ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Ingegneria e Architettura
Corso di Laurea (Magistrale) in Ingegneria e Scienze Informatiche

A platform for aggregate computing over LoRaWAN network

Tesi di laurea in
Pervasive Computing

Relatore
Prof. Mirko Viroli

Candidato
Andrea Placuzzi

Correlatore
Ing. Dott. Danilo Pianini
Prof. Danny Weyns
Dott. Roberto Casadei

III Sessione di Laurea
Anno Accademico 2018-2019

Abstract

Recent technological developments led to increased computational and networking capabilities of everyday objects. This situation resulted in an increase in number of devices embedded in cyber-physical systems. In order to simplify the design and management of pervasive and heterogeneous systems like these, there is need for new high-level paradigms able to capture concerns like heterogeneity and location of the devices. Aggregate computing is one of these: it proposes to describe the global behaviour of a system by managing global spatio-temporal data structures, and abstracting details of its physical network, as topology and communication technology. A related problem with the design of complex pervasive systems is verifying their behaviour in a real scenario, because it is generally expensive, complicated, and not always possible in practice. A partial solution to the problem is testing this kind of systems using simulations. Even though simulations execute a system model, it should be noted that such model is only a system abstraction; however they can still provide reliable insights on the system behaviour and performance. In the Internet-of-Things context, an emergent enabling communication technology for situated devices is LoRaWAN. LoRaWAN is a network protocol that allows long range communications and low energy consumption, at the cost of limited data rate. There are currently no platforms for aggregated languages that support their execution over LoRaWAN networks. Moreover nowadays there are no simulators supporting real simulation of aggregate system over LoRaWAN networks: however there are simulators supporting aggregate applications or LoRaWAN networks. The contribution of this thesis is to provide a platform that supports the LoRaWAN abstractions as backend of an aggregate computing system, and join it to the existing DingNet simulator achieving a platform allowing aggregate applications simulations over realistic LoRaWAN networks.

Contents

List of Figures

Introduction

The everyday environment we are immersed in is pervaded by devices capable of computing and communicating. Devices with these capabilities enable to think and design smart-environments, that accordingly to the domain and scale of the application can be smart-cities, smart-homes, smart-hospitals, and so on. All these systems, and more in general, the cyber-physical systems (CPSs), are composed of myriads of heterogeneous devices. The devices can differ in their computational and communication capabilities, and ability to interact with the environment. This heterogeneity makes it difficult to design open, distributed, and technology independent systems. One of the most particular components to enable interoperability is the communication technology. In fact there exist several wireless communication technologies that differ for their features and limitation, like transmission range, energy consumption, device cost, data rate, and so on. One of the most promising communication technology featuring low power consumption and long communication range is LoRaWAN[1].

In literature, several approaches exist to try to simplify the design of heterogeneous distributed situated systems. One class of approaches proposes the usage of unifying middlewares. The devices applications are supposed to leverage it in order to be able to communicate with each other. One example is Sentilo[2], a cross platform middleware designed for the smart-city of Barcelona. It provides a simple REST interface to send and receive sensor data, and a set of core modules to govern the system, like real time storage, and network security. Its architecture is extendible and allows horizontal scalability from single servers to clusters. A different take on the problem is proposed by global to local paradigms. These paradigms do not focus on programming single devices and on their communication, but they try to interpret the whole system as a single computational machine with a space-time extension and to program it. Aggregate computing [4] is one of these paradigms and proposes a devoted language. The language considers the entire set of system devices as a single computational machine; further, it abstracts from the specific network protocols used at low level.

[1] https://lora-alliance.org/sites/default/files/2018-07/lorawan1.0.3.pdf
[2] http://www.sentilo.io

The behaviour of a complex pervasive system, and of an aggregate system as well, cannot be tested on a single machine; at the same time, testing in the real world is usually inconvenient, complex, and expensive, or even straightforwardly impossible. A partial solution to test the behaviour of a system prior to deployment is via simulation. There are several simulators that aim to simulate Internet-of-Things (IoT) systems. These simulators can capture different levels of abstraction and be generic or specific for some application domains. Generally, the more you increase the flexibility of the simulator, the more you pay in performance or lose in the proximity between the real system and the executed model. One simulation tool, designed in University of Bologna, is Alchemist [14]. Alchemist has been used in the past, for instance, for simulating: crowd detection [6, 19] and evacuation [18], mixed edge-cloud computation [7], opportunistic instant messaging [8], smart vehicle counting [21], etc. Alchemist allows simulating aggregate systems, but it abstracts from the network specifications, and along this calls the realistic simulation of networking protocol and physical layers. A specialised platform for simulating the network layer and dedicated to LoRaWAN system is DingNet. This platform is useful to simulate CPSs that includes a high number of Long Range (LoRa)-based devices; considering also the high-density of devices that can be integrated in these kind of networks [11].

The purpose of this thesis is to allow the simulation of aggregate programs on a network of LoRaWAN devices. To do so two main activities will be performed:

- design and implementation of a LoRaWAN network platform for the Protelis [15] programming language;
- design and implement a connection layer between the existing DingNet simulator and the above mentioned platform for the execution of Protelis program.

Thesis Structure. Accordingly, the reminder of this thesis is structured as follows. Chapter 1 provides an overview of the aggregate computing paradigm, LoRaWAN protocol, and the DingNet simulator. Chapter 2 exposes the application layer and other features, which improves DingNet adding the support to execute applications over LoRaWAN, and the work done to allow the execution of Protelis application over the LoRaWAN network. Chapter 3 illustrates two case-studies developed in the new simulation framework. Finally, Chapter 4 concludes this thesis by summarising its main contribution and introducing future works and interesting topics to evaluate.

Chapter 1

Background

This chapter provides an overview of the main concepts and technologies on which this thesis is based. Section 1.1 introduces the Aggregate Computing paradigm; it discusses its core idea, the theory and basic mechanism under it, and finally introduces Protelis, a language rooted in this paradigm. Section 1.2 discusses the LoRaWAN network protocol, finally Section 1.3 introduces the DingNet LoRaWAN network and the relative simulator.

1.1 Aggregate Computing

Recent technological developments led to computational and networking capabilities being more and more integrated with everyday objects, such as smartphones, wearable devices, drones, smart vehicular systems, domestic appliances, and other kinds of sensors and actuators. This development led to distributed systems counting many devices differing from each other for computational capabilities and communication technologies, and producing pervasive heterogeneous and complex systems. The classic approach for distributed systems proposes a device-centric point of view. The goal is to obtain the global behaviour as emergent phenomena from the interaction of single devices. So the focus of the designer is on local structure, behaviour, and interaction. The final result is typically a system strongly dependent on communication technology that tends to be rigid and costly to testing, evolve, and maintain.

Aggregate computing is an emerging programming paradigm devoted to modern distributed systems [4], which proposes a paradigm shift. It proposes to program the entire set of devices as a single entity. The goal is to define the global behaviour in a declarative way focusing on manipulation of global spatial and temporal data structures. This allows the designer to abstract from the real device that will execute the program, its communication capability, network infrastruc-

ture, and its execution platform [20]. The final result will be a complex, modular, extendible, self-adaptive and self-organized distributed system. The aggregate computing approach offers base functionalities that are composable with each other to assemble advanced and user-friendly APIs. In order to organize all the provided functionalities, a multilayer stack has been designed, which collects them all by abstraction level. As it is possible to see from the stack in Figure 1.1, the base constructs are based on the field calculus [2] theory.

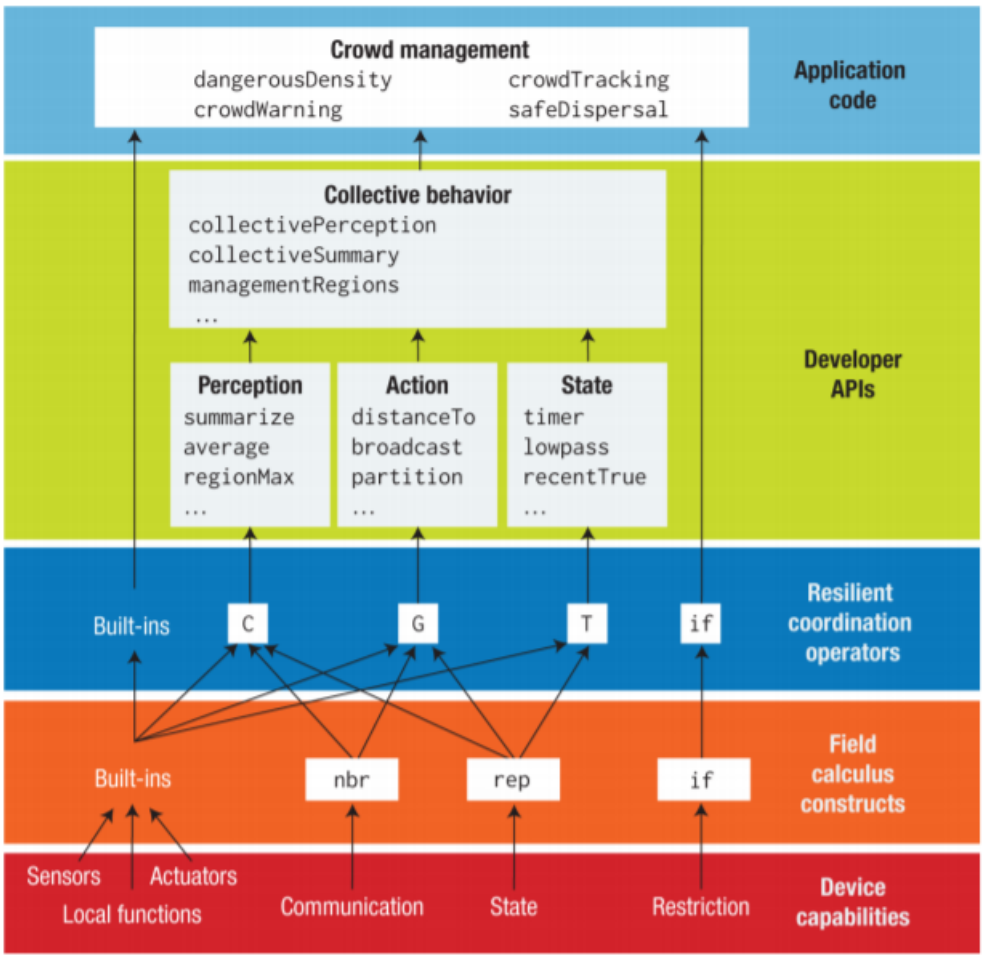


Figure 1.1: Aggregate programming stack. [4]

1.1.1 Field-calculus

The field calculus is a theoretical model that describes a set of primitives to manipulate the concept of computational field. A computational field is a distributed

data structure that maps every networked device to some local value. In field calculus, everything is a field (value, variable, expression, function) and the constructs to build and manipulate it are:

- **Functions**, $b(e1, \ldots, e_n)$ applies function b to arguments $e1, \ldots, e_n$. The output field is obtained by the point-wise evaluation of the operator to the input fields;
- **Stateful computation**, $rep(x \leftarrow v)\ \{s1; \ldots; s_n\}$ a local variable x is defined and initialized to value v. Then the value is periodically updated with the result of statements $s1; \ldots; s_n$;
- **Interaction**, $nbr(s)$ share the local value of s, represented as a field, with its neighbours. The result is a field of fields (the same information shared from neighbours) that is possible to reduce to a simple field using *hood* operators;
- **Domain restriction**, $if(e)\ \{s1; \ldots; s_n\}\ else\ \{s'_1; \ldots; s'_n\}$ split the field in two sub-field according to the evaluation of e. Where e is true is applied the statement $s1; \ldots; s_n$, while where e is false is applied the statement $s'_1; \ldots; s'_n$.

