIONED_ADDRESS     ::= 0xFFFF

NEXT_HOP_NOT_FOUND_EXCEPTION          ::= "Next hop not found!"
UNREACHABLE_ADDRESS_EXCEPTION         ::= "Destination address is
    unreachable"
NETWORK_ALREADY_STARTED_EXCEPTION    ::= "This node already started the
     network operation!"

KEEP_LOOKING_PROBABILITY/int  ::= 50

/**
A node in a tree network.
*/
class TreeNode:
  mgs_buffer_ /List := []
  i_ /int := 0
  is_root_ /bool

  own_addr_ /int? := null
  curr_msg_ /TreeMessage? := null
  already_started_ /bool := false

  next_state_ /int := ?
  curr_state_ /int := STATE_INIT

  /**
  Build a node by specifying if it is the root of the network.
  Only one root node is allowed.
  */
  constructor .is_root_ :

```

```
  if is_root_:
    next_state_ = STATE_ADDRESS_PROVISIONING
    own_addr_ = ROOT_ADDR
  else:
    next_state_ = STATE_ADVERTISE
    own_addr_ = UNPROVISIONED_ADDRESS

/**
If root, kickstarts the network by looking for neighbors.
Else, advertise it's presence in the network.
*/
startNetwork :
  if already_started_:
    print "The network is already started!"
    return

  already_started_ = true

  config := ble.ServerConfiguration
  treeMsgService := config.add_service TREE_MESSAGE_SERVICE
  ownAddr := treeMsgService.add_read_write_characteristic
      OWN_ADDRESS
  newMsg := treeMsgService.add_read_write_characteristic NEW_MSG
  msg := treeMsgService.add_read_write_characteristic TREE_MSG

  task::
    while true:
      if next_state_ == STATE_ADDRESS_PROVISIONING:
          next_state_ = address_provisioning
      else if next_state_ == STATE_ADVERTISE:
          next_state_ = advertise config ownAddr newMsg msg
      else if next_state_ == STATE_CHECK_MSG_BUFFER:
          next_state_ = check_msg_buffer
      else if next_state_ == STATE_FORWARD_MESSAGE:
          next_state_ = forward_message

  return

```

```

/**
Scan for neighboring unprovisioned devices.
If found any, it changes their ownAddr characteristic.
*/
address_provisioning ->int:

  curr_state_ = STATE_ADDRESS_PROVISIONING
  print "Start Address Provisioning state"

  device := ble.Device.default
  devices := List

  device.scan --duration=PROVISIONING_SCAN_DURATION: | remote_device/ble.RemoteDevice |
    if remote_device.data.service_classes.contains TREE_MESSAGE_SERVICE:
      addr := TreeMessage.adv_data_to_address (remote_device.data.manufacturer_data)
      if addr == UNPROVISIONED_ADDRESS and (not devices.contains remote_device.address):
        devices.add(remote_device.address)
        print "Found $remote_device.address"

  print "Scan terminated!"
  if devices.is_empty:
    print  "No devices found!"
    device.close
    return own_addr_ == ROOT_ADDR ? STATE_ADDRESS_PROVISIONING : STATE_ADVERTISE

  clients := List
  devices.do: | address |
    e := catch:
      client := device.connect address
      clients.add client
      print "Connected to $address"
    if e:
        continue.do

  clients.do: | client |
    service := client.read_service TREE_MESSAGE_SERVICE
    addr_char := service.read_characteristic OWN_ADDRESS
    new_msg_char := service.read_characteristic NEW_MSG
    i_ = i_ + 1
    next_addr := (own_addr_*10) + i_
    print "Provisioned $client.address with address $(%x next_addr)"
    addr_char.write_value (TreeMessage.addr_to_bytearray next_addr)
    new_msg_char.write_value #[NEW_ADDRESS_RECEIVED]

  device.close
  return STATE_ADVERTISE

/**
Advertise it's address, waiting for a client connection.
*/
advertise config/ble.ServerConfiguration ownAddr/ble.
    ReadWriteCharacteristic newMsg/ble.ReadWriteCharacteristic msg/
    ble.ReadWriteCharacteristic:

  curr_state_ = STATE_ADVERTISE
  print "Start Advertising state with address $own_addr_"

  device/ble.Device? := null
  try:
    device = ble.Device.default config
    advertiser := device.advertise
    data := ble.AdvertisementData
      --name="toit_device"
      --service_classes=[TREE_MESSAGE_SERVICE]
      --manufacturer_data=#[0xFF, 0xFF, (TreeMessage.
          addr_to_bytearray own_addr_)[0], (TreeMessage.
          addr_to_bytearray own_addr_)[1]]