The behaviour of aggregate systems can be expressed as a functional composition of operators that manipulate (evolve, combine, restrict) computational fields [10].

1.1.2 Building blocks operators

Basic field calculus constructors are low-level operators which do not guarantee to develop self-stabilizing programs as proved in [18]. In complex system, self-stabilizing property guarantees that after each transitory phase, the system returns to a stationary state (if exists), where it is possible to predict its behaviour. [5] introduces a set of building blocks that are programs written with the low-level operators. These programs are self-stabilizing and it is possible to prove that programs written combining only these functions will be self-stabilizing. These building blocks compose the central layer of the aggregate programming stack, Figure 1.1, and the main ones are:

- $G(source, init, metric, accumulate)$: function to spread information across space. First, build a field of the shortest path that starts from the *source* with the chosen *metric*. Then spread the information across the field starting from the *init* value and modifying it at each step with the *accumulate* function;
- $C(potential, reduce, local, null)$: function to accumulates *local* value along the *potential* field until the source. The final value is obtained applying the

reduce function to the *local* value of the crossed nodes. If there is nothing to accumulate default value *null* is returned;

- $T(init, zero, decay)$: function to evolve the state for a specified period. The period starts from *init* and decreases to *zero* with a *decay* rate;
- $S(grain, metric)$: function to split the network into partitions of nodes. This function leverages the *metric* function to define partitions of a size proportional to *grain* and elects a leader for each one.

This set of four functions with the domain restriction operator is able to provide most of the commonly used coordination patterns for distributed systems.

1.1.3 Protelis

Protelis [15] is a language rooted on the aggregate computing paradigm, sharing its semantic with field calculus and inspired on Proto [3]. It provides an implementation of the entire aggregate computing stack, and supports for higher-order field calculus [2], that introduces functions as first-class citizens with a lot of benefits. Protelis adopts a C- or Java-like syntax to reduce the learning curve, but it is a purely functional language. Protelis aims to be a language that allows to use the principal of aggregate computing in a practical way. It tries as much as possible to be interoperable leaning on the existing Java platform. This allows to avoid to rewrite aggregate computing libraries for different platforms, and to use the plethora of Java libraries. According to the aggregate programming languages principles a Protelis program abstracts from the implementation of device capabilities, real network topology and communication technology. In order to fill this abstraction gap Protelis requires the implementation of a back-end that provides a platform operation for every node in the system. A Protelis backend requires to implement:

- **the device model**, it has to display its sensors, actuators, and capabilities provided by the hardware;
- **the communication layer**, it has to perform two tasks: define the neighbourhood policy, and deliver the messages of each device to the others ones in its neighbourhood.

1.2 LoRaWAN

In the context of smart-city and more in general of the IoT, it is important to have devices capable of long-range communications, low battery consumption, and have the lowest possible installation, maintenance, and purchase costs. Low-power wide-area network [17] (LPWAN) is a type of wireless networks that allows to satisfy the first two requirements. Typically devices for this type of networks can communicate in a range spanning from a few hundred meters in urban areas to over 10 Km in open-spaces areas. These networks do not only allow to satisfy the first two requirements, but they also contain the costs as composition of three factors:

- lightweight protocols, which allow for using devices with simple hardware and cheap;
- use of licence-free band frequency with no cost for network occupation;
- low power consumption allowing devices to run for years with a small battery.

LoRaWAN[1] is one of the main LPWAN protocol. A LoRaWAN network, Figure 1.2, is composed of three fundamental elements: end-devices (aka motes), gateways (aka concentrators) and a network-server. The network topology typically is a star-of-stars topology with the gateway that intermediates between motes and network-server. The communication between gateway and network-server is IP-based, while the communication between gateways and motes uses the LoRa (Long Range) modulation.

The role of network-server is to govern and optimise the interactions between motes and gateways and provides the mote's packet to applications in the application server. So among other tasks, it has to filter duplicated packet received from gateways and find the best gateway to deliver an application message to a mote. The LoRaWAN protocol, Figure 1.3, is composed of two layers: the physical layer and the MAC one.

1.2.1 Physical Layer

The physical layer implements the LoRa protocol allowing communication until 15 Km of distance, with a data rate from 0.3 Kb/s to 11 Kb/s with LoRa modulation, and to 50 kb/s with FSK modulation. The most important parameters for LoRa modulation are Bandwidth (BW), Spreading Factor (SF) and the Code Rate (CR). The SF value is in the range from 7 to 12 and represents the number of bits in each symbol. It is an important parameter because it defines: the maximum

[1] https://lora-alliance.org/sites/default/files/2018-07/lorawan1.0.3.pdf

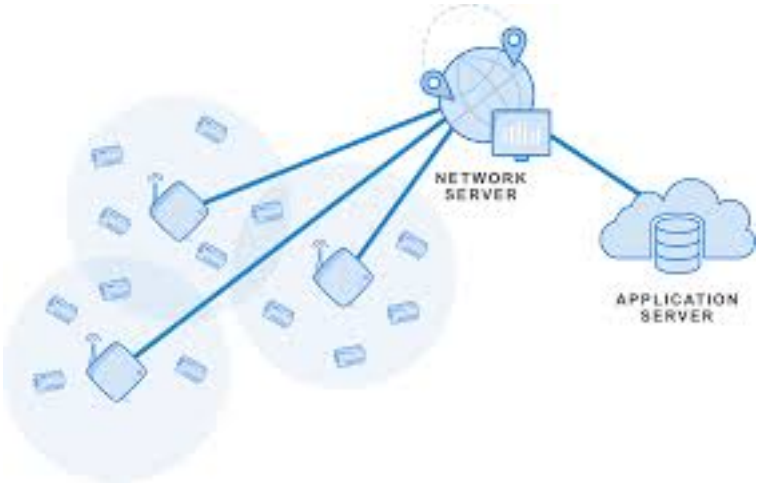


Figure 1.2: LoRaWAN network architecture [12] with all the fundamental elements. Straight lines represent IP-based communication, while the circle around the gateways is the communication range based on LoRa modulation.

packet length, the range of communication, and the bit rate. SF7 means longest packet length and highest bit rate, but shortest communication range; while SF12 means shortest packet length and lowest bit rate, but longest communication range. Another important aspect of SF is that concurrent transmissions with different SF will not produce any collision between them. Finally, LoRa uses the regional Industrial, Scientific and Medical (ISM) band. This means low network cost, but limitation of the duty cycle to maximum 1% for each channel imposed from the European telecommunications standards institute (ETSI).

1.2.2 MAC Layer

The MAC layer regulates the communications between motes and gateways. Uplink communication (mote to gateways) uses an ALOHA-like protocol. Motes start a broadcast communication when they need without checking the channel state, but applying a small random delay. For downlink communication (gateway to mote), in order to reduce the battery consumption of the motes, and respect latency requirement of different applications, the layer defines three different classes of devices (A, B and C) with different behaviours. Devices of class A define two fixed receiving windows after each uplink communication, with the second one opened only if the mote receives a communication during the first. This class has the lowest battery consumption with the drawback of the highest latency. Devices of class B allow to define more receiving windows, but it is necessary to keep motes and server synchronised via synchronised Beacons. This class has lower latency

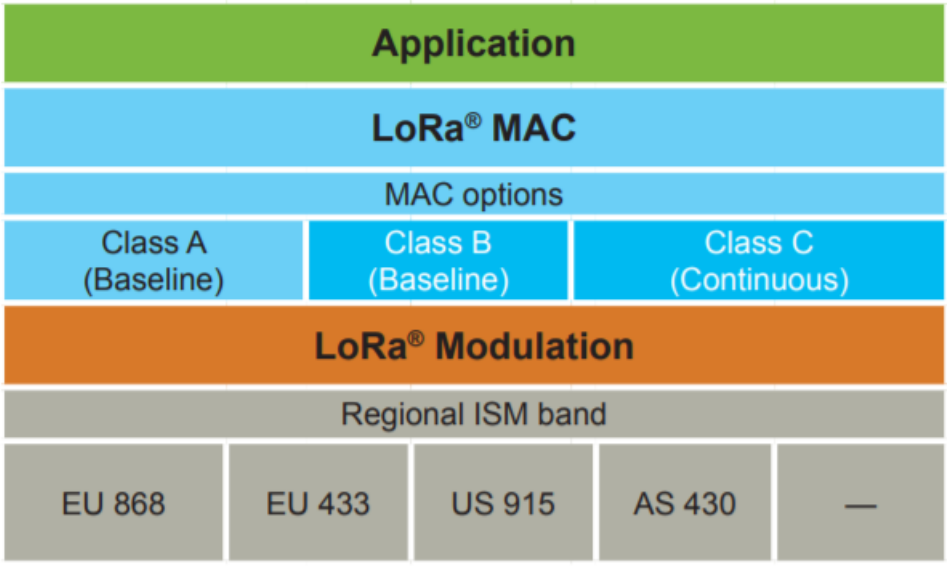


Figure 1.3: LoRaWAN protocol stack [1] where brown rectangle represent the physical layer and the azures rectangles the MAC one.

than class A but also higher battery consumption. Finally, devices of class C always allow to receive messages except while executing an uplink communication. This class has the lowest latency, but the highest battery consumption.

1.3 DingNet: a LoRa-over-MQTT network and simulator

The LoRaWAN protocol does not standardise communications after the gateways, but only defines that they are IP-based. LoRa-over-MQTT identifies a network where the communication between gateways and network-server is implemented using the publish-subscribe pattern via MQTT protocol, and the application server provides data to applications in different ways, but at least via MQTT. An example of LoRa-over-MQTT architecture is ChirpStack[2], see Figure 1.4, that is an open-source LoRaWAN Network Server stack to manage all the LoRaWAN network components and provides data to applications in several ways.

DingNet[3] is a real LoRaWAN network of class A composed of 11 gateways that cover the entire city of Leuven and adopts a LoRa-over-MQTT architecture. DingNet allows adding users' devices and applications to the network for research

[2]https://www.chirpstack.io/ (Jan 2020)
[3]https://admin.kuleuven.be/icts/english/services/dingnet (Jan 2020)

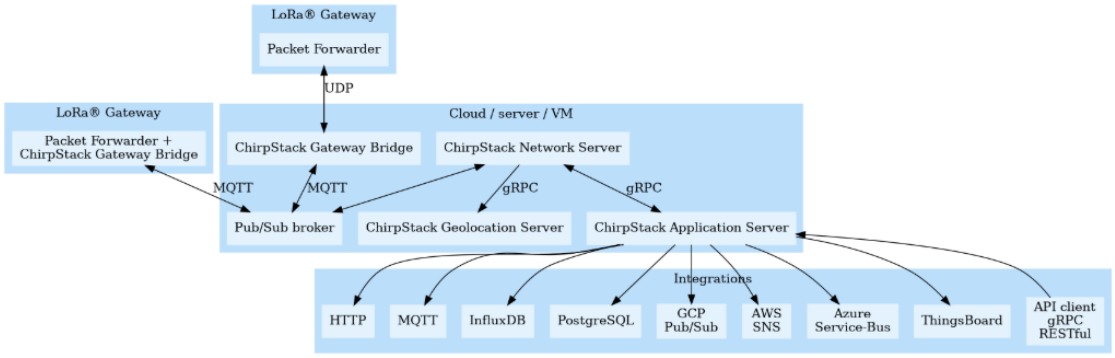


Figure 1.4: Example of the LoRa-over-MQTT architecture of ChirpStack. [9]

purposes. The DingNet Simulator [16] allows for simulation of applications in different scenarios before deploying them in the real network reducing costs and time for the deployment. It is a time-driven simulator focused on the simulation of LoRa communications between gateways and motes while the remainder of the network is simulated with a higher-level of abstraction. The simulator allows configuring the environment of the network in terms of gateways, motes, and typology of areas that compose the environment. Different terrains are represented in the simulator as different type of area; like forest, open space, and building area. Each area differs for how much the transmission power is decreased; for example, a building area decreases power more than an open space one. Motes can be stationary, but also mobile with the possibility do define their path. For each mote is possible to configure all the most important parameters for a LoRa transmission, like bandwidth, spreading-factor and transmission power. The simulator for each transmission computes the time-on-air based on the previous parameters and the packet size. Then it applies an algorithm to decrease the transmission power based on the distance from the source and typology of areas that the transmission is going through to find all the gateway inside the communication range. Every transmission is considered arrived at a gateway when enough time (time-on-air) has passed from the beginning of the transmission. Finally, the gateway applies a particular algorithm to check if the transmission collided with others. If not it publishes the received packet on the MQTT broker for the applications. At the end of each simulation is possible to export different information for mote's transmission:

- received power from the gateways (dBm);
- time on air (ms);
- used power for the transmission (dBm);

- used energy for the transmission (mJoule);
- collision with others transmissions (true/false).

Nowadays, the simulator is used mainly to simulate self-adaptive applications in large scenarios to increase the number of transmissions correctly delivered to at least one gateway reducing the energy consumption from the motes.

Concluding remarks. This chapter introduced the aggregate computing paradigm, discussing its benefits for pervasive and heterogeneous systems, and the language Protelis. Then it introduced LoRaWAN, which is one of the main LPWAN protocols. Finally it introduced DingNet a LoRaWAN network and its simulator.

Chapter 2

Contribution

This chapter explains the contribution of this thesis. Section 2.1 starts exposing all the work done to extend and evolve the DingNet simulator; Section 2.2 illustrates the work done to support the execution of aggregate computing programs over the DingNet simulator, and in particular Protelis programs.

2.1 Extension and evolution to DingNet

This section exposes all the improvements and extensions applied to the DingNet simulator to achieve an extendible and configurable simulator, which simulates the entire LoRa-over-MQTT network. Previous work on the simulator were focused mainly on three areas:

1. **LoRaWAN communications**: simulation of bi-directional communication between LoRa motes and LoRa gateways;
2. **GUI**: provide a good user interface that simplifies the configuration of simulations and allows the user to see the simulations results;
3. **Self-adaptive applications**: evaluation of algorithms to reduce the energy consumption for LoRa motes communications, but ensuring that each transmission is received at least from one gateway.

The following part of the section discusses the contribution to the DingNet simulator (part of this work has been done in collaboration with Federico Quin, a PhD student at KU Leuven).

2.1.1 Platform requirements

This section illustrates the identified requirements that the platform DingNet has to satisfy to enable simulations of applications over the LoRa-over-MQTT network stack. The requirements are grouped in functional requirements, and non-functional requirements.

Functional requirements

The functional requirements are the following:

- **Entities displacement and motes mobility:** the simulator has to allow to displace both the entities (gateways and motes) in the entire simulation region. It has also to allow the mote mobility in each direction to emulate realistic movement;
- **LoRa transmission:** the simulator has to simulate the communication between gateways and motes. This means emulate the propagation of LoRa transmissions based on the source configuration. It has to compute the transmission power decay, considering the distance from the source and the terrains crossed by the signal, in order to identify the gateways that receive the transmission. Finally, it has to be able to detect the collisions among different transmissions;
- **Configuration of the LoRa transmission parameters:** the platform has to support the configuration of the parameters that effect the LoRa transmissions, verifying the validity of their values. These parameters are transmission power, spreading factor, bandwidth, and code rate;
- **Communication protocol gateways-motes:** the MAC layer of the LoRaWAN protocol defines an ALOHA-like protocol to regulate the communication from mote to gateways, and three different interaction schemes to standardise the communication from gateway to mote. The simulator has to implement the simplest type of ALOHA protocol and adopt the interaction schema for the devices of "class A" as default. However, it has to allow to change these protocols in future to evaluate the simulated systems with different ones;
- **Managing incoming message mote-side:** the motes entities are able to receive packets from the network-server for network administration purposes, and from the applications. So, the platform should allow to manage the incoming packets mote-side, applying actions with side effect to the mote or to the environment in accord with the packet payload;

- **Application layer:** it is the layer that includes the interaction between the gateways and the applications, and the involved entities. This platform want to simulate LoRaWAN networks with LoRa-over-MQTT architecture, so the interactions between gateways and application are intermediated by the network-server and they are based on MQTT. This layer can be composed by several services (for example the geolocation one), but the only mandatory is the network-server;

- **Time-frame simulations:** the platform has to allow the user to perform simulation with a predetermined duration even for more than a day. It is useful to verify how the system reacts in different scenarios and conditions that can happen in large time-frames;

- **Configurable sensors:** the behaviour of the simulated applications can depend from the sensed data received from the motes. So, it is important provide configurable sensors, that guarantee to the user to define the values to produce in each zone of the simulated region in every instant. This is useful to simulate the applications in different scenarios, and to assure the simulation reproducibility;

- **Transmission statistics:** the simulator should be able to export statistics about the transmissions to evaluate and compare the different solutions simulated. Information of interest are the transmission received power from the gateways, the energy consumed by the motes, the number of collision among transmissions, the time on air of each transmission, and the transmission source position.

Non-functional requirements

The identified non-functional requirement concerns with the **project maintainability**. In this context techniques to automatise the project life cycle have to be applied, and mainly to automatise dependency management, project build, execution of all the tests in fresh environments, enforce adoption of a common style, and optionally also deployment of the project. Automatise all these activities is useful to increase the productivity and maintain a good quality project over time.

2.1.2 Problem analysis and Design

This section discusses the requirements above illustrated and exposes the adopted solution. Requirements as simulation of LoRa transmissions, configuration of their parameters, and export of transmission statistics are already satisfied, so they are not further discussed.

Entities displacement and motes mobility

DingNet already allows these features, but with limitations due to the spatial reference system (SRS) of the simulation environment. The SRS is a discrete one and this leads to:

- low precision in displacement of gateways and motes in the environment;
- mobile motes can move only in horizontal or vertical directions.

In order to increase the precision for the displacement of the entities and move mobile motes in each direction with straight lines, the discrete SRS is converted to a continuous one.

Communication protocol gateways-motes

Actually the simulator encapsulates the communication protocols of gateways and motes inside the class *NetworkEntity*, that is their base class. This does not allow to evaluate simulated systems with different protocols, but more important this means that gateways and motes can apply only the same protocol. The architecture proposed in Figure 2.1 generalises the behaviour of the two entities exporting the protocol logic outside the base class with the two strategies *Sender* and *Receiver*. It grants to define different protocols for each entity allowing the motes to apply a ALOHA-like protocol, and to regulate the communication gateway-mote with the interaction schemes defined by the classes of device. Finally, it also allows to evaluate the network behaviour with several variants of protocols.
Sender interface defines methods to send a packet, check the transmission status, and manage parameters that influence the transmission; while the *Receiver* interface defines methods to receive incoming transmissions, and allows to the *NetworkEntity* to specify how manage them.