    advertiser.set_data data
    advertiser.start --connection_mode=ble.
```

```
          BLE_CONNECT_MODE_UNDIRECTIONAL

      received/int? := null

      timeout := catch:
        with_timeout (ADVERTISING_DURATION): device.
            wait_for_client_connected

      if timeout:
        print "Advertising timeout reached"
        advertiser.close
        device.close
        return own_addr_ == UNPROVISIONED_ADDRESS ? STATE_ADVERTISE :
            STATE_CHECK_MSG_BUFFER
      else:
        print "Connected as slave"
        received = newMsg.value[0]
        device.wait_for_client_disconnected
        if received == NEW_MESSAGE_RECEIVED:
          curr_msg_ = TreeMessage msg.value
          advertiser.close
          device.close

          if curr_msg_.get_dst_addr == own_addr_:
            time := (Time.now.plus --h=2).local
            print "Message received at time $(%02d time.h):$(%02d time
                .m):$(%02d time.s):$(%03d time.ns/1000000)"
            return message_handler curr_msg_

          else:
            return STATE_FORWARD_MESSAGE

        else if received == NEW_ADDRESS_RECEIVED:
          own_addr_ = TreeMessage.bytearray_to_address ownAddr.value
          print "Provisioned with address $own_addr_"
          advertiser.close
          device.close
          return STATE_ADDRESS_PROVISIONING

      advertiser.close
      device.close
      return STATE_ADVERTISE
    finally:
      if device : device.close

  /**
  Check message buffer if there are messages waiting to be sent.
  */
  check_msg_buffer ->int:
    curr_state_ = STATE_CHECK_MSG_BUFFER
    print "Start Check Message Buffer state"


    if mgs_buffer_.is_empty:
      print "No messages in message buffer"
      return STATE_ADVERTISE

    else:
      print "$mgs_buffer_.size message(s) in the buffer"
      curr_msg_ = mgs_buffer_.first
      mgs_buffer_ = mgs_buffer_[1..].copy
      time := (Time.now.plus --h=2).local
      print "Message debuffered at time $(%02d time.h):$(%02d time.m):
         $(%02d time.s):$(%03d time.ns/1000000)"

      return STATE_FORWARD_MESSAGE

  /**
  Forward a message in the tree network.
  If the next node in the path is not found, it randomly chooses
      between keep looking
  or going back to advertising
  */
  forward_message ->int:

    curr_state_ = STATE_FORWARD_MESSAGE
    print "Start Forward Message state"

    next_hop := ?

    if own_addr_ > curr_msg_.get_dst_addr:
      // Going up.
      next_hop = own_addr_/10

    else:
      // Going down.
      dst_lvl :=  level curr_msg_.get_dst_addr
      next_lvl := (level own_addr_) + 1
      next_hop = curr_msg_.get_dst_addr / ((math.pow 10 (dst_lvl -
          next_lvl)).to_int)

    if not is_reachable next_hop:
      throw UNREACHABLE_ADDRESS_EXCEPTION

    device/ble.Device? := null
    next_hop_addr := null
    try:
      device = ble.Device.default
      scan_err := catch:
        next_hop_addr = find_next_hop device next_hop
      if scan_err:
        print "$scan_err"
        choice := random 1 101
        if choice <= KEEP_LOOKING_PROBABILITY:
          print "Random choice: keep looking for next hop"
          device.close
          return STATE_FORWARD_MESSAGE
        else:
          print "Random choice: go back to advertise"
          tmp := [curr_msg_]
          mgs_buffer_.do: | it |
            tmp.add it
          mgs_buffer_ = tmp.copy
          device.close
        return STATE_ADVERTISE
    else:
      client/ble.Client? := null
      e := catch:
        client = device.connect next_hop_addr
        print "connected to $client.address"
      if e:
        print "Error $e.. Trying again..."
        return STATE_FORWARD_MESSAGE

      service := client.read_service TREE_MESSAGE_SERVICE
      msg_char := service.read_characteristic TREE_MSG
      new_msg_char := service.read_characteristic NEW_MSG
      msg_char.write_value curr_msg_.get_message_bytearray
      new_msg_char.write_value #[NEW_MESSAGE_RECEIVED]
      return STATE_ADVERTISE

  finally:
    if device : device.close


find_next_hop device/ble.Device next_hop/int:
  device.scan --duration=FORWARDING_SCAN_DURATION: | remote_device/
      ble.RemoteDevice |
    if remote_device.data.service_classes.contains
        TREE_MESSAGE_SERVICE:
      addr := TreeMessage.adv_data_to_address (remote_device.data.
          manufacturer_data)
      if addr == next_hop:
        print "next hop ($next_hop) found: $remote_device.address"
        return remote_device.address

  throw NEXT_HOP_NOT_FOUND_EXCEPTION

/**
Insert a message in the message buffer.
*/
send_message msg/TreeMessage:
    mgs_buffer_.add msg
    return