Managing of incoming message mote-side

The simulator partially supports the managing of incoming message mote-side: it is designed only the structure to manage the *MAC Commands* (special commands exchange between network server and motes for network administration) motes side. The architecture illustrated in Figure 2.2 is designed to complete the managing of incoming messages to a LoRa mote, allowing to use all the information contained in the payload. *ReceivedPacketStrategy* defines the strategy to use to store all the incoming packets and manage the pending queue of packets to consume. *ConsumePacketStrategy* defines how to use the information in the payload to produce a side-effect on the LoRa mote or on the environment. Every mote

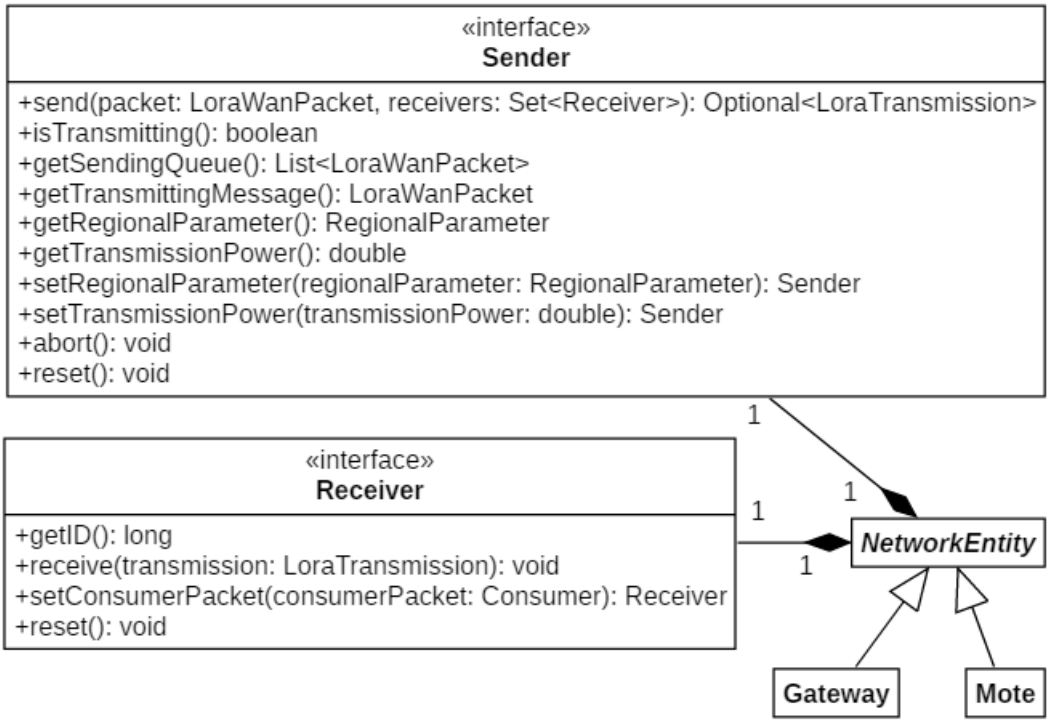


Figure 2.1: *NetworkEntity* architecture with *Sender* and *Receiver* interfaces.

can have a list of *ConsumePacketStrategy*, which are performed in an ordered way with strategies that can use or ignore the packet. Figure 2.2 shows two implementations of *ReceivedPacketStrategy*, and none of *ConsumePacketStrategy*. This because the first strategies are cross domain while the second ones are domain specific in respect of the simulated application.

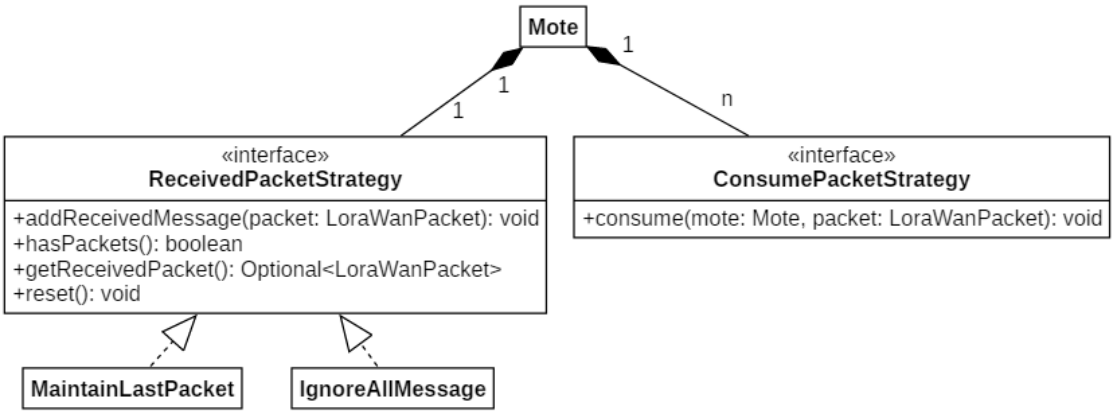


Figure 2.2: Architecture to manage incoming packets mote side

Application layer

The actual application layer provided by DingNet connect directly gateways and applications without the mediation of the network-server, and without use MQTT as communication technology. So, in order to obtain the desired simulation platform, and provide realistic simulation over the LoRaWAN stack it is necessary to implements both.

MQTT communication

MQTT has to be used as communication technology for both the bi-directional communications gateways to network-server and network-server to applications, but there are no requirements that obligate to use the same broker for both the interactions. During the problem analysis phase the following requirements were identified for the MQTT client:

Req1. allow simulations with different client abstractions, optimisations, and realise a simulator technology independent for MQTT client's implementation;

Req2. avoid MQTT messages conversions to domain specific objects at each topic subscription.

In order to fulfil Req1 it is necessary to allow to switch client implementation in a simple and fast way (for example from a mock implementation to a real one). To do so the interface *MqttClient* is defined, which allows to avoid to use directly a particular implementation. It represents a MQTT client and it provides all the basic functionalities to interact with a MQTT broker. This interface grants to use custom implementations of MQTT client or external libraries implementing a wrapper extending the interface. In order to satisfy Req2 it is necessary to delegate conversions from and to the MQTT message type to the client, which interacts with the broker and knows how manipulates them. This requires to specify the receiving message type during the topic subscription phase. Figure 2.3 shows the *MqttClient* interface and its actual available implementations.
The designed architecture is not valid only for this project, but it is reusable in different projects, so it is decided to export it as an external library. The library is actually available on github[1] and a release on Maven Central Repository is planned shortly.

Network server

The network server is the entity appointed to regulate the communication between gateways and applications, but there isn't any specification that defines which tasks

[1] https://github.com/Placu95/MqttClientWrapper

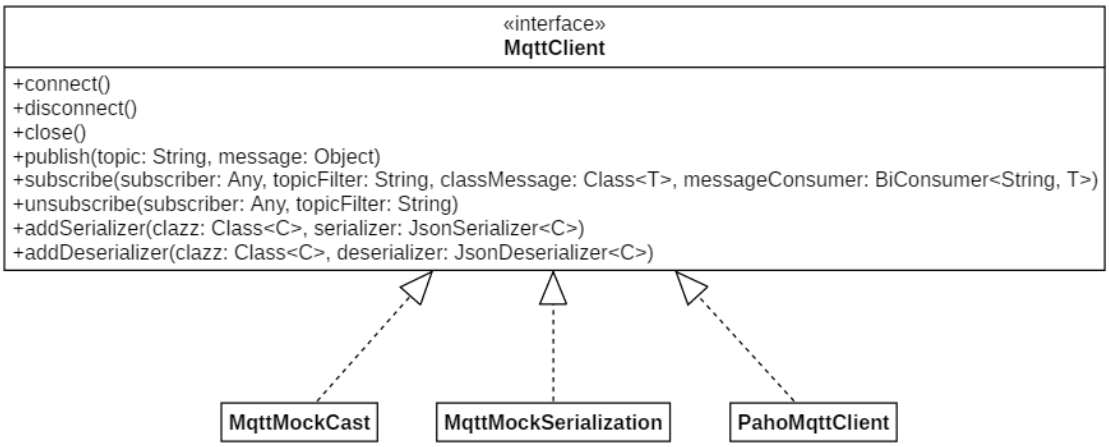


Figure 2.3: *MqttClient* interface and its available implementations. Thanks to *addSerializer* and *addDeserializer* methods it is possible to define custom strategies to convert messages for each client.

has to perform and which is its architecture. Different providers propose different solutions. The solution designed for the simulator is composed of an autonomous simulator entity that performs two tasks:

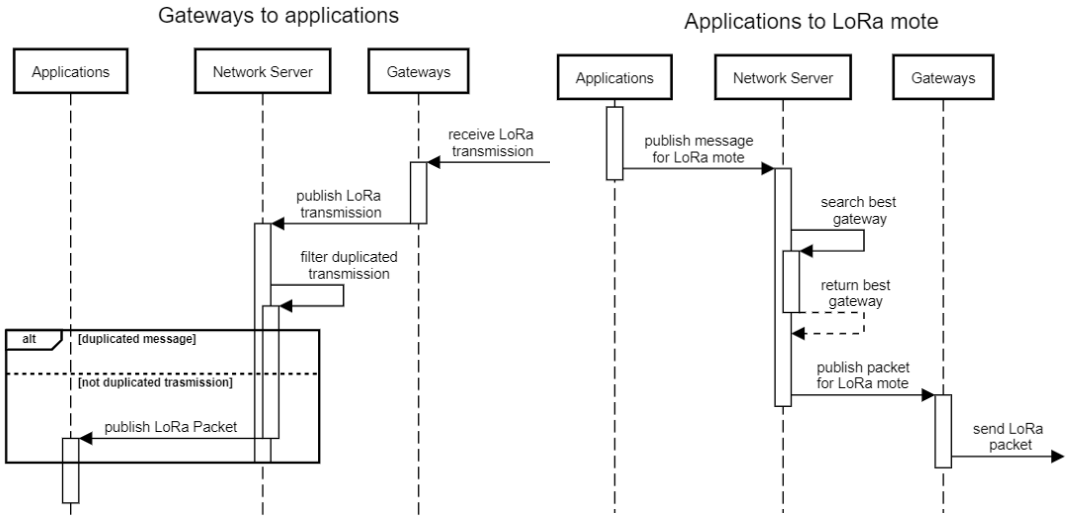
1. filtering of duplicated messages in communication from gateways to applications;

2. selections of the best gateway to deliver an application's message to a LoRa mote. Default strategy to choose the gateway selects the gateway that has received the last transmission from the LoRa mote with more transmission power, but it is possible to change it defining different strategies.

Figure 2.4a introduces the new communication schema from a gateway, which receives a LoRa transmission, to the application; while Figure 2.4b introduces the new communication schema from an application, which wants to send a message to a LoRa mote, to the selected gateway that performs the LoRa transmission.

Time-frame simulations

The platform actually does not support time-frame simulations, but only single run simulations, and multiple run simulations. A single run simulation finishes when all the mobile LoRa motes arrive at the respective destination, while a multi run consists simply in run more time a single run simulation. So, it is necessary to define a new type of simulation that satisfy the requirement. Another problem is the impossibility to perform simulation of more than a day dues to the actual



(a) Communication from gateways to application

(b) Communication from application to gateways

Figure 2.4: Communications between gateways and applications

time representation. *Timed run* is the new type of simulation introduced to go beyond the limitations of the single run one. This type of simulation differs from the single run only for the termination condition. The condition requires to define the duration of the simulation, so the condition evaluates it as finished when the defined time is passed. In order to enable simulation of more than a day, it is

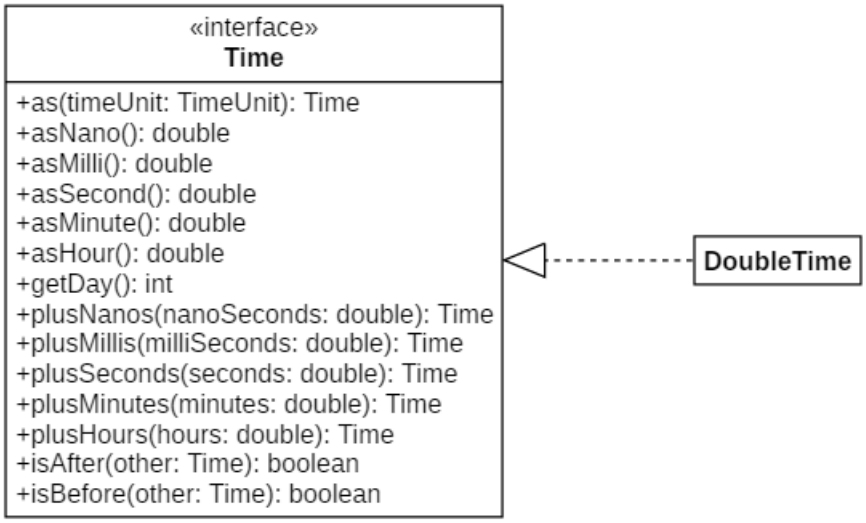


Figure 2.5: Time representation that provides all the basic functionalities.

necessary to change the time representation from the java class *LocalTime* to a new one. The new time representation is proposed in Figure 2.5. It does not enable only multi-days simulation, but also it simplifies the time manipulation providing methods for the different unit of measure, including the milliseconds that are the default one used by the simulator.

Configurable sensors

DingNet already defines its own concept of sensor defined by the interface *SensorDataGenerator*, and some its implementations. The problem is that miss a support to configure with low effort the values that the sensors has to produce. In order to solve the problem, it wants to define an extendible architecture starting from the sensor interface already defined, which grants to configure the values to produce. The idea is to have sensors that produce values in a configurable way splitting the region of simulation in a matrix of configurable dimension. So for each cell of the matrix it is defined a list of configurations. Each cell's configuration defines the starting time of validity and the range of producible values. Then starting from this configuration is produced a tricubic spline interpolation function where the three variable are: the two coordinates of the matrix and the time; while the result is the corresponding sensor value. Finally, when a mote requires a new value to the sensor, it produces the value considering the mote position and the simulation time. The result is that each sensor inside the same cell produces the same value at the same time. The architecture of this sensor is showed in Figure 2.6, and:

- **Cell** represent the cells that compose the configurable matrix;
- **RangeValue** is the range of validity for the value to produce;
- **RangeDataGenerator** is the abstract class that loads the configuration file of the sensor, generates the interpolating function and produces values for LoRa motes. It requires only to define the type of the two interfaces *Cell* and *RangeValue*.

In order to define new configurable sensors it is require only to implement the *RangeDataGenerator* abstract class specifying the type of *Cell* and *RangeValue*. The actual file format for the sensor configuration file is *toml*[2], but thanks to the *konf*[3] library (used to parse the file) it will be possible to change the file format with low effort.

[2]https://github.com/toml-lang/toml (Feb 2020)
[3]https://github.com/uchuhimo/konf (Feb 2020)

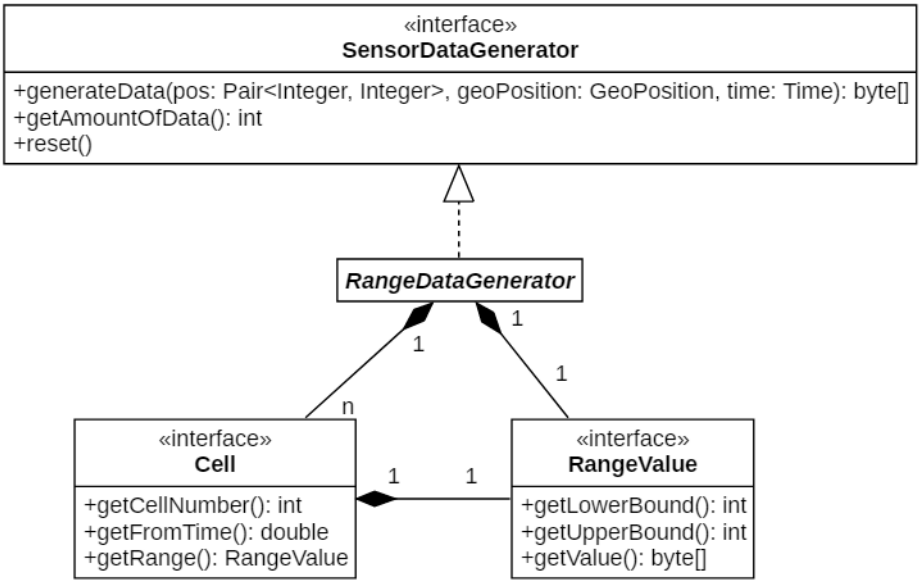


Figure 2.6: Architecture of the configurable sensor

Project maintainability

In order to automatise the dependency management, and the project build was evaluated two build tool: *Apache Maven* and *Gradle*. *Gradle* is preferred to *Apache Maven* for several reasons[4]. The main ones are: performance, highest readability of the project's configuration file due to a less verbose syntax, and better script support (with the possibility to write them in kotlin with *Gradle Kotlin DSL*). Instead, *Travis-CI*[5] is chosen as continuous integration service to run a new build of the project and execute all its test in fresh environments after every change. Travis-CI is chosen because it is free, well integrated with *GitHub* which hosts the project, and support to automatise the deployment. It is configured to test the project in all the main operative systems (linux, windows, and osx) and with different java versions (11 and 12), but is not configured to automatise the deployment. Finally, *Checkstyle*[6] is adopted to enforce the use of a common style in the entire project, and its execution is planned every time Travis-CI performs all the class of tests.

[4]https://gradle.org/maven-vs-gradle (Feb 2020)
[5]https://travis-ci.com (Feb 2020)
[6]https://checkstyle.sourceforge.io/ (Feb 2020)

2.2 Aggregate programming over a LoRa-over-MQTT network

This section discusses the integration between the aggregate computing paradigm and a LoRa-over-MQTT network like DingNet; enabling the simulation of aggregate applications on this platform. In order to join these two concepts, at first is necessary to define how to map the networks entities in an aggregate computing system, and second identifies potentially additional requirements or limitations for these entities. In the aggregate computing viewpoint, a system is composed of a set of distributed heterogeneous computational entities, called nodes. Nodes execute the same global program and communicate with a subset of them defined by a neighbourhood policy. While, in a LoRaWAN network the main entities are: LoRa motes, gateways, and network server; but at application level the only interesting entities are the LoRa motes. So, following the aggregate vision it is natural to map each LoRa mote in a node that represents its digital twin (from now on called LoRa node) inside the aggregate system.
On the one hand, LoRa nodes can be considered as generic nodes and they do not require any specific neighbourhood's policy, or the use of particular communication technology to interact with their neighbours. But on the other hand, they have to support communication via MQTT to enable bidirectional communication with the respective physical counterparts. Finally, it is necessary to analyse if all the network entities, or only some of them, can host the aggregate nodes. In [13] the authors propose a software architecture to enable the execution of aggregate computing programs on LoRa motes, but after evaluations of the proposed solution, they identify some issues due to the physical limitations of the communication technology. These issues do not allow to execute aggregate computing program directly on the LoRa motes, but they are not valid for the other network entities and there is any paper that identifies others possible issues. Even if all the LoRaWAN network entities excluded the motes can host the aggregate node (LoRa nodes or other types of nodes), it is important to specify a limitation for the LoRa nodes. The LoRa nodes can interact with the respective LoRa mote only following the interaction schema defined from the LoRa-over-MQTT networks. For example, if a LoRa node is hosted by a LoRa gateway, that receives directly the transmissions of the respective LoRa mote, it cannot receive directly the packet, but it has to wait that the packet is published on the MQTT broker.

2.2.1 Integration of Protelis with DingNet

The only things to do to enable the development of Protelis applications over the DingNet network is to satisfy the requirement previously identified. That

requirement represents a specific capability of this type of nodes and Protelis defines a single place appointed to host it, the *ExecutionContext*. Figure 2.7 shows the model of the *ExecutionContext* designed for the LoRa nodes.

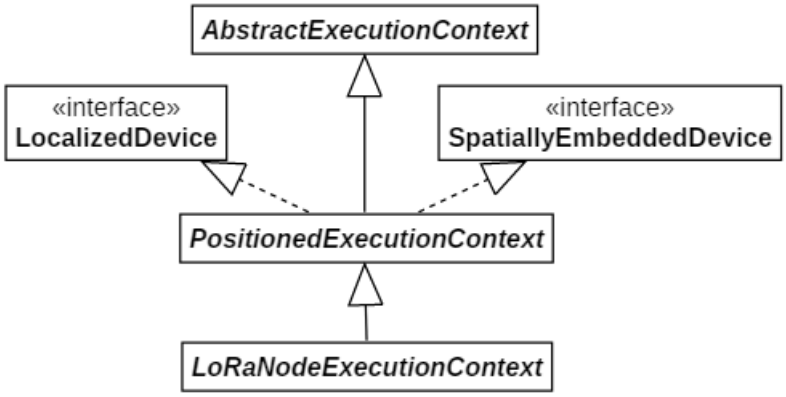


Figure 2.7: Model of *ExecutionContext* for LoRa nodes

For devices spatially embedded and mainly used to transmit sensor values, their position is a relevant information. So, *PositionedExecutionContext* is defined, which extends *AbstractExecutionContext* (provided by Protelis) implementing the two interfaces that define functions to obtain spatial information of the device. *LoRaNodeExecutionContext* is the basic execution context for a LoRa node that:

- adds support for MQTT communication, satisfying the identified requirement;
- encapsulates the subscription to the MQTT topic to receive the sensed values from the respective LoRa mote;
- manages the received packet adding the sensed values to the knowledge-based of the node, or modifying its position if the value belongs to the GPS sensor.