  /**
  Returns the current address of the node.
  */
  get_address -> int:
    return own_addr_

  /**
  Returns the number of children of the node.
  */
  get_i ->int:
    return i_

  /**
  Returns a list of the address of the children.
  */
  get_children ->List:
    children := List
    i_.repeat: | i |
      children.add (own_addr_*10) + i

    return children

  /**
  Returns the current state of the node (Address provisioning,
      advertising, checking message buffer, forwarding message)
  */
  get_state ->int:
    return curr_state_

  /**
  Message handler function. It is called when a node receive a message
      with dstAddr == ownAddr.
  */
  message_handler msg/TreeMessage:
    opc := msg.get_opcode
    if opc == DATA_MESSAGE:
      data := msg.get_msg_data
      print "received $data from $msg.get_src_addr"

    if opc == DATA_REQUEST:
      print "Received Data Request from $msg.get_src_addr"
      //data_response := get_data
      data_response := random 0 0xFFFFFF
      response := TreeMessage msg.get_src_addr own_addr_ DATA_RESPONSE
          data_response
      send_message response
      print "Sent $data_response as response to $msg.get_src_addr"

    if opc == DATA_RESPONSE:
      data := msg.get_msg_data
      print "Received $data as response from $msg.get_src_addr"

    if opc == FORWARD_DATA_REQUEST:
      fwd_addr := msg.get_msg_data & 0x00_00_FF_FF
      response := TreeMessage fwd_addr own_addr_
          FORWARDED_DATA_REQUEST msg.get_src_addr
      if is_reachable fwd_addr:
        send_message response
        print "Forwarded DATA REQUEST from $msg.get_src_addr to
            $fwd_addr"
      else:
        throw UNREACHABLE_ADDRESS_EXCEPTION

    if opc == FORWARDED_DATA_REQUEST:
      rec_addr := msg.get_msg_data
      print "Received forward data request from $rec_addr"
      //data_response := get_data
      data := random 0 0xFF
      response := TreeMessage ROOT_ADDR own_addr_
          FORWARD_DATA_RESPONSE (rec_addr + (data<<16))
      send_message response
      print "Sent $data as forward response to $rec_addr"

    if opc == FORWARD_DATA_RESPONSE:
      fwd_addr := msg.get_msg_data & 0x00_00_FF_FF
      data := msg.get_msg_data>>16
      response := TreeMessage fwd_addr own_addr_
          FORWARDED_DATA_RESPONSE (msg.get_src_addr + (data<<16))
      if is_reachable fwd_addr:
        send_message response
        print "Forwarded DATA RESPONSE from $msg.get_src_addr to
            $fwd_addr"
      else:
        throw UNREACHABLE_ADDRESS_EXCEPTION

    if opc == FORWARDED_DATA_RESPONSE:
      rec_addr := msg.get_msg_data & 0x00_00_FF_FF
      data := msg.get_msg_data>>16
      print "Received $data as forward response from $rec_addr"

    if opc == PROVISIONING_REQUEST:
      "Starting address provisioning triggered by request from $msg.
          get_src_addr"

      children := get_children
      children.do: | c |
        tmp := TreeMessage c own_addr_ PROVISIONING_REQUEST 0
        send_message tmp
      return STATE_ADDRESS_PROVISIONING
    return STATE_CHECK_MSG_BUFFER

/**
Given a destination address, returns whether that address is
    reachable from this node.
That is, if that address is either a descendant or an ancestor of
    this node.
*/
is_reachable dst_addr/int ->bool:

  lvl_diff := ((level dst_addr) - (level own_addr_)).abs

```

```
    if own_addr_ < dst_addr:
      return own_addr_ == (dst_addr/((math.pow 10 lvl_diff).to_int))
    else:
      return dst_addr == (own_addr_/((math.pow 10 lvl_diff).to_int))


/**
Convenience method.
Returns the tree level of a node given its node address.
Correspond to the number of digits of the address, except for the root
    , which has level 0.
address must be a positive integer < 65535.
*/
level address/int -> int:
  6.repeat: | lvl |
    if address/((math.pow 10 lvl).to_int) == 0:
      return lvl

  return -1
```

Toit is a new object-oriented programming language for microcontrollers. The Toit virtual machine enables multiple independent apps to run side-by-side through software-based fault isolation. Toit is being developed as open source by the Danish company Toitware ApS, which collaborates with DTU Compute in the EU project TRANSACT. Although there are a plethora of programming solutions for IoT devices, they typically either involve low-level programming or their high-level programming requires too many resources. The objective of the thesis is to develop a communication solution for IoT devices using the Toit language. The solution proposed in this thesis is a tree-based network that allows devices to exchange data over Bluetooth Low Energy data channels without involving cloud connectivity.