The introduced model enables the design of Protelis application composed of LoRa nodes, but also of not LoRa nodes. Protelis applications with the model illustrated in Figure 2.8 are not only valid for the DingNet network but for all the LoRa-over-MQTT networks that provide LoRa mote's data on a MQTT broker. In order to execute Protelis applications on the top of DingNet simulator, enabling the simulation of Protelis applications composed of LoRa nodes, one last small operation is necessary: unify the concept of time. This operation is necessary because the behaviour of every Protelis node depends also from the time in which their execution is scheduled, and the LoRa packets are received. To do this it is sufficient wrapping the *LoRaNodeExecutionContext* using the simulator time concept.

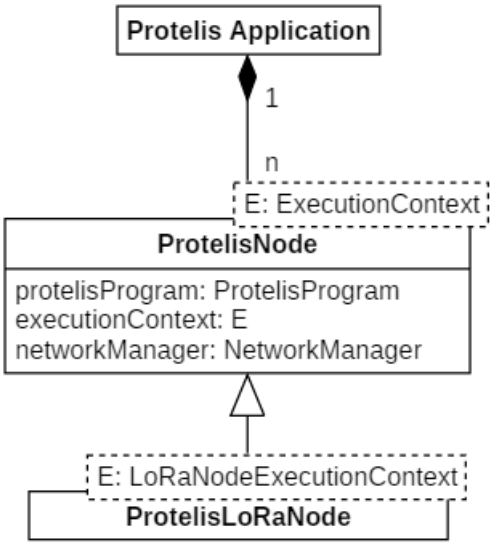


Figure 2.8: Abstract model of a Protelis application composed of LoRa nodes

Concluding remarks. This chapter discussed the main works done during this thesis. First it exposed the main improvements and evolution on DingNet simulator. Then it illustrated the support to simulate Protelis application inside the DingNet simulator.

Chapter 3

Case studies

This chapter illustrates two case studies developed on the DingNet simulator.

The case study in section 3.1 shows the DingNet platform after its extension. The main new features used are the application layer that enables the communication between LoRaWAN gateways and applications, the managing of the incoming messages mote-side, and the new type of sensor. To do so, a system that requires a bi-directional communication between the application and the motes, and with the application behaviour dependent on the payload of the motes packets is conceived.

The case study in section 3.2 exploits all the new features added in the DingNet simulator. It uses the integration for the execution of Protelis application over DingNet, and the new type of simulation (*TimedRun*) in addition to the features already used in the previous case study.

3.1 Case study: Pollution-aware user navigation

Leuven is a cycling city where most of the inhabitants and students use daily their bike to move across the city. In this context we want to realise a system able to provide to the user the healthiest route to reach a destination. The route generation will be based on the air quality level, based on the CAQI index[1], of the crossed areas to reach the destination. The user, in order to obtain the route, has to require it to the application deployed in a remote server. The system uses data received from the sensor network to create a city map of quality air. Then, the application has to define the route to a destination that optimise the trade-off between air quality and length of the route. The system should be able to recompute the best route if the environment condition change, and communicate it to the user. The sensor network can be composed of two types of sensors:

[1] https://www.airqualitynow.eu/download/CITEAIR-Comparing_Urban_Air_Quality_across_Borders.pdf

- **Fixed**: positioned along the roads and at intersections;
- **Mobile**: placed on public transport or bicycles.

All the sensors have to be deployed in a LoRaWan network. Similarly also the user device, which interacts with the application to require the route, has to use the LoRa technology.

3.1.1 Design of the system

Figure 3.1 shows the high level architecture of the system, and introduces the main entities. The main entities are the sensors devices, the user devices, and the routing application. As requirement both sensors devices and user devices have to be displaced inside the LoRaWAN network and use the LoRa technology to communicate with the routing application. Using the DingNet simulator to simulate the LoRaWAN network:

- the routing application is mapped in a generic application deployed in the application server that communicates with the LoRaWAN network via MQTT;
- the sensor devices, which have to send only packet with the sensed value, can be mapped in the *Mote* simulator entity;
- the user devices are special devices, because they do not send only packets with the sensed value. They have also to require the route for a destination and be able to manage the received packets with the route. Actually there is not simulator entity with these specific abilities, so it will be necessary to define it.

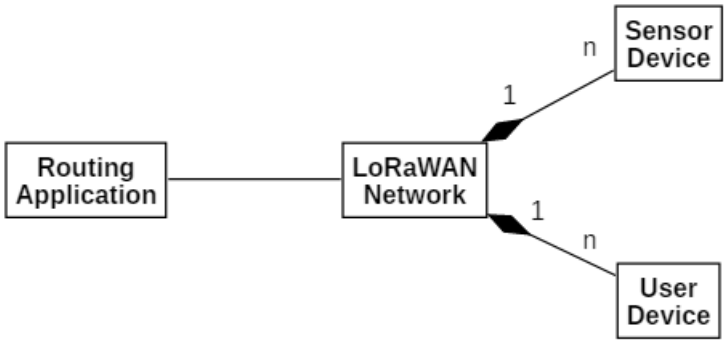


Figure 3.1: High level architecture of the system.

Interaction between application and devices

If on the one hand the transmissions from the devices (both sensor and user) to the application are in compliance with the maximum packet's length defined by the LoRaWAN standard (1 byte for packets with the sensed value, and 16 bytes for the packet to require the route); on the other hand it is impossible to send the packet from application to the user device with the entire route, so the only way to do it is to split the packet. In order to send the entire route to the user device, two approaches are available:

1. define a specific interaction protocol to send all the packets with the entire route to the user device immediately after its computation;

2. send a packet with part of the route only when the user has finished the previous part of the route.

Considering also the requirement to recompute the route if the environment condition change, the second option is chosen, enabling to recompute the route before send its next part.

User device model

The user device has to perform three activity: require the route, send updates of its position, and manage the incoming packets. If on the one hand the last two activities can be performed also from a *Mote*, on the other hand it cannot require the route. For this reason in Figure 3.2 a new entity simulator is introduced, the *UserMote*. It extends the *Mote* adding the logic to require the route when needed sending a packet with starting and destination positions. Then to manage incoming packets are chosen *MaintainLastPacket* as the strategy to store them until that they are consumed, and *ReplacePath* as the only strategy to consume them. It consumes each packet updating the user route in accord with the packet payload until the route is completed. To perform the last activity, send updating of the user position, is only necessary to add the GPS sensor to the *UserMote*, and send it when the user is closer to the sub-route destination.

Routing application

The application to find the best route implements an A-star algorithm on the graph of the streets of the city. The weight of the edges of the graph corresponds to the distance between the two points multiply for a factor that represents the air quality level in that street. The values sensed by the sensors are retrieved subscribing to the topics where the LoRaWAN network publishes them. The route is sent to

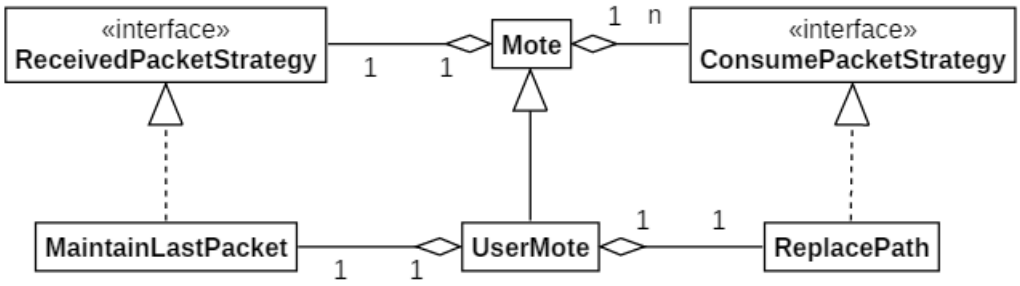


Figure 3.2: *UserMote* model.

the user mote publishing the massage with sub-route on its receiving topic. The application recomputes the best route only when the user device communicates its new position and if some environment condition is changed from the previous computation.

3.1.2 Simulation in DingNet

Here, simulations of two possible small scale scenarios are illustrated. The simulations are conducted over the city of Leuven with the DingNet simulator. First, the setup of the two simulated scenarios is presented; then the simulation results discussed in a qualitative way based on simulation snapshots.

Setup

Both scenarios has a common configuration and differ only for the starting position of the user. The environment is composed of:

- 2 gateways;
- 4 fixed sensors equipped with a set of sensors and send directly the final CAQI index value;
- one mobile sensor that follows a path like a public transport. It assumes the same behaviour of a fixed sensor, but is equipped also of a GPS sensors;
- user that requires a route to a destination.

All the types of LoRa devices are configured in a way to try to reduce collision between transmissions. In the first scenario the user starts from the south of the city, which is the most polluted in the simulation configuration; while in the second scenario the user starts from the north of the city.

Results

Figure 3.3 and Figure 3.4 show snapshots taken from the two simulated scenarios. The transparent layer represents the air quality level in that location based on the received sensor value. The mote number 4 is the mobile sensor and the red line is its route. The user is identified by the cyclist. Its line (route) is of two colours: blue and red. The blue one is the complete route computed by the application, while the red one is the sub-route received by the user until that time.

Figure 3.3 shows three snapshots of the first simulated scenario. The first snapshot shows the initial situation and the first computed route. The second snapshot shows how after an environment conditions change, the route is recomputed with a longer one, but considered better by the application, combining distance with pollution. So the user has received a new sub-route of the new best route that starts from its actual position. Finally, the last snapshot shows the user arrived at destination.

Figure 3.3: Three snapshots of a simulation run with changing of the route dues to the change of the environment conditions.

Figure 3.4 shows three snapshots of the second simulated scenario. In this case the user best route never changes because the change of the environmental conditions do not affect the areas crossed by him.

Although they are simple simulations with few sensors and only one user device, it can be deduced that use LoRa technology also for user devices is possible only in certain scenarios. In these scenarios the number of transmissions necessary to transmit all the route is high, so according to LoRaWAN limitations it is possible only for short routes. Moreover considering a real scenario where the number of user is higher (es. more than one thousand) the network congestion will increase with also the probability of collisions among transmissions, which require re-transmission worsening the situation.

Figure 3.4: Three snapshots of a simulation run without changing of the route despite the change of the environmental conditions.

3.2 Case study: Monitoring and control of air quality

Nowadays air pollution is a very common problem of cities of all over the world. Two of the main strategies used to reduce the emission of polluting gas are traffic bans in strategic city's areas, and maximum temperature allowed in public and private building heating.

In this context we want to realise a monitoring system for the air quality based on the CAQI index, which is able to apply strategies to maintain under control the air pollution level. The sensor network is composed of a set of fixed sensors scattered around the city, and mobile sensors placed on public transports (like bus or public bicycles). All the sensor devices are equipped at least with a sensor for the particular matter 10 (PM10). The idea is to displace strategically the fixed sensors to achieve good city coverage, using mobile sensors for its refinement and to reduce reading errors of the fixed ones.

In a first step to reduce air pollution it has been chosen to control the maximum temperature allowed for the heating of buildings. The idea is to allow the system to manage the building heating systems control devices (from now on building devices). So the system can set the maximum reachable temperature based on pollution level of its area.

All the sensors have to be displaced in a LoRaWAN network to communicate their sensed data to reduce their cost and the cost for their maintenance. Building devices have not any particular requirements, they have not problems of energy consumption because they can be connected to the power line of the building. The same policy can be adopted for the Internet connection, this allows to avoid to use LoRa technology reducing the number of LoRa devices in the network and increasing their communication capability.

3.2.1 Design of the system

Aggregate computing is a good approach for this system for several reasons. First of all, it is a heterogeneous system composed for at least two types of devices (sensor and building device) with different capabilities like connectivity, computational resources, and their interaction with the environment. Furthermore it is composed of an high number of devices (one device building for each house of the city, a set of fixed sensors, and a set of mobile ones) so scalability can be a problem. But with aggregate computing is possible to solve it scaling horizontally and moving the computational node in different network devices without the need to change the program. Figure 3.5 shows a high level architecture of the system.

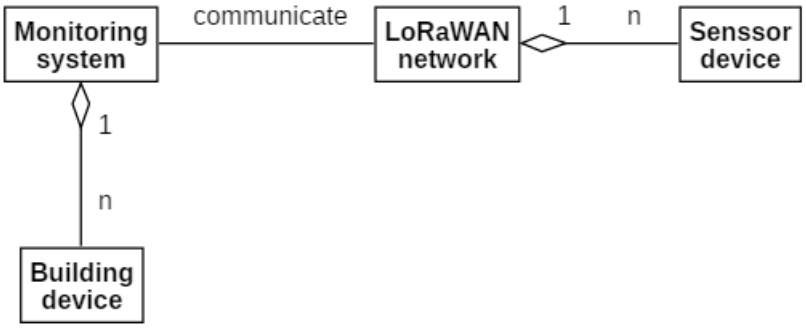


Figure 3.5: High level architecture of the system

Designing the system with aggregate computing is important to model the two different kinds of devices and the communication layer of the aggregate nodes.

Sensors and building devices models

The sensor devices are LoRa motes, so they are mapped in *Mote* inside DingNet, while, according to section 2.2.1, in the aggregate application they can be mapped in simple *ProtelisLoRaNode*. The building devices are not LoRa motes, so they do not require to be mapped in *ProtelisLoRaNode* and they can be generic Protelis nodes. Anyway it is decided to map them in *BuildingNode*, which extends *ProtelisLoRaNode*, Figure 3.6a. This because:

- the building node has not a physical mote then its topic will never receive a MQTT message, so it has not any overhead;
- all the types of nodes have a base *ExecutionContext* where introduce functions domain specific;
- if in the future they will have a physical mote, it will not be necessary to change the system architecture.

Figure 3.6b completes the model of the entities with their *ExecutionContext*. *SensorExecutionContext* updates the knowledge-base of the node computing the CAQI index at each new sensed data received. It contains also all the methods for the logic domain specific. *BuildingExecutionContext* extends it adding the capability to modify the temperature in its physical counterpart.

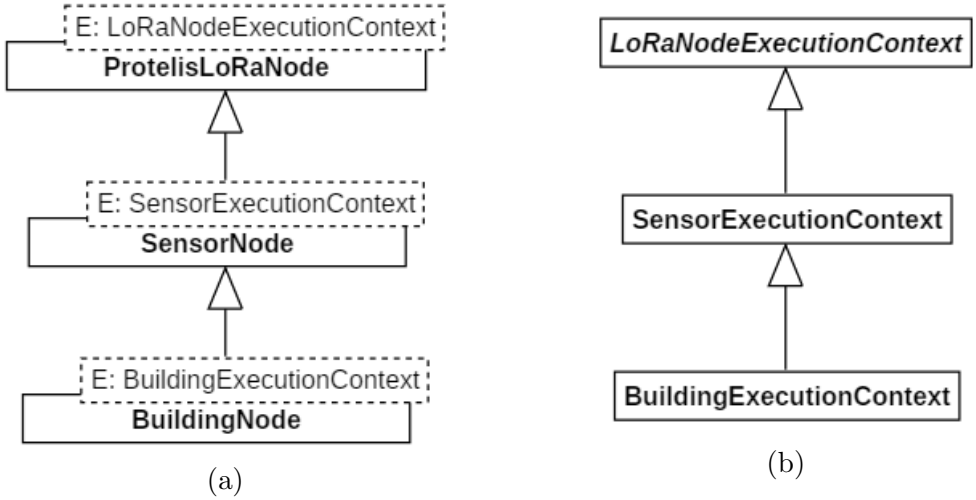


Figure 3.6: Model of the sensor and building entities.

Interaction between Protelis nodes

In order to complete the Protelis backend is required to implement the communication layer. It has to:

1. define the neighbourhood policy to determine the neighbours of each node;
2. design and implement the communication between the Protelis nodes.

As neighbourhood policy a distance based one is chosen. This policy is chosen considering the application domain, in fact the behaviour of each node depends on the environment state in its area.
MQTT is chosen to implement the *NetworkManager* and enable the communication between Protelis nodes. MQTT is chosen because it is a lightweight protocol and enables devices to send the same message to more devices with only one communication. This is very important in a large scale system where the nodes are displaced in many places and the connectivity can go down.
This system is composed also of mobile nodes and when one node change position its neighbours can change, as the neighbourhood of other nodes. So the definition of the neighbourhood cannot be done only at configuration time, but it has

to performed also after. The *NeighborhoodManager* is an autonomous entity defined to manage the neighbourhood of all the nodes. It receives the update of the node position, recomputes the neighbourhoods, and communicates them to all the nodes. This entity allows also to modify the composition of the system at run-time, adding or removing entities with all the neighbourhood updated automatically.

3.2.2 Protelis program

After discussing the design of the Protelis back-end, this section introduces the Protelis program for the global behaviour of the system. The program, visible in Listing 3.1, requires only 25 lines of code. Methods *decreaseTemp* and *increaseTemp* modify the temperature of the building devices of a delta temperature every half an hour. These methods are built on top of the function *cyclicFunction*, which is contained in the developer API of the aggregate stack. Lines 16 - 23 first create a computational field of sensed values and distances from the respective sensor. Then they manipulate it defining a field of maximum temperature allowed for each device based on CAQI index. The final part of the program defines the target temperature for each device and selects the correct method to achieve it.

3.2.3 Simulation in DingNet

Here, a simulation of a possible scenario is illustrated. The simulation is conducted over the city of Leuven with the DingNet simulator. First, the setup of the simulated scenario is presented; then the simulation results are discussed in a qualitative way based on simulation snapshots.

Setup

The configuration of the simulation provides for an environment composed of the following entities:

- 9 of the 11 DingNet network gateways (the others two are outside of the simulation region);
- 8 fixed sensors equipped with the PM10 sensor;
- 2 mobile sensors equipped with PM10 and GPS sensors;
- 3 building devices displaced in three different areas of the city.

All the sensors are configured to send a new measurement every hour, according to the CAQI index specifications for this polluting gas. The low number of mobile sensors and building devices is only due to visualisation purposes.

Listing 3.1: Protelis program for the monitoring application

```
module protelis:homeHeating_timer
import it.unibo.acdingnet.protelis.util.Const.ProtelisEnv.*
import org.protelis.lang.datatype.Option.*
import protelis:state:time
import protelis:utility

public def decreaseTemp() = env.put(CURRENT_TEMP, rep (
  v <- env.get(CURRENT_TEMP)) {
  cyclicFunction(1800, { roundToDecimal(v - self.getDecreaseDelta
    ()) }, v)
})
public def increaseTemp() = env.put(CURRENT_TEMP, rep (
  v <- env.get(CURRENT_TEMP)) {
  cyclicFunction(1800, { roundToDecimal(v + self.getIncreaseDelta
    ()) }, v)
})

let pollutionField = nbr(mux(env.has(IAQLEVEL)) {
  optionally([self.nbrRange(), env.get(IAQLEVEL)])
} else {
  absent()
})
let sensorValues = foldUnion([pollutionField]).filter { it.
    isPresent() }.map { it.get() }
let maxTemperature = self.temperatureByPollution(
  idw(sensorValues))
optionally(env.get(DESIRED_TEMP, JAVA_NULL))
  .map { mux(it < maxTemperature) { it } else { maxTemperature } }
  .map {
    if(it < env.get(CURRENT_TEMP)) { decreaseTemp }
    else { if(it > env.get(CURRENT_TEMP)) {
      increaseTemp } else { emptyFun } }
  }
  .orElse(emptyFun).apply()
```

Results

Figure 3.7 shows three snapshots taken from a simulation run of five days. The transparent layer represents the air quality level, which is obtained applying an inverse distance weighting on sensed values in the range of 1Km. Green colour means "very low" level, while red colour means "high" level. The two motes with a red line are the mobile sensors, while the others are the fixed ones. The building devices are represented with a black dot and a text with pattern "X/Y/Z". X is its current temperature, Y is its desired temperature, and Z is its maximum reachable

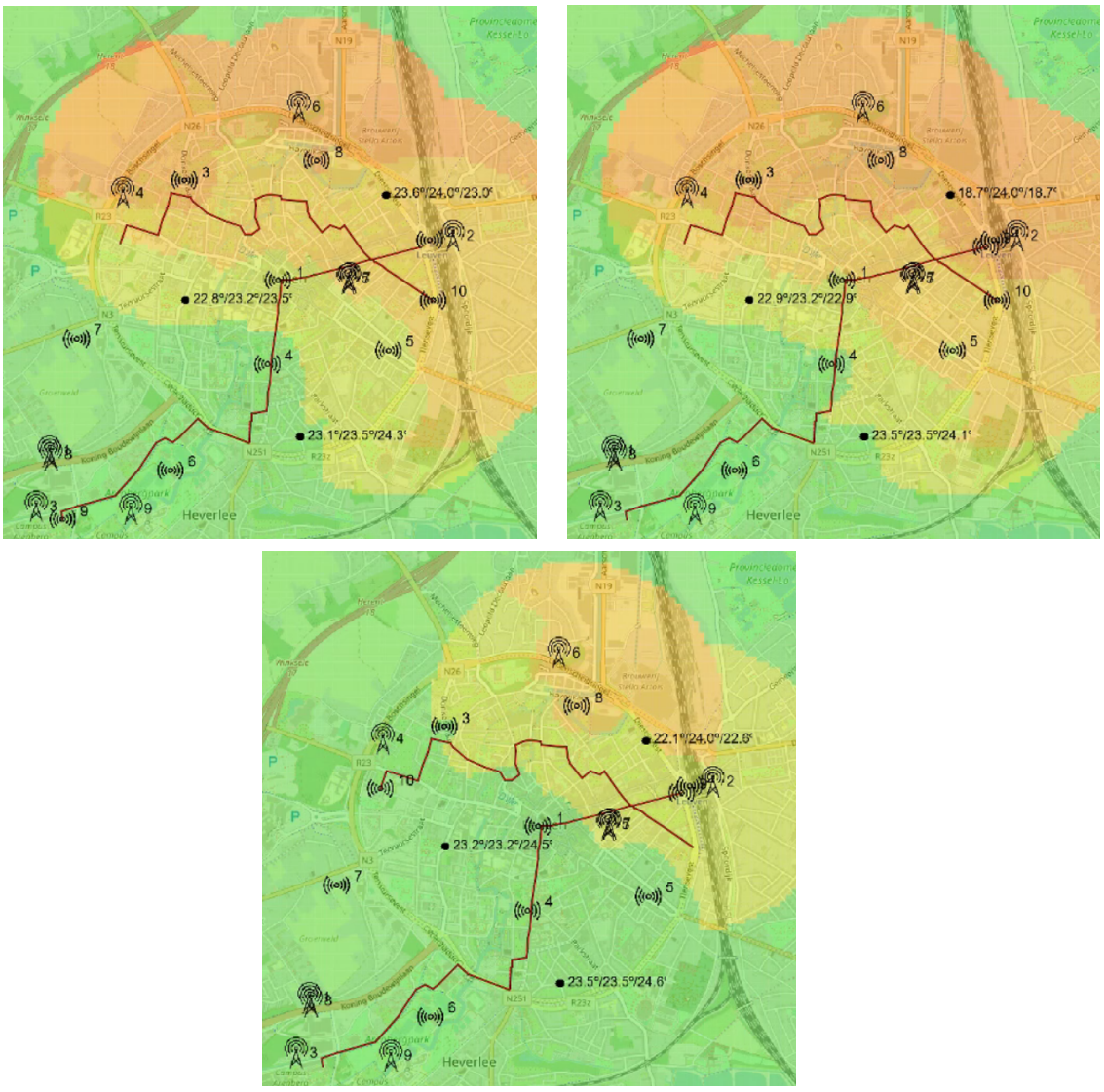


Figure 3.7: Three snapshots of a simulation run.

temperature based on pollution level. The three snapshots show how the pollution changed during the five days of simulation, and how the building maximum reachable temperature is adapted consequently. In particular the building in the north of the city has always the desired temperature greater than the maximum reachable. The building in the centre of the city has first the desired temperature greater than the maximum reachable, but after the pollution change it is the opposite and the building reaches its desired temperature. Finally, the building in the south has always the desired temperature lower than the maximum reachable.

Concluding remarks. This chapter illustrated two case-studies developed on the DingNet simulator. The first one showed the new application layer of the simulator and the new main features. It confirmed the validity of LoRaWAN as enabling technology for a sensor network and its limits regarding the communication from application to LoRa devices. The second case study showed an application developed using all the features of the platform and proved its validity as a platform to simulate Protelis applications over a LoRaWAN network.

Chapter 4

Wrap-up

4.1 Conclusion

This thesis focus on providing a platform to simulate aggregate systems over LoRaWAN networks. At first, the focus has been on the improvement and the extension of the DingNet simulator, by refining its model and adding new features. After that, the support to simulate aggregate systems inside it has been designed. During this phase the focus has been on producing a platform that allows to move the simulated system to a real deployment with a low effort. Finally, two different case studies have been illustrated as a proof of concept of all the improvements and extensions introduced. A demo of the first case study was showed during the Day of Science in Flanders to introduce the LoRaWAN technology receiving a lot of attention. The second case study was actually evaluated only in a qualitative way, but further evaluations of quantitative nature are under investigation and they will be the subject of a future publication. In the end it is possible to say that the achieved platform enables the simulation of an aggregate system over a LoRaWAN network, and allows to move it in a real deploy with a low effort. To do so only two activities are necessary:

1. change the time conception used from the Protelis nodes, from the simulator time to the time of the real device that hosts the node;

2. change the MQTT client implementation (if a mock one is in use) and the MQTT broker address to subscribe to receive the mote packets.

So the entire aggregate system developed for the simulation (Protelis backend and program) does not require any changing.

4.2 Future work

Several improvements and interesting topics in different areas are available starting from the work illustrated in this thesis.

One area is to further improve the DingNet simulator. For example, it is possible to refine the model by simulating the loss of synchronism among the devices clock. It is possible to do so with different abstraction levels and required effort; starting with the application of a simple jitter to the clocks, arriving to introduce a stochastic model like a Markov chain.

Another area concerns the case study exposed in section 3.2. Here it is possible to evolve the program applying the techniques of machine learning to predict periods with high pollution and take countermeasures in advance. For example, it is possible to train a neural network that predicts the pollution level of the following days, considering the actual situation and the weather forecast.

Finally, an interesting topic deals with an extension of the thesis subject. This thesis is focused on joining the aggregate computing with the communication technology LoRaWAN, and provide a platform to simulate this kind of systems. But real complex pervasive scenarios do not use usually only one communication technology. So, to support aggregate applications in real use case scenarios, a middleware is needed to fill the network abstraction gap introduced by the paradigm. Consequently, the future work is to find an existing middleware that can fill this abstraction gap complying with the aggregate computing requirements. For example, Sentilo could be an interesting starting solution.

Bibliography

[1] LoRa Alliance. What is lorawan, jan 2020. https://lora-alliance.org/sites/default/files/2018-04/what-is-lorawan.pdf.

[2] Giorgio Audrito, Mirko Viroli, Ferruccio Damiani, Danilo Pianini, and Jacob Beal. A higher-order calculus of computational fields. *ACM Transactions on Computational Logic*, 20(1):1–55, January 2019.

[3] Jacob Beal and Jonathan Bachrach. Infrastructure for engineered emergence on sensor/actuator networks. *Intelligent Systems, IEEE*, 21:10 – 19, 04 2006.

[4] Jacob Beal, Danilo Pianini, and Mirko Viroli. Aggregate programming for the internet of things. *IEEE Computer*, 48(9):22–30, 2015.

[5] Jacob Beal and Mirko Viroli. Building blocks for aggregate programming of self-organising applications. *Proceedings - 2014 IEEE 8th International Conference on Self-Adaptive and Self-Organizing Systems Workshops, SASOW 2014*, pages 8–13, 03 2015.

[6] Roberto Casadei, Giancarlo Fortino, Danilo Pianini, Wilma Russo, Claudio Savaglio, and Mirko Viroli. Modelling and simulation of opportunistic IoT services with aggregate computing. *Future Generation Computer Systems*, sep 2018.

[7] Roberto Casadei, Danilo Pianini, Mirko Viroli, and Antonio Natali. Self-organising coordination regions: A pattern for edge computing. In *Lecture Notes in Computer Science*, pages 182–199. Springer International Publishing, 2019.

[8] Roberto Casadei, Mirko Viroli, Giorgio Audrito, Danilo Pianini, and Ferruccio Damiani. Aggregate processes in field calculus. In *Lecture Notes in Computer Science*, pages 200–217. Springer International Publishing, 2019.

[9] ChirpStack. Chirpstack architecture, jan 2020. https://www.chirpstack.io/img/graphs/architecture.png.

[10] Ferruccio Damiani, Mirko Viroli, and Jacob Beal. A type-sound calculus of computational fields. *Science of Computer Programming*, 117, 12 2015.

[11] Alexandru Lavric. LoRa (long-range) high-density sensors for internet of things. *Journal of Sensors*, 2019:1–9, February 2019.

[12] The Things Network. Lora network architecture, jan 2020. https://www.thethingsnetwork.org/docs/network/.

[13] Danilo Pianini, Ahmed Elzanaty, Andrea Giorgetti, and Marco Chiani. Emerging distributed programming paradigm for cyber-physical systems over LoRaWANs. In *2018 IEEE Globecom Workshops (GC Wkshps)*. IEEE, December 2018.

[14] Danilo Pianini, Sara Montagna, and Mirko Viroli. Chemical-oriented simulation of computational systems with ALCHEMIST. *J. Simulation*, 7(3):202–215, 2013.

[15] Danilo Pianini, Mirko Viroli, and Jacob Beal. Protelis: practical aggregate programming. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*, pages 1846–1853, 2015.

[16] Michiel Provoost and Danny Weyns. Dingnet: A self-adaptive internet-of-things exemplar. pages 195–201, 05 2019.

[17] Usman Raza, Parag Kulkarni, and Mahesh Sooriyabandara. Low power wide area networks: An overview. *IEEE Communications Surveys & Tutorials*, 19(2):855–873, 2017.

[18] Mirko Viroli, Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. Engineering resilient collective adaptive systems by self-stabilisation. *CoRR*, abs/1711.08297, 2017.

[19] Mirko Viroli, Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. Engineering resilient collective adaptive systems by self-stabilisation. *ACM Transactions on Modeling and Computer Simulation*, 28(2):1–28, mar 2018.

[20] Mirko Viroli, Roberto Casadei, and Danilo Pianini. On execution platforms for large-scale aggregate computing. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing, UbiComp Adjunct 2016, Heidelberg, Germany, September 12-16, 2016*, pages 1321–1326, 2016.

[21] Mirko Viroli, Roberto Casadei, and Danilo Pianini. Simulating large-scale aggregate MASs with alchemist and scala. In *Proceedings of the 2016 Federated Conference on Computer Science and Information Systems*. IEEE, October 2016.